Purely Functional Implementation of Attribute Grammars

Zuiver Functionele Implementatie van Attributengrammatica’s

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. Dr. H.O. Voorma,
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 9 december 1999 des namiddags te 12:45 uur

door

João Alexandre Baptista Vieira Saraiva

geboren op 21 september 1967
te Póvoa de Lanhoso, Portugal
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Pro-gramming research and Algorithmics). The printing of this thesis was partially supported by the Fundação para a Ciência e a Tecnologia, Ministério da Ciência e da Tecnologia.
# Contents

Acknowledgements v

1 Introduction 1
   1.1 Language-based Environments - An Example 2
   1.2 Attribute Grammars 4
      1.2.1 Higher-Order Attribute Grammars 6
   1.3 Incremental Evaluation 7
   1.4 Attribute Grammars and Functional Programming 7
   1.5 Structure of the Thesis 9

2 Definitions and Notations 11
   2.1 Context-Free Grammars 11
      2.1.1 Concrete and Abstract Grammars 13
   2.2 Attribute Grammars 15
      2.2.1 Attributed and Decorated Trees 17
      2.2.2 Circularities 18
   2.3 Attribute Grammar Specification 20
      2.3.1 The BLOCK Language 20
      2.3.2 The BLOCK Context-Free Grammar 21
      2.3.3 The BLOCK Attribute Grammar 23
      2.3.4 Attribute Dependencies 29
   2.4 Attribute Grammar based Systems 30
      2.4.1 Shortcomings of Attribute Grammars 32
   2.5 Higher-Order Attribute Grammars 33
      2.5.1 Higher-Order Attribute Grammar Specification 36
      2.5.2 Reducing Higher-Order Attribute Grammars 38
   2.6 Parsing with Combinators 39
      2.6.1 Parsing the BLOCK Language 40

3 Multiple Traversal Functions 43
   3.1 Multiple Traversal Functional Programs 44
   3.2 Circular Programs 49
   3.3 Catamorphisms 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1</td>
<td>Deriving a Circular Program</td>
<td>52</td>
</tr>
<tr>
<td>3.4</td>
<td>Attribute Grammars in a Functional Setting</td>
<td>55</td>
</tr>
<tr>
<td>3.4.1</td>
<td>The Circularity Test</td>
<td>58</td>
</tr>
<tr>
<td>3.5</td>
<td>Attribute Grammars as Circular Programs</td>
<td>59</td>
</tr>
<tr>
<td>3.6</td>
<td>Attribute Grammars as Multiple Traversal Programs</td>
<td>61</td>
</tr>
<tr>
<td>3.6.1</td>
<td>L-Ordered Attribute Grammars</td>
<td>61</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Ordered Attribute Grammars</td>
<td>62</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Chained Scheduling Algorithm</td>
<td>65</td>
</tr>
<tr>
<td>3.6.4</td>
<td>The Visit-Sequence Paradigm</td>
<td>66</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Inter-Traversal Attribute Dependency</td>
<td>68</td>
</tr>
<tr>
<td>3.6.6</td>
<td>The Binding-Tree Paradigm</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>Deforested Attribute Evaluators</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Functional Multiple Traversal Attribute Evaluators</td>
<td>74</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Attribute Grammar Lifetime Analysis</td>
<td>74</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Visit-Tree based Attribute Evaluators</td>
<td>76</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Deforested Strict Attribute Evaluators</td>
<td>83</td>
</tr>
<tr>
<td>4.2</td>
<td>Deforestation of Lazy Attribute Evaluators</td>
<td>88</td>
</tr>
<tr>
<td>4.3</td>
<td>Deforestation of Higher-Order Attribute Grammars</td>
<td>89</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Evaluation of Higher-Order Attribute Grammars</td>
<td>90</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Deriving Deforested Evaluators for HAGs</td>
<td>93</td>
</tr>
<tr>
<td>4.4</td>
<td>Deforested Attribute Grammar System</td>
<td>96</td>
</tr>
<tr>
<td>4.5</td>
<td>Extending the Grammar</td>
<td>99</td>
</tr>
<tr>
<td>5</td>
<td>Generic Attribute Grammars</td>
<td>101</td>
</tr>
<tr>
<td>5.1</td>
<td>Modularity in Attribute Grammars</td>
<td>102</td>
</tr>
<tr>
<td>5.2</td>
<td>Generic Attribute Grammars</td>
<td>103</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Generic Attribute Grammar Specifications</td>
<td>106</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Circularities</td>
<td>109</td>
</tr>
<tr>
<td>5.3</td>
<td>Generic Attribute Evaluators</td>
<td>112</td>
</tr>
<tr>
<td>5.4</td>
<td>Semantic Compositionality</td>
<td>117</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Extending the Productions of a Non-terminal</td>
<td>117</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Composing genag Components</td>
<td>118</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Inter-Module Attribute Dependencies</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>Functional Incremental Attribute Evaluation</td>
<td>121</td>
</tr>
<tr>
<td>6.1</td>
<td>Incremental Attribute Evaluation</td>
<td>122</td>
</tr>
<tr>
<td>6.2</td>
<td>Visit-Function Memoization</td>
<td>123</td>
</tr>
<tr>
<td>6.2.1</td>
<td>The Visit-Function Memoization at Work</td>
<td>126</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Managing the Memo Tables</td>
<td>129</td>
</tr>
<tr>
<td>6.3</td>
<td>Incremental Evaluation of HAGs</td>
<td>130</td>
</tr>
<tr>
<td>6.4</td>
<td>Memoization of Visit-Tree based Attribute Evaluators</td>
<td>133</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Visit-Tree versus Binding-Tree Approach</td>
<td>137</td>
</tr>
</tbody>
</table>
6.5 Memoization of Deforested Attribute Evaluators ........................................... 139
6.6 Decorating Shared Call-Trees ........................................................................... 142
6.7 Projection of Attributes ................................................................................... 147
6.8 Classification Criterion for Memoing Visit-Functions ..................................... 149
6.9 Function Cache Updating Algorithms .............................................................. 151
   6.9.1 LRU based Purging Strategy ..................................................................... 151
   6.9.2 Obsolete based Purging Strategy .............................................................. 152
   6.9.3 ULE based Purging Strategy ................................................................... 153

7 LRC: A Generator for Purely Functional Language-based Systems .......................... 155
   7.1 The Architecture of LRC ............................................................................ 155
      7.1.1 The Higher-Order Attribute Grammar Processor ................................ 156
      7.1.2 The Visit-Function Generator ............................................................... 157
      7.1.3 The Runtime System ........................................................................... 160
   7.2 A Language-based Environment for the BLOCK Language ......................... 161
      7.2.1 Applications Developed using LRC ...................................................... 164
   7.3 Performance of the Incremental Engine ....................................................... 165
      7.3.1 Binding-Tree versus Visit-Tree Performance Results ......................... 165
      7.3.2 Projection of Attributes ...................................................................... 168

8 Conclusions ........................................................................................................... 171
   8.1 Future Work .................................................................................................. 172
   8.2 Final Remarks ............................................................................................... 172

Bibliography ............................................................................................................. 173

Index ......................................................................................................................... 183

Samenvatting ............................................................................................................ 187

Curriculum Vitae ...................................................................................................... 189
Acknowledgements

Four years ago, I took the difficult decision of leaving family, friends and colleagues to follow a Ph.D. program in Utrecht. Now, I see that decision as a fortunate one. Following a Ph.D. project in a foreign country was, and is, an extremely enriching experience, even in its difficult moments. Utrecht itself, so beautiful and cozy, has been one of the reasons why I felt lucky and thankful during this stay. But there were other encouraging things: the opportunity to work with excellent colleagues, to meet wonderful people and to make lifetime friends. In the end, the whole experience of being here, with these people, taught me many important things. Only a small part of them is expressed in the scientific outcome of this thesis. Although never mentioned in this book, many people contributed to its elaboration. I would like to use these lines to acknowledge every person and institution that contributed to its fulfilment.

First of all, I would like to thank Doaitse Swierstra and Matthijs Kuiper for giving me the precious opportunity to do my Ph.D. in Utrecht, on leave from University of Minho, Portugal. In the last four years, Doaitse and Matthijs supervised my research in the perfect way: giving me the freedom I wanted, while steering me in the right direction. I thank Doaitse Swierstra for being a stimulating supervisor, always full of new ideas, enthusiasm and energy. Doaitse provided me with many useful contacts which have been very important not only for me but also for my university. I also thank Doaitse for the help in writing the *samenvatting* of the thesis. Matthijs has been a great friend since the very first day I arrived in Utrecht. I thank him for the encouragement and countless fruitful discussions about our research and the LRC system. Even after leaving the Academia, Matthijs was always “there”, ready to discuss my research. It has been a privilege to work with both of you.

My stay at Utrecht University would not be possible without the support of the Department of Informatics of University of Minho. I thank my colleagues who always supported this project and made it possible for me to stay in Utrecht for (almost) four years. I especially would like to thank Pedro Henriques who, since the first moment, believed in me and in this project. Pedro gave me the motivation and strength needed to accept it and see it through. Thank you for the continuous encouragement, support and friendship.

I also want to thank the members of my reading committee, in alphabetical order: Prof. Dr. Pedro Henriques, Prof. Dr. Rinus Plasmeyer, Dr. Gert v.d. Steen, Dr. Eelco Visser and Prof. Dr. Reinhard Wilhelm, for reviewing this thesis.

Several institutions contributed to this thesis in different ways. The research was par-
tially supported by the Programme Prodep II. The Department of Computer Science of Utrecht University, the Training and Mobility of Researchers (TMR) Programme of the European Community, the Fundação Luso-Americana para o Desenvolvimento and the Fundação Luso-Brasileira provided financial support to participate in workshops and conferences. I also want to thank Philips Research and Dr. Joachim Trescher for receiving Rui and Dario, my first undergraduate students, and co-supervise their final assignments at Philips Research.

Utrecht is a city filled with cultural events, students, great cafés, bicycles, canals, flowers and other captivating details which mark its particular personality. It was a wonderful experience to live here. However, the most important memories I will take with me to Portugal are the special moments I shared with the friends I made. It is difficult to express in words my gratitude to Francisco, Dora, Cristina and Eduardo, Annemieke and Luc, Pablo, Bram, Robert-Paul, Rosa and Sérgio, Briggite and Arjan, Rui, and Edwin. I just want to say ”thank you very, very much”.

I also would like to thank the group of friends in Braga. Thanks for the postcards, the emails, the phone calls, the visits and all the great moments we spent together in the (always) few days I’ve been back “home”. A special thanks to Filipe, Herculano, Jaime and Raquel, Mana, and Rui who, since the secondary school, always made my (student) life even more enjoyable. To them, and to the unnamed ones, “muito obrigado”.

These acknowledgements would be incomplete without mentioning my family. I want to thank my sister Raquel and my brother Miguel for the visits, the countless phone calls, the emails and the belief in my abilities. Like the rest of the family, they have always been supportive, available and committed when I needed them the most.

In Utrecht, I was lucky enough to be joined by Joana. There are no words to express what Joana is to me and how she is present in this thesis. Thank you for always being there.

Finally, to my parents: Ao longo da vida tive a sorte de encontrar professores excelentes e fazer amigos extraordinários. Gostaria de terminar em Português agradecendo aos dois melhores professores e amigos, a mamã, Maria Estela, e o papá, Bernardino Saraiva. Sem os seus ensinamentos, o seu amor, a sua força, a sua confiança nas minhas capacidades e todo o seu apoio esta tese nunca teria sido possível. Especialmente aos dois as minhas desculpas por me ter ausentado por tanto tempo.
Chapter 1

Introduction

Languages are the mechanism that links people and computers. Throughout their history, languages have always changed according to the needs of their users. Programming languages are no exception. Recent developments in modern programming languages provide powerful mechanisms, like modularity and polymorphic type systems, which allow us to structure programs and abstract from computations, thus changing the way we construct programs and, most importantly, improving our productivity. Programs are now easier to write, to reuse, and to manipulate.

Developments in programming languages are also changing the way in which we construct such programs: naive text editors are now replaced by powerful programming language environments which are specialized for the programming language under consideration and which help the user throughout the editing process. Helpful features like highlighting keywords of the language or maintaining a (predefined) beautified indentation of the program being edited are now provided by several text editors. Language-based environments, however, go a step ahead. They use knowledge of the programming language to provide the users with more powerful mechanisms to develop their programs. This knowledge is based on the structure and the meaning of the language. To be more precise, it is based on the syntactic and (static) semantic characteristics of the language. Having this knowledge about a language, the language-based environment is not only able to highlight keywords and beautify programs, but it can also detect features of the programs being edited that, for example, violate the properties of the underlying language. Furthermore, a language-based environment may also give information to the user about properties of the program under consideration. Finally, it may hide from the user some peculiarities included in the language that are relevant only for the programs/computers that process/execute them. All these features make a language-based environment an effective mechanism to increase the productivity of users.

This thesis concerns the formal specification and efficient implementation of language-based environments. Attribute grammars are the formalism we will use to specify such environments. Purely functional attribute evaluators are the implementations we derive from attribute grammars. We introduce two new purely functional implementations for attribute
Chapter 1. Introduction

Grammars. The efficient implementation of language-based environments is achieved by using incremental evaluation. This thesis presents several techniques for attribute grammars aiming at improving the performance of their implementations. These techniques, and a generator for our purely functional attribute evaluators, have been implemented. This thesis also addresses the problem of modularity and reusability in attribute grammars. An extension to the classic attribute grammar formalism that provides genericity, modularity and reusability is proposed.

1.1 Language-based Environments - An Example

We shall consider now a motivating example: the Bib\TeX language-based environment. The Bib\TeX language is a bibliographic database definition language. A text, or to be more precise, a database of Bib\TeX items, must obey the rules of this language. We will not bother the reader with a detailed description of the rules of the language. We have two reasons for this: firstly, they are not relevant for our presentation, and secondly, and most importantly, the reader does not need to know such rules to be able to use/understand the Bib\TeX language-based environment.

For the reader who is not familiar with bibliographic databases at all, let us briefly explain what a bibliographic database is. Basically, it is a collection of bibliographic entries (we will call them bib-entries). Each entry consists of a type, a citation key and a list of fields. The type (the bib-type) specifies whether the entry is a reference to a book, or to an article, etc. The citation key is a unique name that identifies the bib-entry and that makes citations possible. The list of fields specifies the author, title, etc. Different fields are usually required for different types of bibliographic entries. The Bib\TeX language is no exception. Figure 1.1 shows the Bib\TeX environment. A bibliographic entry is displayed at the center.

The user of the Bib\TeX environment does not need to be an expert in the Bib\TeX language to maintain her/his bibliographic database. For example, the user does not need to know that, in Bib\TeX, an entry-type name is preceded by an @ character, or that the required fields for the entry-type article are the author, title, journal and year. The Bib\TeX environment presents this information to the user. To add a new entry the user has to press the button Add Entry and to choose the type of reference he wants to include in the database. The environment informs the user of the list of all possible bib-type entries. After that, the Bib\TeX environment automatically displays a new Bib Entry window which is adjusted to the type of the entry concerned. It displays, for example, the fields required for that particular entry, i.e., the fields the user has to fill in. Besides that, the environment also provides immediate feedback to the user concerning the consistency of the database. If it is the case that, for instance, the user defines a key that is already defined in the database, then a warning message is immediately displayed. Consequently, a language-based environment guides the user in writing correct programs.

The Bib\TeX environment also provides facilities for more traditional editing. For example, the text presented in window Bib\TeX Language is a (beautified) textual representation
Figure 1.1: The BibTeX language-based environment. It consists of: the Main Window (left) that allows the user to browse the database and edit, add and delete bib-entries. The Bib Entry (center) that displays the selected entry and allows the user to update it. The BibTeX Language (right) that presents the textual BibTeX representation of the database and which provides a more traditional editor.

of the database. The user can directly browse or update the database in that textual representation, as well. In this case, knowledge about the syntax characteristics of the language is required. Another interesting feature of the BibTeX environment is the fact that it displays the fields in a consistent way. By consistent we mean that there is a pre-defined order of display for the fields, and, for each bib-type that order is respected, independently of the order in which they occur in the textual database.

After each interaction with the user, the BibTeX language-based environment immediately adjusts its interface (if needed), and also provides immediate feedback concerning whether or not such an interaction is legal. That is to say, the environment has to react and to provide answers in real-time. Consequently, the delay between the user interaction and the system response is an extremely important aspect in such interactive systems.
1.2 Attribute Grammars

Language-based environments deal with formal languages, i.e., languages that can be defined by a formal method. In order to construct a language-based environment for a particular language, we need a formal method to define languages. Obviously, we cannot define a language by just enumerating the set of all its valid sentences, since most languages have an infinite number of them (e.g., BibTeX). Consequently, we need a finite formalism to defined the non-finite set of valid sentences. *Attribute grammar* is such a formalism.

An attribute grammar consists of a context-free grammar, and a set of attributes and attribute equations. The context-free grammar of a language specifies the (finite) set of symbols of the alphabet, and defines which sequences of those symbols form a syntactically valid sentence. On the other hand, the set of attributes and attribute equations describe semantic properties of the language. The static semantics of the language is specified by establishing conditions on the attributes. In other words, the context-free grammar defines the structure of the language while the attributes and their equations define the meaning of the language.

We informally illustrate attribute grammars with the help of the BibTeX language-based environment. We shall start by defining the set of syntactically valid sentences of BibTeX. A sentence in BibTeX is a collection of items, which we call entries. So, in terms of context-free grammars we will express this in the following production:

\[ \text{BibTeX} \rightarrow \text{Entries} \]

where BibTeX and Entries are called non-terminal symbols. The non-terminal symbol Entries denotes all possible lists of entries that may occur in BibTeX. A list of entries is one bib-entry followed by more entries. This might be expressed in the following production:

\[ \text{Entries} \rightarrow \text{BibEntry Entries} \]

or it is an empty list of entries, which we represent by the symbol \( \epsilon \) and we write:

\[ \text{Entries} \rightarrow \epsilon \]

As we have said before, a bib-entry consists of three parts: an entry-type, a citation key and the list of fields. These parts are delimited by punctuation symbols, according to the syntactic rules of BibTeX. We define a bib-entry by the following production:

\[ \text{BibEntry} \rightarrow '@' \text{BibType} '{' \text{CitationKey Fields} '} \]

Let us analyse in more detail this production. The symbols surrounded by quotes are symbols of the vocabulary of BibTeX. They must occur literally in a bib-entry. Thus, this production describes which sequence of symbols constitutes a syntactically valid entry. In this case, the sequence starts with the symbol '@' followed by a sequence of symbols forming a syntactically correct bib-type expression, as well. Furthermore, the remaining symbols must start with symbol '{', which must be followed by symbols that define a valid citation key and a list of fields. Finally, the last symbol of the bib-entry must be the symbol '}'.

The productions define a hierarchical tree structure for the language. For the reader who is familiar with attribute grammars, the window *BibTeX Language* in Figure 1.1...
just shows a “pretty printed” representation of such a structured tree for the particular bibliographic database under consideration.

We proceed now to extend the definition of \texttt{BibTeX} with attributes. We focus on a particular semantic aspect of the \texttt{BibTeX} language: the consistency of cross-references in the database. A cross-reference is a citation inside the database, \textit{i.e.}, in \texttt{BibTeX} a bib-entry may refer to a citation key of another bib-entry. For example, in Figure 1.1 the article where the citation key is \texttt{KS98}, refers to the proceedings \texttt{CC98} in which it is published. When constructing an environment for \texttt{BibTeX}, we aim at informing the user whenever a cross-reference refers to a citation key that is not included in the database under consideration. Furthermore, we wish to impose no restrictions in the language-base environment regarding whether the cross-referenced entry must occur after or before any entry that refers to it.

To maintain such consistency of the database we introduce attributes. First, we associate the attribute \texttt{collectedKeys} to the bib-entries. This attribute is used to collect all the citation-keys defined in the database. We associate attribute equations with the productions that specify how the occurrences of attributes are defined. For example, the attribute equation below specifies that the list of collected keys of the father symbol consists of the key of the bib-entry “plus” the keys collected in the remaining entries, \textit{i.e.}, its second child. To distinguish different occurrences of a non-terminal symbol, we labelled them with their occurrence number.

\[
\text{Entries}_1 \rightarrow \text{BibEntry} \text{Entries}_2 \\
\text{Entries}_1.\text{collectedKeys} = \text{cons} \text{BibEntry.key} \text{Entries}_2.\text{collectedKeys}
\]

where the so-called semantic function \texttt{cons} adds an element (a key, in this case) to the front of a list (of keys). The attribute \texttt{key} associated to a bib-entry refers to its citation key. Observe that this rule moves the list of keys from the right into the left hand side of the production. In terms of trees, we are moving (context) information \textit{up} in the tree. For this reason, the two previous attributes are called \textit{synthesized attributes}.

Once we have collected all the citation keys, we can easily detect which entries refer to non-existent keys: we just “move” the collected list of keys to the context where cross-references may occur (\textit{i.e.}, to the context of a bib-entry). After that, we check whether or not the entry refers to a key that is in the collected list of keys.

In order to pass the collected list of keys to the context of a bib-entry, we augment the grammar with the attribute \texttt{availableKeys}. Now, we have to define that the available keys in the database are the keys synthesized in attribute \texttt{collectedKeys}. We formalise this rule in the next attribute equation:

\[
\text{BibTex} \rightarrow \text{Entries} \\
\text{Entries.availableKeys} = \text{Entries.collectedKeys}
\]

That is to say, we move the collected list of keys, \textit{i.e.}, the context information, \textit{down} in the tree. Attributes that propagate context information down in the tree are called \textit{inherited attributes}. Finally, we are ready to define that a cross-reference in one field must be in the set of available keys. In terms of attribute grammars, this condition is formulated as follows:
The semantic function `mustBeIn` is written in infix notation to stress the analogy with our informal description.

One key property of attribute grammars is the fact that they are executable. That is, from attribute grammars, the so-called attribute evaluators can be automatically constructed. An attribute evaluator is a function that takes as argument a tree, assigned by the context-free grammar, and computes the values of the instances of attributes which are associated to the nodes of the tree. This process is known as attribute evaluation or decoration. Thus, languages can be specified by attribute grammars and language-based tools can be automatically obtained from their attribute grammar specification. The BibTEX language-based environment is one example of such a tool.

### 1.2.1 Higher-Order Attribute Grammars

Attribute grammars are a suitable formalism to describe formal languages. Classical attribute grammars have a severe drawback: every computation has to be expressed in terms of the tree structure defined by the context-free grammar under consideration. However, there are several semantic properties of a language that are not easily expressible by induction on that particular tree structure. Under the classic formalism it is not possible to define first a better suited tree structure, since the tree is fixed during attribute evaluation.

To show this limitation of attribute grammars, let us consider a second semantic aspect of the BibTEX environment. In this environment, the user can select bib-entries to be displayed. Each entry is displayed in a Bib Entry window that is specialized for the type of the selected entry. As we have explained before, there is an order of display for the fields of the entry, which only depends on the type of the entry, and not on the order in which the fields occur in the database. This is a typical example, where a clear and elegant solution is obtained if, in the first place, we define the structure of display of the fields and only then decorate such structure with the fields of the selected entry.

In this case, a question arises: how can we provide a clear and elegant solution for such a semantic property within the attribute grammar formalism? The answer is to use higher-order attribute grammars. Higher-order attribute grammars augment classic attribute grammars with higher-order attributes. A higher-order attribute is an attribute, called attributable attribute, whose value is a tree with which we associate attributes again.

Next, we present the fragment of the higher-order attribute grammar that answers the previous question. First, we declare an attributable attribute named `windowEntry` of type `BibEntry`. That is, we define a (higher-order) attribute whose values are trees: the (sub)trees that represent bib-entries. Next, the attributable attribute `windowEntry` is instantiated with the tree that defines the required tree structure. This structure is defined by the semantic function `createEntryWindow` based on the type of the entry. After that, the tree is decorated with the fields of the entry concerned.
It is worthwhile to note that the attributable attribute will be constructed and decorated during attribute evaluation. Thus, the interface of BibTeX will be adjusted immediately as the result of decorating the higher-order attribute. This characteristic makes higher-order attribute grammars particularly suitable to model interactive environments.

### 1.3 Incremental Evaluation

Higher-order attribute grammars provide clear and elegant solutions to specify language-based environments. Moreover, attribute grammars and their higher-order variant are executable, and, thus, implementations for such environments can be automatically obtained.

One of the key features to handle interactive environments is the ability to perform efficient re-computations after each interaction with the user. Under these environments, each interaction typically has a small impact in the overall re-computation of the system, i.e., most of the computations induced by the interactions are, actually, computations performed previously. Consequently, an efficient implementation of such an environment aims at reusing “old” computations in order to avoid as much re-computation as possible. In other words, language-based environments rely heavily on incremental evaluation.

Let us be more precise about incremental evaluation. Let $e$ be an attribute evaluator and $t$ the tree representation of the input being edited by a language-based environment. Under an incremental model of evaluation, after a user interaction $t$ is modified to $t'$, and an immediate response of the system is provided by evaluating $e(t')$. Because the difference from $t$ to $t'$ is often small, the evaluation of $e(t)$ and $e(t')$ is also small. An evaluator that uses information of $e(t)$ to compute $e(t')$ is called an incremental evaluator.

Incremental evaluation can be efficiently obtained by using memoized functions. A memoized function is a pure function that “remembers” the arguments to which it has been applied, together with the results. Consequently, an incremental evaluator can be obtained by deriving purely functional implementations of attribute grammars and by using function memoization.

### 1.4 Attribute Grammars and Functional Programming

Attribute grammars and functional programming are closely related. In both styles of programming, the programmers typically structure their programs base on the processing of a language by recursion over an abstract representation of the language, i.e., an abstract syntax tree. For each production in the abstract syntax, both styles define a function that specifies how a construct is to be translated. Functional programming is known to yield concise and clear solutions. Occasionally, however, functional programming may yield large and complex programs, where attribute grammars have obvious and simple solutions.
Let us present again the attribute equation that formalizes the rule defining that the keys available in the database are the keys collected in attribute \textit{collectedKeys}.

\begin{verbatim}
BibTex \to Entries
Entries.availableKeys = Entries.collectedKeys
\end{verbatim}

This is a typical example where an inherited attribute (\textit{availableKeys}) depends on a synthesized attribute (\textit{collectedKeys}) of the same non-terminal symbol (\textit{Entries}). Although such dependencies are natural in attribute grammars, they may lead to complex and counterintuitive (functional) solutions.

For example, this straightforward functional solution will require two traversals over the tree structure defined by non-terminal \textit{Entries} (recall that grammars define trees). A first traversal synthesizes the list of keys, and a second one uses this list to collect the entries that refer to non-available keys. A problem, however, arises when intermediate values computed in the first traversal function are used in the second one. In this case, an explicit intermediate data structure has to be defined to “glue” together the traversal functions. For complex problems, where a possibly large number of traversal functions is required, the functional programmer would have to write a large number of such “gluing” data structures. Furthermore, in such cases it would be extremely difficult to define that “glue” and to connect all the traversal functions.

A more elegant solution can be obtained in a lazily evaluated functional language. Under lazy evaluation, the so-called circular programs can be used, and they eliminate the need to glue different traversal functions. Circular programs simply merge the traversal functions into a single “circular” functional definition. As a result, the previous two traversal functions can be expressed as follows:

\begin{verbatim}
(collectedKeys, invalidrefs) = evalEntries entries collectedKeys
\end{verbatim}

Note the counterintuitive circular definition: \textit{collectedKeys} is both an argument and a result of the same function call. Although the circular definition seems to induce both cycles and non-termination of those programs, the fact is that the lazy evaluation machinery may still be able to determine, at runtime, the right order to evaluate circular definitions.

Thus, in attribute grammars and lazy functional circular programs, the programmer needs not to concern himself with partitioning the program into a number of different traversals. In the attribute grammar formalism, the order of computation is derived automatically after the data dependencies induced by the attribute equations. In a lazily evaluated language that ordering is determined at runtime by the data dependencies, on demand by the lazy evaluation machinery. Furthermore, in both styles of programming no “gluing” data structures have to be defined. On the other hand, the two styles of programming differ in one key aspect: while the writer of an attribute grammar does not have to concern himself with the existence of cycles induced by his attribute definitions, confident that well-known attribute grammar techniques (statically) detect them for him, the functional programmer has to guarantee himself that no cycles may occur when his program is executed. This important property of attribute grammars ensures the correct termination of attribute grammars when their implementations are executed on some unknown input.
1.5 Structure of the Thesis

Consequently, such functional programs are much easier to write and to understand if we consider them as the representation of an attribute grammar within a functional programming language. This thesis addresses this problem, i.e., the representation of attribute grammars in a purely functional setting.

1.5 Structure of the Thesis

This thesis is organized as follows: First, in Chapter 2, some preliminary definitions and notations which will be used throughout this thesis will be given. A motivating example will be introduced and an attribute grammar [Knu68] and its higher-order variant [VSK89, SV91] are defined and discussed in great detail. This example will be used as the running example in several chapters. In Chapter 3, we discuss multiple traversal functional programs. We shall consider two approaches to write such programs. In the first place, we present the purely functional approach, where such programs are expressed either by standard strict, multiple traversal functions, or by lazy circular programs [Bir84b]. Secondly, we present the attribute grammar approach. We also present the visit-sequence paradigm [Kas80] and the binding-tree based evaluators [Pen94]. In Chapter 4 we introduce two new purely functional mappings for attribute grammars. The visit-tree based attribute evaluators [SKS97b, SSK99] and the deforested attribute evaluators [SS99a] are formally defined. The former uses the so-called visit-trees to “glue” together different traversal functions, while the latter is a data type free functional evaluator. The absence of any explicit data type definition makes the deforested attribute evaluators highly modular and reusable. This is the subject addressed in Chapter 5. We introduce generic attribute grammars that provide genericity, reusability and modularity in the context of attribute grammars based systems [SS99a]. After that, in Chapter 6 we present functional incremental attribute evaluation. Incremental evaluation is achieved via function memoization [PSV92]. We describe the incremental behaviour of the visit-tree based evaluators and we compare it with the binding-tree approach. A new approach for the memoization of deforested evaluators is proposed. Furthermore, we present several techniques aimed at improving the incremental performance of functional implementation of attribute grammars. Finally, several algorithms to discard memoized functions from the memoization table are discussed and proposed [SKS97a]. As part of our research, we have implemented our techniques in the LRC system [KS98], a generator for purely functional language-based systems. In Chapter 7 we present the architecture of the system and some experimental results. We will also specify a language-based environment for our running example. Finally, in Chapter 8 we draw some conclusions concerning the present techniques, and we indicate possible directions for future research.
Chapter 1. Introduction
Chapter 2
Definitions and Notations

Summary

This chapter presents the definitions and the notations that will be used throughout the thesis. Context-free grammars, attribute grammars, higher-order attribute grammars are defined. A motivating attribute grammar, its higher-order variant and its combinator parser are presented and analysed in great detail.

The purpose of this chapter is to introduce some definitions, notations and examples that will be used throughout this thesis. We formally define context-free grammars, attribute grammar, higher-order attribute grammars and we present parsing with combinators. Each of these formalisms is used to define formal languages, i.e., languages whose syntax and semantics can be formally specified. To define such grammar specifications we need a notation. We use a “standard” attribute grammar notation. To define the semantic equations and the semantic functions we decided to use a notation based on the standard functional programming language HASKELL [JHA+99]. The reasons for this choice are the following: firstly, a functional language is the standard notation to express the semantic equations within the attribute grammar formalism and, in this thesis, it is also the target language for their implementations; secondly, it is an elegant and concise notation to express such specifications, and to define their implementations. Furthermore, there are several good books [Bir98, Tho99] and tutorials [HPF97, BA98] for HASKELL. Consequently, we omit its definition in this thesis. Ordinary mathematics are also expressed in a purely functional style.

2.1 Context-Free Grammars

Attribute grammars form an extension of the context-free grammar formalism [ASU86].
Definition 2.1 (Context-Free Grammar) A context-free grammar (CFG) is a triple $G = \langle V, P, S \rangle$. $V$ is the vocabulary, a finite non-empty set of grammar symbols. $\Sigma \subset V$ is the alphabet, that is the set of terminal symbols or terminals. $N = V - \Sigma$ is the non-empty set of non-terminal symbols or non-terminals. $P \subseteq N \times V^*$ is a finite set of productions. $S \in N$ is the start symbol or axiom.

A production $p \in P$ is denoted by $p : X_0 \rightarrow X_1 X_2 \ldots X_n$. $X_0$ is the left-hand side symbol of $p$ and it is denoted by $lhs(p)$. $X_1, X_2, \ldots, X_n$ are the right-hand side symbols of $p$, denoted by $rhs(p)$. A pair $\langle p, i \rangle$, with $0 \leq i \leq n$ is called an occurrence of grammar symbol $X_i \in V$. We define the number of right-hand side symbols of $p$ as $|p|$. We say that $p$ is a production applied on $X$ if and only if $\langle p, 0 \rangle = X$. Furthermore, we say that a production $p$ is a terminal production if it has no non-terminal symbols on its right-hand side. In other words, for all $X \in rhs(p)$, $X \in \Sigma$. An empty right-hand side is represented by the symbol $\epsilon$. A production with a single grammar symbol, i.e., a production of the form $A \rightarrow \epsilon$, is called $\epsilon$-production.

The notation introduced for productions stresses the rewriting nature of the context-free grammars: the left-hand side non-terminal can be rewritten to its right-hand side. This process can be used to derive possible sequences of grammar symbols for a CFG. We define the relation $\Rightarrow$, called directly derives, as follows: for any $\alpha \in V^+$ and $\beta \in V^*$, $\alpha \Rightarrow \beta$ if and only if $\alpha = \alpha_1 A \alpha_2$, $\beta = \alpha_1 \gamma \alpha_2$, and $A \rightarrow \gamma \in P$, with $\alpha_1, \alpha_2 \in V^*$. The transitive and reflexive closures of $\Rightarrow$, denoted by $\Rightarrow^+$ and $\Rightarrow^*$, are defined as usual.

The language generated by $G$, denoted by $L(G)$, is the set of sequences of terminal symbols that can be derived by rewriting the start symbol $S$. Formally, it is defined as follows:

$$L(G) = \{ \mu \in \Sigma^* \mid S \Rightarrow^* \mu \}$$

We say that $G$ generates a sequence $s$ if $s \in L(G)$, and $s$ is called a sentence generated by $G$. Note that a context-free grammar can derive sequences of terminal symbols, if terminal productions exist. We say that a symbol $X \in V$ is accessible or derivable from $Y \in N$ if there is a derivation of the form $Y \Rightarrow^* \alpha_1 X \alpha_2$, with $\alpha_1, \alpha_2 \in V^*$. If a single sequence of derivations exists for every $s \in L(G)$, we say that the grammar is unambiguous, otherwise, it is ambiguous. Two context-free grammars $G_1$ and $G_2$ are equivalent and we write $G_1 \equiv G_2$ if and only if $L(G_1) = L(G_2)$.

A context-free grammar is assumed to be complete, i.e., every symbol is accessible from the start symbol and every non-terminal can derive a sequence of only terminal symbols.
2.1. Context-Free Grammars

Definition 2.2 (Complete Context-Free Grammar) A context-free grammar $G = \langle V, P, S \rangle$ is a complete context-free grammar (CCFG) if and only if:

$$\forall X \in V \exists \mu, \nu \in V^* \land \delta \in \Sigma^*: S \Rightarrow^* \mu X \nu \Rightarrow^* \delta$$

As usual in context-free grammars, we distinguish two classes of terminal symbols $\Sigma = L \cup \Gamma$: the set of literal symbols $L$ and the set pseudo-terminal symbols $\Gamma$. The former consists of the symbols of the alphabet that do not play a role in the semantics of the underlying language. Language keywords and punctuation symbols are examples of literal symbols. Pseudo-terminal symbols are non-terminal symbols for which the productions are implicit, traditionally expressed as regular expressions [ASU86] and provided by an external lexical analyser. Examples of such symbols are integers, strings and identifiers of the language.

Non-terminal and pseudo-terminal symbols induce a set of values, i.e., terms. They will therefore be also regarded as types. We define the function $T_G : (N \cup \Gamma) \to T$, where $T$ is a set of sets (types) and for all $X \in (N \cup \Gamma)$, $(T_G X) \in T$ is the set of all possible values of $X$. Like Pennings [Pen94], we introduce a second notation for productions which capture their term-like nature. A production $p \in P$ is also denoted as $X_0 = p X_1 \ldots X_n$, where the name of the production, i.e. $p$, also indicates the term constructor function $p$. The type of the constructor function is $p : T_G X_1 \to \cdots \to T_G X_n \to T_G X_0$. Roughly speaking, non-terminal symbols correspond to tree type constructors, and productions correspond to value constructors. Note that the symbol $p$ is overloaded: it denotes the entire production “$p \in P$”, its name “$p : \ldots$”, and the term constructor function “$p \cdots$”.

2.1.1 Concrete and Abstract Grammars

Context-free grammars specify syntactic characteristics of languages. We shall distinguish two classes of context-free grammars: the concrete context-free grammar and the abstract context-free grammar.

A concrete context-free grammar $G$, which we shall abbreviate as concrete grammar, defines the set $L(G)$ of valid sentences of the language. That is, it specifies how the vocabulary symbols $V$ of $G$ may form a syntactic valid sentence of the language $L(G)$. Generally, a concrete grammar aims at being easily used to derive sentences of the language under consideration. Grammars are also used to directly guide the construction of programs, known as parsers, that check whether a stream of symbols is a sentence of the language or not. Parsers usually perform other actions, known as semantic actions, while checking the stream of symbols. For example, they build an abstract representation of the stream. Such an abstract representation can be the final result of its processing or it can be an intermediate data structure which is used for further processing, e.g., for semantic analysis. Usually that intermediate data structure is a tree. An unambiguous concrete context-free
grammar $G$ defines a unique concrete syntax tree for every sentence of $L(G)$.

**Definition 2.3 (Concrete Syntax Tree)** A concrete syntax tree, also called derivation tree or parse tree, for a sentence $s$, generated by a complete context-free grammar $G = \langle V, P, S \rangle$, is a term, where

- Each node is labelled by a symbol $X \in V \cup \{\epsilon\}$;
- If a node labelled by $X_0$ has the children labelled by $X_1, X_2, \ldots, X_n$, then, there is a production $p : X_0 \rightarrow X_1 X_2 \ldots X_n \in P$;
- The label of the root of the term is $S$;
- The labels of the leaves of the term, concatenated from left to right, form $s$.

The parsing techniques used by a parser may impose restrictions on the concrete context-free grammar. For example, efficient parsing techniques as $LL(1)$ and $LR(1)$ require the grammar to be in a particular form [ASU86]. Consequently, different parsing techniques may require different concrete context-free grammars to define exactly the same language. In order to describe the semantics of the language, however, we wish to abstract from both the concrete syntactic characteristics and the parsing techniques of the language, and concentrate on its abstract structure only. An abstract context-free grammar (ACFG), or simply abstract grammar, describes precisely the abstract structure of the language. Furthermore, an ACFG is designed to provide a suitable structured representation of the language, which will be used as the underlying representation to describe the semantics of the language. Consequently, an abstract grammar includes only the symbols of the alphabet that play a role on the semantics of the language. Thus, we assume that literal symbols are included in the concrete grammar but not in the abstract one.

**Definition 2.4 (Abstract Syntax Tree)** An abstract syntax tree (AST) generated by a complete context-free grammar $G = \langle V, P, S \rangle$, is a term, where

- Each node is labelled by a production $p \in P$;
- Every node labelled by a $\epsilon$-production is a leaf.
- Every node labelled by $p$, with $X_0 = p X_1 \ldots X_n$, has $n$ children $T_1, \ldots, T_n$ (in that order), where $T_i$, with $1 \leq i \leq n$, is again an abstract syntax tree labelled with a production applied on $X_i$.

---

1Informally, we shall say that concrete grammars are for parsing and abstract grammars are for semantic analysis.
2.2 Attribute Grammars

A context-free grammar describes syntactic properties of a particular language. Attribute grammars extend this formalism in order to describe the semantic properties of such a language as well. An attribute grammar abstracts from the concrete and, instead, starts from the abstract context-free grammar of the language.

A context-free grammar is extended as follows: with every grammar symbol and every production, so-called attributes are associated. Each attribute describes a property of the language. The attributes of every (non-terminal) symbol are partitioned into two sets: inherited and synthesized attributes. For every occurrence of a grammar symbol in a production, there is a so-called attribute occurrence. With every production, three sets of attribute occurrences are assigned: the set of local attribute occurrences of the production, the set input occurrences, and the set of output occurrences. The set of input occurrences consists of all inherited attribute occurrences of the left-hand side symbol of the production and all the synthesized attribute occurrences of the symbols in the right-hand side of the production. Similarly, the set of output occurrences contains all the synthesized attribute occurrences of the left-hand side symbol and all the inherited attribute occurrences of the symbols in the right-hand side of the production. Then, with every output attribute occurrence and every local attribute an semantic equation is defined, specifying the value of that attribute occurrence in terms of other attribute occurrences.

Attribute grammars are declarative: values of attribute occurrences are defined in terms of each other. Ways of actually implementing evaluation are, at the most, suggested, but not prescribed. The following definition of attribute grammar is similar to the ones presented in [Kas80, Alb91b, Pen94] with slight differences in the adopted notation.

**Definition 2.5 (Attribute Grammar)** An attribute grammar \((AG)\) is a triple \(AG = \langle G, A, D \rangle\). \(G = \langle V, P, S \rangle\) is a context-free grammar. \(A\) is a finite set of attributes, partitioned into sets \(A_{\text{nont}}(X)\) and \(A_{\text{loc}}(p)\) for each \(X \in N\) and \(p \in P\). \(A_{\text{nont}}(X)\) are further partitioned into two disjoint subsets \(A_{\text{inh}}(X)\) and \(A_{\text{syn}}(X)\). \(D = \langle T, E \rangle\) is the semantic domain of \(AG\). \(T\) is a finite set of types and \(E\) is a finite set of semantic equations.

\(A\) and \(D\) together are known as the attribution rules of the attribute grammar. Informally, \(G\) specifies the (abstract) syntax of the (source) language and \(A\) and \(D\) specify the
(static) semantics of the language. Every attribute \( a \in A \) is associated with either a grammar symbol \( X \in N \) or a production \( p \in P \). \( A_{nont}(X) \) is the set of attributes associated with non-terminal \( X \). An element \( a \in A_{nont}(X) \) is denoted by \( X.a \), and it is either inherited if \( a \in A_{inh}(X) \) or synthesized if \( a \in A_{syn}(X) \). Attributes have a type. The function \( T : A \rightarrow T \) associates a type with every attribute. Thus, for every \( a \in A \), \((T a) \in T\) is the set of all possible values of \( a \).

The attributes \( a \in A_{loc}(p) \) are known as local local attributes of \( p \) and are denoted by \( p.a \). Each local attribute \( p.a \) induces a local attribute occurrence of \( p \). The set of all local attribute occurrences of \( p \) is \( O_{loc}(p) = A_{loc}(p) \). A production \( p \in P, p : X_0 \rightarrow X_1 \cdots X_n \), with \( n \geq 0 \), has an attribute occurrence \( \langle p, i, a \rangle \) if \( a \in A_{nont}(X_i) \), with \( 0 \leq i \leq n \). For every occurrence \( \langle p, i \rangle \) of a symbol \( X_i \in N \) in a production, sets of attribute occurrences are associated as follows:

\[
O_{inh}(\langle p, i \rangle) = \{ \langle p, i, a \rangle \mid a \in A_{inh}(X_i) \},
\]

\[
O_{syn}(\langle p, i \rangle) = \{ \langle p, i, a \rangle \mid a \in A_{syn}(X_i) \},
\]

\[
O_{nt}(\langle p, i \rangle) = O_{inh}(\langle p, i \rangle) \cup O_{syn}(\langle p, i \rangle).
\]

We define the set of all attribute occurrences associated with the non-terminal occurrences of \( p \) as \( O_{ntocc}(p) = \bigcup_{0 \leq i \leq |p|} O_{nt}(\langle p, i \rangle) \). The set of attribute occurrences of \( p \) is \( O_{pr}(p) = O_{loc}(p) \cup O_{ntocc}(p) \). Furthermore, the sets of input and output occurrences of a production \( p \) are defined as follows:

\[
O_{inp}(p) = O_{inh}(\text{lhs}(p)) \cup \bigcup_{1 \leq i \leq |p|} O_{syn}(\langle p, i \rangle),
\]

\[
O_{out}(p) = O_{syn}(\text{lhs}(p)) \cup \bigcup_{1 \leq i \leq |p|} O_{inh}(\langle p, i \rangle).
\]

The function \( T \) is overloaded to work on attribute occurrences, too. Thus, \( T \langle p, i, a \rangle = T a \).

Finally, we define the set of all attribute occurrences of \( AG \) as \( O_{AG} = \cup_{p \in P} O_{pr}(p) \).

For every production \( p \in P, p : X_0 \rightarrow X_1 \cdots X_n \), with \( n \geq 0 \), a set of semantic equations, also called attribute equations, \( E_p \) is associated with \( p \), which define the value of every attribute occurrence in \( O_{out}(p) \cup O_{loc}(p) \) in terms of other attribute occurrences in \( O_{pr}(p) \). Thus \( E_p \) is a set of equations of the form

\[
(\alpha_1, \ldots, \alpha_l) = f \; \beta_1 \beta_2 \cdots \beta_k
\]

where \( k \geq 0 \) and \( \alpha_i \in O_{out}(p) \cup O_{loc}(p) \). Every \( \beta_i \), with \( 1 \leq i \leq k \), is an attribute occurrence in \( p \), or it is an occurrence of a grammar symbol in \( p \). If it is a grammar symbol occurrence, that is \( \beta_i = \langle p, q \rangle, 1 \leq q \leq |p| \), we say that symbol \( \langle p, q \rangle \) is syntactically referenced in the semantic equations of \( AG \). Note that occurrences of non-terminal symbols in \( p \) may be directly used on the right-hand side of a semantic equation. This is known as a syntactic reference. Formally, the symbols that occur on the right-hand side of an equation are \( \beta_i \in O_{pr}(p) \cup \bigcup_{0 \leq q \leq |p|} \langle p, q \rangle \).

In this equation, \( f \) is a function, called semantic function, that maps the values of \( \beta_1, \ldots, \beta_k \) to the values of \( \alpha_1, \ldots, \alpha_l \). The type of \( f \) is \( t_1 \rightarrow \cdots \rightarrow t_l \rightarrow (t_j, \ldots, t_k) \),
\[ \forall_{1 \leq i \leq l}(T \beta_i) = t_i \text{ and } \forall_{1 \leq i \leq k}(T \alpha_i) = t_i. \] The type of function \( T \) is extended to work on grammar symbols too: \( T : A \cup N \cup \Gamma \to T \). We say that a semantic equation is a \textit{copy rule} if \( f = \text{id} \). We have \( E = \bigcup_{p \in P} E_p \).

We define the sets \( O_{def}(p) \) and \( O_{use}(p) \), which consist of all attribute occurrences that are \textit{defined} and \textit{used} in \( p \), as follows:

\[
O_{def}(p) = \{ \alpha \mid \{(\ldots, \alpha, \ldots) = f \ldots \beta \ldots \} \in E_p \} \\
O_{use}(p) = \{ \beta \mid \{(\ldots, \alpha, \ldots) = f \ldots \beta \ldots \} \in E_p \land \beta \in O_{pr}(p) \}
\]

The value of every local and output attribute occurrence of \( p \) must be defined by exactly one single semantic equation. Thus the following two conditions must hold for every \( p \in P \):

- \( O_{def}(p) = O_{loc}(p) \cup O_{out}(p) \)
- \( \forall_{\{(\ldots, \alpha_1, \ldots, \alpha_j, \ldots), (\ldots, \alpha_k, \ldots) = g \ldots \}} \subseteq E_p \alpha_i \neq \alpha_j \neq \alpha_k \)

We say that an attribute grammar \( AG = \langle G, A, E \rangle \) is a \textit{complete attribute grammar} if and only if the two previous conditions hold for every production \( p \in P \) of \( G \) and \( G \) is a complete context-free grammar. Furthermore, an attribute grammar is said to be in \textit{Bochmann normal form} if no used occurrences are in the set of output attribute occurrence, \( i.e., O_{use}(p) \cap O_{out}(p) = \{ \} \).

Semantic equations induce dependencies among attribute occurrences. In one equation of the form \( \{(\alpha_1, \ldots, \alpha_l) = f \beta_1 \ldots \beta_k \} \) every attribute occurrence \( \alpha_i \), \( 1 \leq i \leq l \), \textit{depends} on each used attribute occurrence \( \beta_j \), \( 1 \leq j \leq k \). Thus, \( E_p \) induces a dependency graph \( DP(p) \subseteq O_{pr}(p) \times O_{pr}(p) \) ("Dependencies in Production \( p \)") that summarizes the attribute dependencies associated with the production \( p \). The vertices of \( DP(p) \) are the attribute occurrences in \( O_{pr}(p) \) and the arcs are defined as follows: There is a directed arc from \( \beta \) to \( \alpha \) and we write \( \beta \rightarrow \alpha \), if \( \alpha \) depends on \( \beta \), with \( \alpha, \beta \in O_{pr}(p) \). Formally, it is defined as

\[
DP(p) = \{ \beta \rightarrow \alpha_1, \ldots, \beta \rightarrow \alpha_n \mid ((\alpha_1, \ldots, \alpha_n) = f \ldots \beta \ldots \} \in E_p \}. \\
DP = \bigcup_{p \in P} DP(p)
\]

\text{the relation of direct dependencies among attribute occurrences associated to productions.}

### 2.2.1 Attributed and Decorated Trees

An abstract context-free grammar assigns an abstract syntax tree to every sentence of the language. The attribution rules of an attribute grammar assign attributes to the nodes of the syntax tree. The resulting tree is called \textit{attributed tree} or \textit{undecorated tree}. An attributed tree \( T \) is defined as follows: to every node \( N \) of \( T \) that is an instance of non-terminal symbol \( X \) (\( i.e., X = \text{nont}(N) \)), \textit{attribute instances} are assigned. These instances correspond to the attributes of \( X \). For each attribute \( a \in A_{\text{nont}(X)} \) the correspondent instance is denoted by \( N.a \). Every local attribute of \( p \), \( l \in A_{\text{loc}}(p) \) induces an attribute instance of node \( N \), with \( p = \text{prod}(N) \). Such instances are denoted as \( N.l \).

\textit{Attribute evaluation}, \textit{attribute evaluation} also called \textit{tree decoration}, is the process that computes values of attribute instances within an attributed tree \( T \) according to the semantic equations of the underlying AG. A program that performs attribute evaluation is called
attribute evaluator. A decorated tree is an attributed tree in which all attribute instances have a value that was computed according to the attribution rules of the grammar.

The meaning of a sentence $s$ of the language generated by $G$ consists of the values of the (synthesized) attribute instances associated with the root node of the decorated attributed tree, assigned to $s$. In other words, the meaning is the function from inherited to synthesized attributes at the root of the tree. We say that two attribute grammars are equivalent if they associate the same meaning to every sentence of the (same) language they define.

**Definition 2.6 (Equivalent Attribute Grammars)** Two attribute grammars $AG_1 = \langle G_1, A_1, D_1 \rangle$ and $AG_2 = \langle G_2, A_2, D_2 \rangle$ are equivalent, and we write $AG_1 \equiv AG_2$, if and only if they define the same language $L$, i.e., $G_1 \equiv G_2$ and the attribution rules $A_1 \cup D_1$ and $A_2 \cup D_2$ are such that they associate the same meaning to every sentence $s \in L$.

We say that an AG is well-defined if the values of the attribute instances within each $T$ can be computed by an attribute evaluation process.

**Definition 2.7 (Well-defined Attribute Grammar)** An attribute grammar $AG = \langle G, A, D \rangle$ is well-defined, if the attribution rules $A \cup D$ are such that for each attributed tree $T$ generated by $G$, the values of the attribute instances within $T$ are effectively computable.

Traditionally, this criterion is established in such a way that for any attributed tree generated by an AG, no attribute depends (either directly or indirectly) on itself [Knu68].

### 2.2.2 Circularities

Before we proceed to discuss circularities in attribute grammars, let us first define dependencies within attributed trees. Attribution rules induce dependencies among attribute instances, just as they do with attribute occurrences. We say that attribute instance $N_j, \alpha$ depends on instance $N_i, \beta$ if and only if $\langle p, i, \beta \rangle \rightarrow \langle p, j, \alpha \rangle \in DP(p)$ and $p = prod(N)$. As a result, the dependencies among attribute instances in an attributed tree $T$ induce an attribute dependency graph for $T$, which we denote with $DTR(T)$. For each attributed tree $T$, $DTR(T)$ is defined by taking the attribute instances within $T$ as its vertices. This graph contains a directed arc from $N_j, \alpha$ to $N_i, \beta$ if the attribute instance $N_j, \alpha$ depends on the attribute instance $N_i, \beta$. Basically, the dependency graphs of the productions are “pasted together” on their instances: the tree nodes.

The dependencies among attribute instances specify that certain attribute values must be computed before others. Traditional definitions of well-definedness in AGs [Knu68, Alb91b] require that the attribute dependency graph induced for every attributed tree is a-cyclic. In other words, attribute instances must have a partial evaluation ordering, and
no attribute instance may transitively depend on itself. We say that an attribute grammar
is non-circular if the induced graph $DTR(T)$, for any tree $T$ generated by the grammar, is
cycle free.

**Definition 2.8 (Non-Circular Attribute Grammar)** An attribute grammar $AG$ is
non-circular, if for every attributed tree $T$ generated by $AG$, $DTR(T)$ is a-cyclic.

Non-circular attribute grammars guarantee that it is always possible to find an execution
order for semantic functions such that the value of each attribute instance is computed
exactly once, and each function applies to previously evaluated arguments only. That is to
say that, the underlying attribute evaluation model is strict. As we shall see in Chapter 3,
it is this strict model of attribute evaluation that makes this class of attribute grammars
attractive: efficient strict implementations can be automatically derived for non-circular
attribute grammars. Furthermore, there are well-known attribute grammar techniques
that statically check for circularities within attribute grammars [Knu68, Alb91b, Pen94].

In Section 3.4.1 we will present the standard circularity test that statically checks
whether an attribute grammar is non-circular or not. After that, i.e., in Section 3.6.2 we
will present Kastens’ ordered algorithm that statically computes an order for the evaluation
of the attributes [Kas80]. As a result, of this algorithm, we construct an abstract
computational model for AGs, based on visit-sequences, that represents the strict model
of attribute evaluation.

Note that forbidding circular attribute dependencies is inconvenient because some gram-
mars could naturally be expressed as a simple circular attribute grammar. In this case,
we need a broader interpretation of the concept of well-defined attribute grammar. Let
us first define *circular attribute grammars*. We say that an attribute grammar is circular
if it induces attribute dependency graphs where cycles can occur. Traditionally, circular
attribute grammars are regarded as ill-defined, i.e., not well-defined [Knu68, WG84]. Nev-
evertheless, viable semantics might be assigned to circular attribute grammars. In Section 3.4
we will discuss evaluation mechanisms that are based on the idea that the main objective
of an attribute evaluator is not to compute the values of all the attribute instances within
an attributed tree $T$, but instead, to compute just the meaning of the corresponding pro-
gram. In this case, only the value of attribute instances that do contribute to the value
of the synthesized attribute instances of the root of $T$ are really computed. All the other
attribute instances are not computed, and, consequently, they may transitively depend on
themselves.

Another means to relax the traditional interpretation of an AG as being well-defined is
based on the use of non-strict functions and incomplete values in attribute evaluation. In
other words, we can achieve a broader notion of an AG through a lazy semantics execution
model [KS87, Joh87, Aug93].
2.3 Attribute Grammar Specification

In the previous section we have formally defined attribute grammars. We shall now introduce a motivating example that will be used throughout this thesis. This is our favourite example to present the attribute grammar formalism and also to introduce the notation we adopt to define attribute grammar specifications.

Before we proceed to define our first attribute grammar, let us briefly describe how we shall organize our specification. Attribute grammars are modular: they can be (naturally) decomposed into fragments. Each of the fragments describes a particular semantic domain of the language under consideration. In Section 2.4.1 and Chapter 5 we will discuss modularity in attribute grammars. Next, we present the BLOCK language and we formally specify its processor.

2.3.1 The BLOCK Language

Consider a very simple language that deals with the scope rules of a block structured language: a definition of an identifier x is visible in the smallest enclosing block, with the exception of local blocks that also contain a definition of x. In this later case, the definition of x in the local scope hides the definition in the global one.

We shall analyse these scope rules via a toy language: the BLOCK language. One sentence in BLOCK consists of a block, and a block is a (possibly empty) list of statements. A statement is one of the following three things: a declaration of an identifier (such as decl a), the use of an identifier (such as use a), or a nested block. Statements are separated by the punctuation symbol “;” and blocks are surrounded by square brackets. A concrete sentence in this language looks as follows:

\[
\text{sentence} = [ \ \text{use } x \ ; \ \text{use } y \ ; \ \text{decl } x \\
\hspace{1em} [ \ \text{decl } y \ ; \ \text{use } y \ ; \ \text{use } w ] \\
\hspace{2em} \text{decl } y \ ; \ \text{decl } x \\
\] \]

This language does not require that declarations of identifiers occur before their first use. Note that this is the case in the first two applied occurrences of x and y: they refer to their (later) definitions on the outermost block. Note also that the local block defines a second identifier y. Consequently, the second applied occurrence of y (in the local block) refers to the inner definition and not to the outer definition. In a block, however, an identifier may be declared once, at the most. So, the second definition of identifier x in the outermost block is invalid. Furthermore, the BLOCK language requires that only defined identifiers may be used. As a result, the applied occurrence of w in the local block is invalid, since w has no binding occurrence at all.

We aim to develop a program that analyses BLOCK programs and computes a list containing the identifiers which do not obey to the rules of the language. Thus, this
program, called block, is a static semantic analyser for the BLOCK language. It has the following type:

\[
\text{block} ::= \text{Prog} \rightarrow [\text{Name}]
\]

, where Name is the type of the BLOCK identifiers.

In order to make the problem more interesting, and also to make it easier to detect which identifiers are being incorrectly used in a BLOCK program, we require that the list of invalid identifiers follows the sequential structure of the input program. Thus, the semantic meaning of processing the example sentence is \([w,x]\), i.e.:

\[
\text{block sentence} = [w,x]
\]

Next, we shall describe the program block in the traditional attribute grammar paradigm. First, we define the concrete and the abstract syntax of BLOCK via two context-free grammars. After that, we define the semantics of the language by extending the grammar with attributes and attribute equations.

### 2.3.2 The BLOCK Context-Free Grammar

Let us start by defining the set of valid sentences of BLOCK by the following concrete context-free grammar \(CG_{\text{block}} = \langle V, P, S \rangle\). As usual in context-free grammars, we present the list of productions \(P\) only. We use the standard Backus Naur Formalism (BNF), with no extensions, to denote the productions of the grammar. The vocabulary \(V\) can be easily inferred from \(P\). The axiom \(S\) is non-terminal \(\text{Prog}\). We define the following convention to denote the grammar symbols: non-terminal and pseudo-terminal symbols start with a capital letter. Literal symbols are surrounded by character “’”.

<table>
<thead>
<tr>
<th>Fragment 1: The BLOCK concrete grammar.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Root</strong> : ( \text{Prog} \rightarrow \text{'} [\text{ Stats } ] \text{'} )</td>
</tr>
<tr>
<td><strong>OneSts</strong> : ( \text{Stats} \rightarrow \text{ LstSts } )</td>
</tr>
<tr>
<td><strong>NoSts</strong> : ( \text{Stats} \rightarrow \epsilon )</td>
</tr>
<tr>
<td><strong>AStat</strong> : ( \text{LstSts} \rightarrow \text{ Stat } )</td>
</tr>
<tr>
<td><strong>ConsLstSts</strong> : ( \text{LstSts} \rightarrow \text{ Stat } \text{';'} \text{ LstSts } )</td>
</tr>
<tr>
<td><strong>CDecl</strong> : ( \text{Stat} \rightarrow \text{'} decl \text{'} \text{Name} )</td>
</tr>
<tr>
<td><strong>CUse</strong> : ( \text{Stat} \rightarrow \text{'} use \text{'} \text{Name} )</td>
</tr>
<tr>
<td><strong>CBlock</strong> : ( \text{Stat} \rightarrow \text{'} [\text{ Stats } ] \text{'} )</td>
</tr>
</tbody>
</table>

The axiom defines a block surrounded by square brackets and the non-terminal Stats defines the statements of the blocks. The body of a block is either an empty statement, which is defined by the \(\epsilon\)-production NoSts, or, it is a non-empty statement, in which case, the body of a block is defined by non-terminal LstSts. Non-terminal LstSts defines a sequence of one or more non-terminal symbols Stat separated by the literal terminal ‘ ; ’. The non-terminal Stat defines the three statements of BLOCK: definitions, uses and nested blocks.
The grammar symbol \textit{Name} is a pseudo-terminal symbol. As usual in attribute grammars, we assume that those symbols are externally provided (typically by a lexical analyzer). That is, their implicit productions are defined externally and they produce values that, in this case, are used as identifiers of the BLOCK language. Furthermore, we assume that the literal symbols also denote their types. Note that the type \textit{Name} was used when we defined the type of \textit{block}. In Section \ref{sect:lex}, we will present a lexical analyzer for the BLOCK language where this type is defined.

The BLOCK concrete grammar is used for the syntactic analysis of the BLOCK language. It guides the derivation process of each concrete sentence of the language. It also assigns a unique concrete syntax tree to each syntactically valid sentence of the language (see Figure \ref{fig:concreteGrammar}). In Section \ref{sect:lex} we will return to the syntactic analysis of the BLOCK language and we will use this grammar to guide the construction of a parser for BLOCK.

However, syntactically valid sentences may fail to correspond to semantically valid sentences of the language. Such sentences violate the semantic rules of the language. Consider, for example, the sentence \textit{sentence}: it is syntactically correct because, as we said before, can be generated by the \textit{CG}\textsubscript{block}, but it is semantically incorrect, because two identifiers (\textit{x} and \textit{w}) of \textit{sentence} violate the rules of the language. To describe the semantic rules of the language we focus on its abstract structure. So, before we extend our grammar with attribute and attribute equations, we have to define the abstract syntax of the BLOCK language.

Let us start by informally defining the abstract structure of a BLOCK sentence: \textit{an abstract sentence in BLOCK is a list of statements, where each statement is the declaration or the use of an identifier, or is a nested block}. The syntactic property according to which statements are separated by a punctuation symbol is irrelevant for the abstract structure of the language. Formally, we define such abstract language by the following productions:

\[
\begin{align*}
\text{Its} & = \text{ConsIts} \quad \text{It Its} \\
 & \mid \text{NilIts} \\
\text{It} & = \text{Decl} \quad \text{Name} \\
 & \mid \text{Use} \quad \text{Name} \\
 & \mid \text{BLOCK} \quad \text{Its}
\end{align*}
\]

\textit{Fragment 2: The BLOCK abstract grammar.}

As expected, the literal symbols, \textit{e.g.}, \texttt{'decl'}, \texttt{'use'}, \texttt{'[', ']'}, and \texttt{';'}, are not mentioned in the abstract grammar. Observe also that in the concrete grammar we have used two non-terminal symbols to specify the fact that the body of a block is a possibly empty list of statements separated by the character \texttt{";"}. As we have said above, this is irrelevant for the abstract structure of the language where non-terminal \textit{Its} simply defines a list of statements. Figure \ref{fig:concreteGrammar} presents the concrete and the abstract syntax tree assigned by the concrete and the abstract grammar, respectively, for the example sentence.

Generally, the abstract context-free grammar is simpler than the concrete one. Consequently, it is also simpler to extend the abstract grammar with attribute and attribute
2.3. Attribute Grammar Specification

Figure 2.1: The concrete (left) and the abstract (right) syntax tree assigned by the concrete and the abstract grammar for the example sentence.

equations, than to extend the concrete one. In our simple example the grammars slightly differ. In more complex and realistic examples the difference between the grammars may be greater. In both cases, however, a function mapping the concrete into the abstract grammar has to be defined. We will return to this subject in Section 2.5.1.

2.3.3 The BLOCK Attribute Grammar

We are now in a position to define the semantics of the BLOCK language. Before we proceed to extend the abstract grammar with the attributes and the equations, let us first informally analyse in more detail the semantics of the language.

The BLOCK language does not force a declare-before-use discipline. Consequently, a conventional implementation of the required analysis naturally leads to a program that traverses each block twice: once for processing the declarations of identifiers and constructing an environment and a second time to process the uses of identifiers (using the computed environment) in order to check for the use of non-declared identifiers. The uniqueness of identifiers is checked in the first traversal: for each newly encountered identifier declaration it is checked whether that identifier has already been declared at the same lexical level. In this case, the identifier has to be added to a list reporting the detected errors. The
algorithm for the processor of the BLOCK language looks as follows:

<table>
<thead>
<tr>
<th>1st Traversal</th>
<th>2nd Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Collect the list of local definitions</td>
<td>- Use the list of definitions as the global environment</td>
</tr>
<tr>
<td>- Detect duplicate definitions</td>
<td>- Detect use of non defined names (using the collected definitions)</td>
</tr>
<tr>
<td>(using the collected definitions)</td>
<td>- Combine “both” errors</td>
</tr>
</tbody>
</table>

As a consequence, semantic errors resulting from duplicate definitions are computed during the first traversal and errors resulting from missing declarations, in the second one.

Next, we shall define the attribution rules of the attribute grammar. According to the above semantic rules, for every block we compute three things: its environment, its lexical level and its invalid identifiers. The environment defines the context where the block occurs. It consists of all the identifiers that are visible in the block. The lexical level indicates the nesting level of a block. Observe that we have to distinguish between the same identifier declared at different levels, which is a valid declaration (e.g., “decl y” in *sentence*), and the same identifier declared at the same level, which is an invalid declaration (e.g., “decl x” in *sentence*). Finally, we have to compute the list of identifiers that are incorrectly used, i.e., the list of errors.

Those three things correspond to three attributes. Attributes have a type. The type of the lexical level is the primitive HASKELL type *Int*. Apart from the primitive types of HASKELL, we also assume that the type of the pseudo-terminal symbol *Name* exists (it is provided externally). In other words, the type of BLOCK identifiers is *Name*. We define two new types: the type of the environment, denoted by *Env*, that is an association list from identifiers to lexical levels, and the type *Err* that represents the type of the list of invalid identifiers. These types are defined in HASKELL as follows:

```haskell
data Tuple = Pair Name Int

type Env = [Tuple]
type Err = [Name]
```

Observe that we have used a different notation to define new types. Types, however, can be directly defined within the attribute grammar formalism: it is common in AGs to use additional non-terminal symbols to define new data types. For example, the environment can be defined by the following two non-terminal symbols and respective productions:

```
Tuple = Pair Name Int
Env = CONSEnv Tuple Env |
     NILENV
```

Type *Env* is isomorphic with non-terminal *Env*: the term constructor functions CONSENV and NILENV correspond to the HASKELL built-in list constructor functions : and [], respectively. We will use both notations to define types.
Let us now describe the semantic domains of the block language. In order to focus on each semantic domain individually we split the attribute grammar specification into fragments. Every fragment corresponds to a particular semantic domain of our language. Basically we have three semantic domains: the environment, the lexical level and the errors. Therefore we will have three AG fragments, each of which contains the attribution rules defining the respective semantic domain.

We start by defining the construction of the environment. Every block defines an environment. The environment of a local block includes the environment of its outer block together with its own local declarations. We have two alternatives to construct such environment: either we synthesize the local declarations using a bottom-up strategy and we “add” these declarations to the environment of the outer block in order to obtain the required environment, or we thread the environment of the outer block through the body of the local block and we accumulate its local declarations. Although both alternatives seem to follow the algorithm, only the second one produces the desired results. Observe that the duplicated declarations are detected during the construction of the environment. The first alternative detects those duplications, but points out the wrong occurrence of a duplicated declaration: since it performs a bottom-up strategy (in other words, it traverses the block sentences from right-to-left) it is the first occurrence of the declaration of an identifier that is marked as duplicated, instead of the second one. Thus, we adopt the second alternative, since it detects the duplications following the sequential structure of the input\(^2\).

Every block inherits the environment of its outer block. Thus, we associate an inherited attribute \(\text{dcli}\) of type \(\text{Env}\) to the non-terminal symbols \(\text{Its}\) and \(\text{it}\) that define a block. The inherited environment is threaded through the block in order to accumulate the local definitions and in this way synthesizes the total environment of the block. We associate a synthesized attribute \(\text{dclo}\) also of type \(\text{Env}\) to the non-terminal symbols \(\text{Its}\) and \(\text{it}\), which defines the newly computed environment. Inherited (synthesized) attributes are prefixed with the down (up) arrow \(\downarrow\) (\(\uparrow\)).

\[
\text{Its} \quad <\downarrow \text{dcli} : \text{Env}, \uparrow \text{dclo} : \text{Env}> \\
\text{It} \quad <\downarrow \text{dcli} : \text{Env}, \uparrow \text{dclo} : \text{Env}>
\]

Next, we associate semantic equations with every production. We use the following notation for the semantic equations: within the equations of a production, different occurrences of the same symbol are denoted by distinct subscripts. For example, the occurrences of the grammar symbol \(\text{Its}\) in production \(\text{ConsIts}\) are denoted in the equations as follows: \(\langle \text{ConsIts}, 0 \rangle\) is denoted by \(\text{Its}_1\) (its first occurrence) and \(\langle \text{ConsIts}, 2 \rangle\) is denoted by \(\text{Its}_2\) (its second occurrence). No subscript is used for symbols that occur only once in a production. Thus, we simply denote \(\langle \text{ConsIts}, 1 \rangle\) by \(\text{It}\). Attribute occurrence \(\langle \text{ConsIts}, 0, \text{dclo} \rangle\) is denoted by \(\text{Its}_1, \text{dclo}\). The semantic equations are written as HASKELL-like expressions. The semantic equations for the environment look as follows:

\(^2\)In functional programming this technique is used for improving functional programs and is known as accumulation parameters. The techniques is studied in [Bir84a].
Chapter 2. Definitions and Notations

Fragment 3: Computing the environment.

The only production that contributes to the synthesized environment of a block is `DECL`. The single semantic equation of this production makes use of the semantic function `:` (written in infix notation) to build the environment. Note that we are using the HASKELL type definition presented previously. The constructor `PAIR` is used to bind an identifier to its lexical level. The single occurrence of pseudo-terminal `Name` is syntactically referenced in the equation since it is used as a normal value of the semantic function. All the other semantic equations of this fragment simply pass the environment to the left-hand side and right-hand side symbols within the respective productions. The semantic function is the identity function, which we omit from the copy rule.

Now that the total environment of a block is defined, we pass that context down to the body of the block in order to detect applied occurrences of undefined identifiers. Thus, we define a second inherited attribute of type `Env`, called `env`, to distribute the total environment. It should be noticed that attribute `dclo` can be used to correctly compute the required list of errors. We choose to distribute the list of declarations in a new attribute to demonstrate our techniques. As we will discuss in Chapter 3, with this approach we force a two traversal (strict) evaluation scheme. Although this approach is not really needed in the trivial `BLOCK` language, it is a common feature when defining real languages [SAS98] (this is, for example, the case in the BibTEX language).

Fragment 4: Distributing the environment.

The first semantic equation of `BLOCK` specifies that the inner blocks inherit the environment of their outer ones. As a result, only after computing the environment of a block is it possible to process its nested blocks. That is, inner blocks will be processed in the second traversal of the outer one.

The total environment of the inner blocks, however, is the synthesized environment (i.e., attribute `dcl`, as defined in the second equation. It is also worthwhile to note that the equation `Its.env = Its.dclo` induces a dependency from a synthesized to an inherited attribute of the same symbol `Its`. Although such dependencies are natural in attribute grammar specifications they may lead to complex (functional) implementations. We will
discuss these dependencies in detail in Section 3.4.

Every block has a lexical level. Thus, we introduce one inherited attribute \( \text{lev} \) indicating the nesting level of a block. The Haskell primitive function \(+\) is used to increment the value of the lexical level passed to the inner blocks.

\[
\text{its.lev} = \text{it.lev} + 1
\]

Fragment 5: Computing the lexical level.

Finally, we have to synthesize one attribute defining the (static) semantic errors. We define a second synthesized attribute: \( \text{errs} \) of type \( \text{Err} \). The attribution rules for this semantic domain are shown in Fragment 6.

\[
\text{its.errs} = \begin{cases} \emptyset & \text{if } \text{it errs} \text{ is an empty list} \\ \text{it.errs} \uplus \text{its.errs} & \text{otherwise} \end{cases}
\]

Fragment 6: Synthesizing the list of errors.

There are two semantic functions defined on environments: \( \text{mBIn} \) and \( \text{mNBIn} \). The definition of these functions must be included in the grammar specification. For this reason, attribute grammar specification languages provide an additional notation in which semantic functions can be defined. Generally, this notation is simply a standard programming language. As we have explained at the beginning of this chapter, we use the standard functional programming language Haskell. Thus, the two semantic functions look as follows:

\[
\begin{align*}
\text{mBIn} & : \text{Name} \to \text{Env} \to \text{Err} \\
\text{mBIn} \text{id} & = \text{id} \\
\text{mBIn} \text{id} (t:es) & = \begin{cases} \\
\text{case } t \text{ of} & \\
\text{(Pair } n \text{ l)} & \to \text{if } n = \text{id} \text{ then } \emptyset \\
\text{else } \text{id} & \text{mBIn} \text{ es} \\
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
\text{mNBIn} & : \text{Tuple} \to \text{Env} \to \text{Err} \\
\text{mNBIn} t & = \emptyset \\
\text{mNBIn} t (e:es) & = \begin{cases} \\
\text{if } t = e \text{ then } \text{idTup} t \text{ s} & \text{if } t = e \\
\text{else } t & \text{mNBIn} \text{ es} \\
\end{cases} \\
\end{align*}
\]

Fragment 7: The semantic functions written in Haskell.

We assume that values produced by pseudo-terminal \text{Name} are comparable for equal-

---

3In Haskell, functions can be written between their (two) arguments, rather than preceding them, by enclosing the name in back quotes. A function written in infix notation is called an operator.
The equality function between BLOCK identifiers must be, once again, provided by the external lexical analyser.

The fragments of the attribute grammar presented thus far specify the BLOCK language. Our specification of the BLOCK language, however, does not define the initial environment and the lexical level of the outermost block. Observe that, they are two inherited attributes of the “root” of the abstract grammar, and, consequently, their values have to be provided externally. In order to specify the context of the outermost block within the AG formalism, we add a new non-terminal symbol $P$ to the abstract grammar (Fragment 2). This symbol is the axiom of the abstract grammar.

$$P = R \text{ Its}$$

Now, we can easily write the attribution rules specifying that the initial environment of the outermost block is an empty environment (i.e., it is context-free) and that its lexical level is the value 0.

<table>
<thead>
<tr>
<th></th>
<th>&lt;\uparrow \text{errs} : \text{Err}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P = R \text{ Its}$</td>
<td></td>
</tr>
<tr>
<td>$\text{Its.dcli} = \emptyset$</td>
<td></td>
</tr>
<tr>
<td>$\text{Its.lev} = 0$</td>
<td></td>
</tr>
<tr>
<td>$\text{Its.env} = \text{Its.dclo}$</td>
<td></td>
</tr>
<tr>
<td>$P.\text{errs} = \text{Its.errs}$</td>
<td></td>
</tr>
</tbody>
</table>

*Fragment 8*: The “root” of the abstract grammar.

Patterns of Attribute Propagation

The BLOCK AG, although very simple, presents typical patterns of attribute definitions that occur very frequently in attribute grammar specifications. For example, in Fragment 3 the equations of the productions applied on $\text{Its}$ define a typical left-to-right pattern of attribute propagation within a production. Fragments 4 and 5 present another typical pattern: the use of copy rules to propagate contextual information from the context of non-terminal $X$ to the context of non-terminal $Y$, where $Y$ is derivable from $X$. In terms of the tree it propagates information downwards. In Fragment 6, attribute values are collected and propagated up in the tree.

The propagation of information via copy rules results in large and less legible AG specifications. Typical patterns of attribution, however, can be specified in a concise and comprehensible notation, that makes AG specifications shorter and easier to understand. These compact abstractions can be systematically transformed into the primitive AG notation, thus preserving the semantics of the grammar [KW94, Paa95]. Another technique to eliminate redundant copy rules is to include them as implicit default rules.

\[\text{4In HASKELL we would say that type Name is an instance of class Eq.}\]
There are standard AG notations for most of those schemes of attribution [Kas91a, KW94, RT89], which eliminate the need to propagate attributes in the tree. We will use a special notation to access upward remote attribute occurrences as defined in [RT89]. An upward remote attribute occurrence, written as $\uparrow X.a$, may be used with the semantic equations to denote the value of an attribute instance occurring higher up in the syntax tree. Within the semantic equations of a production $p$, an upward remote attribute occurrence refers to an attribute of a different production, which necessarily occurs between any instance of $p$ and the root of the tree. The value of the upward attribute of $p$ is the value of its first occurrence on the path from the instance of $p$ and the root of the tree. Using such a notation, Fragment 4 can be re-defined as follows:

$$\begin{align*}
\text{It} & \quad \langle \downarrow \text{env} : \text{Env} \rangle \\
\text{It} & \quad = \quad \text{Block \ Its} \\
\text{Its.del} & \quad = \quad \uparrow \text{Its.env} \\
\text{Its.env} & \quad = \quad \text{Its.dclo}
\end{align*}$$

### 2.3.4 Attribute Dependencies

The fragments of attribute grammars previously presented define formally the block language and its desired processor: block. We will describe how to derive such processor from our AG specification in Chapters 3 and 4. To derive correct and efficient programs, however, we first need to guarantee that the attribute grammar under consideration is well-defined (Definition 2.7). One way to establish whether an AG is well-defined or not is by analysing the dependency graphs induced by the attribution rules. So, we proceed now by defining the dependency graphs for our AG.

As usual in attribute grammars, the different fragments of the specification are fused according to the respective productions, in order to construct a complete specification of the language. This complete AG specification is then (globally) analysed to detect possible circularities. As a result of the fusion process each production of the complete AG contains the semantic equations associated to it on each of the fragments. The semantic equations induce dependencies among attribute occurrences. $DP(p)$ is the dependency graph that summarizes the dependencies in production $p$. To provide a better insight in the attribute dependencies of productions, we shall use a graphical notation to represent their dependency graphs.

A production $p$ is graphically denoted as follows: the left-hand side of $p$ is drawn at the top of the picture, and the right-hand side symbols at the bottom, following the order described by $p$. The inherited (synthesized) attribute occurrences of the grammar symbols are placed on their left (right) side. The semantic functions are rendered by labelled rectangles. Arrows are used to represent the attribute occurrence dependencies. Figure 2.2 shows the dependencies induced for each production of our attribute grammar.

Observe that there are two dependencies from a synthesized attribute to an inherited one in productions $R$ and $Block$, respectively. In a strict model of interpretation of the attribute grammar, these dependencies force additional traversals to the abstract syntax...
Figure 2.2: The graphical representation of the dependency graphs $DP$ induced by the block AG.

tree. In a lazy model of interpretation, they induce pseudo-circular definitions. These two models of interpretation of an AG will be discussed in Chapter 3.

The dependencies among attribute occurrences induce a dependency graph among the attribute instances of a syntax tree. Figure 2.3 shows the attributed tree and the induced dependency graph for sentence: $[\text{decl } x ; \text{use } y ; [\text{decl } y ; \text{use } y ] ; \text{decl } x]$.

### 2.4 Attribute Grammar based Systems

The key feature of the attribute grammars is that they are executable, i.e., efficient implementations can be automatically derived from an attribute grammar specification. Traditionally, attribute grammar based systems fuse all the fragments of the specification into an equivalent monolithic AG, before they are able to derive the respective implementation. The construction of such monolithic AG is needed, because standard algorithms that check for circularities on AGs are based on a (static) global analysis of all the attribute dependencies. It is from the monolithic attribute grammar that the implementation is derived. These implementations are usually divided into three main components: the lexical analyser, the parser and the attribute evaluator. These components are briefly described next.

- **Lexical Analyser:** The first component of an AG system is the lexical analyser or scanner, which accepts a sequence of characters and yields a sequence of basic symbols, also called tokens, of the language. The lexical analyser groups characters of the input into symbols of the language, usually ignoring comments and superfluous spaces. Furthermore, it has to distinguish between literal and pseudo-terminal
symbols: the former are completely dealt by the lexical analyser. The latter carry information that must be passed to the attribute evaluator.

\[
\text{scanner} :: \text{[Char]} \rightarrow \text{[Token]}
\]

where \text{Token} is the type of the tokens, \textit{i.e.}, the set of all possible values of the basic symbols of the language.

- **Parser**: The parser accepts the sequence of tokens delivered by the lexical analyser and constructs the concrete syntax tree.

\[
\text{parser} :: \text{[Token]} \rightarrow \text{Tree}
\]

- **Attribute Evaluator**: The attribute evaluator decorates the syntax tree delivered by the parser, assigning a meaning to every tree.

\[
\text{evaluator} :: \text{Tree} \rightarrow (\text{syn}_1, \text{syn}_2, \ldots, \text{syn}_n)
\]
• **The AG System:** An attribute grammar system is the composition of the previous three functions:

\[ \text{ag} = \text{evaluator} \cdot \text{parser} \cdot \text{scanner} \]

Figure 2.4 presents the structure of an attribute grammar system.

---

2.4.1 **Shortcomings of Attribute Grammars**

Attribute grammars are known to provide a limited form of modularity. Although different semantic domains of the language under consideration can be separated into different fragments, they cannot be viewed as separate units of compilation, and, consequently, cannot be analysed independently. Separate analysis and compilation of program components is nowadays a “standard” form of modularity provided by modern programming languages. Within the classical AG formalism, the AG fragments have to be (syntactically) fused, according to the productions, in order to associate a meaning to the AG. Consequently, the AG writer is actually forced to consider all the fragments simultaneously. We call this classical form of modularity *syntactic compositionality*. In Chapter 5 we extend the attribute grammar formalism to provide a modern form of modularity: the AG grammar is separated into *components*, which are autonomous fragments, *i.e.*, they can be analysed and compiled independently.

As a second drawback of the classical attribute grammar is the fact that very often we would like to specify a computation that is not easily expressible by some form of induction over that particular tree. If we were able to compute a better suited structure, we might easily express such computations. In the classical attribute grammar formalism this is not possible because the structure of the syntax trees is fixed during decoration. This problem is very common in attribute grammars, because for most languages their concrete grammars (and correspondent trees) differ from the abstract ones. So, within the classical attribute grammar formalism, it is not possible to express, through attribution, the mapping from a concrete grammar into an abstract one. As a result, we may be forced to express the
semantic equations directly over the (unsuitable) concrete context-free grammar. The CFG used for parsing, dictates the form of the syntax tree. This “abstract” syntax tree is not the optimal starting point to perform the semantic computations, since it is designed in order to describe all the syntactic features of the language, and not its abstract structure.

Usually, attribute grammar based systems provide additional notation to map the concrete tree into the abstract one \( \text{GE90 RT89 GHL+92} \). In the *Synthesizer Generator* \( \text{RT89} \), for example, the abstract syntax tree, which is used as the starting point for attribute evaluation, is computed as a synthesized attribute of the concrete tree. In the *Eli* system \( \text{GHL+92} \) the mapping is handled by a special tool: the *Maptool* \( \text{KW95} \). These two systems, however, provide one step only, since it is not possible to compute a better suited structure during the decoration of the abstract syntax tree.

### 2.5 Higher-Order Attribute Grammars

*Higher-Order Attribute Grammars* \( \text{VSK89 SV91} \) are an important extension to the attribute grammar formalism. Conventional attribute grammars, thereafter also referred to as *first-order attribute grammar*, are thus augmented with *higher-order attributes*. Higher-order attributes are attributes whose value is a tree. We may associate, once again, attributes with such a tree. Attributes of these so-called *higher-order trees*, may be higher-order attributes again. Higher-order attribute grammars have three main characteristics:

- To begin with, when a computation cannot be easily expressed in terms of the inductive structure of the underlying tree, a better suited structure can be computed before. Consider, for example, a language where the abstract grammar does not match the concrete one. Consider also that the semantic rules of such a language are easily expressed over the abstract grammar rather than over the concrete one. The mapping between both grammars can be specified within the higher-order attribute grammar formalism: the attribute equations of the concrete grammar define a higher-order attribute representing the abstract grammar. As a result, the decoration of a concrete syntax tree constructs a higher-order tree: the abstract syntax tree. The attribute equations of the abstract grammar define the semantics of the language. Consequently, the decoration of the abstract tree is performed according to the attribution rules defined for the abstract grammar. That is, the semantic rules of the language are checked during the decoration of the abstract syntax tree.

- Secondly, semantic functions are redundant. In higher-order attribute grammars every computation can be modelled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. For example, a typical application of higher-order attributes is to model the (recursive) lookup function in an environment. Consequently, there is no need to have a different notation (or language) to define semantic functions in AGs.
• The third characteristic is that part of the abstract tree can be used directly as a value within a semantic equation. That is, grammar symbols can be moved from the syntactic domain to the semantic domain.

These characteristics make higher-order attribute grammars particularly suitable to model interactive language-based environments \[TC90\] \[Pen94\] \[KS98\]. In such interactive environments, an incremental execution model is essential, since they provide immediate answers after each user interaction. Generally, incremental evaluation is achieved by memoing, and reusing, values of attribute instances. Consequently, a better incremental behaviour is obtained if every inductive computation is expressed in terms of attribution rules, \textit{i.e.}, in terms of higher-order attributes. On the contrary, if we express such computation in terms of a semantic function, then there are no attribute values and, obviously, no reuse of their values. In Chapter 6, we discuss incremental attribute evaluation and, in Chapter 7, we present the attribute grammar system developed at our department, which models interactive language-based environments via HAGs.

Before we formally define higher-order attribute grammars, let us first discuss how we model higher-order attributes. We use \textit{attributable attributes}, which we will abbreviate to \(a_t\), as attributes associated with productions, very much like local attributes. Like any other attribute, attributable attributes have a type. The type of an \(a_t\) is a (inductive) type induced by a non-terminal symbol of the grammar. Inherited (synthesized) attributes of such non-terminal symbol induce inherited (synthesized) attributes to the respective attributable attribute. Thus, the equations of the productions should include semantic equations to define those attributes, \textit{i.e.}, the attributable attributes and their attributes.

The following definition of higher-order attribute grammar is taken from \[Pen94\].

\textbf{Definition 2.9 (Higher-Order Attribute Grammar)} A higher-order attribute grammar (HAG) is a triple \(HAG = (G, A, D)\). \(G = (V, P, S)\) is a context-free grammar. \(A\) is a finite set of attributes, partitioned into sets \(A_{\text{nont}}(X)\), \(A_{\text{loc}}(p)\) and \(A_{\text{ata}}(p)\) for each \(X \in N\) and \(p \in P\). \(A_{\text{nont}}(X)\) are further partitioned into two disjoint subsets \(A_{\text{inh}}(X)\) and \(A_{\text{syn}}(X)\). \(D = (T, E)\) is the semantic domain of HAG. \(T\) is a finite set of types and \(E\) is a finite set of semantic equations.

\(A\) and \(D\) are now known as the \textit{higher-order attribution rules} of HAG. The set \(A_{\text{nont}}(X)\) consists of all attributes associated with non-terminal symbol \(X\), exactly as defined in Definition 2.5. An element \(a \in A_{\text{nont}}(X)\) is denoted by \(X.a\), and it is either \textit{inherited} if \(a \in A_{\text{inh}}(X)\) or \textit{synthesized} if \(a \in A_{\text{syn}}(X)\). The function \(\mathbb{T} :: A \rightarrow T\) associates a type with every attribute. The sets of attribute occurrences \(O_{\text{inh}}(\langle p, i \rangle)\), \(O_{\text{syn}}(\langle p, i \rangle)\), and \(O_{\text{nt}}(\langle p, i \rangle)\) induced by the occurrence of a symbol \(X_i\) in a production \(p\) are defined in Definition 2.5 and we will not repeat them.

The attributes \(a \in A_{\text{loc}}(p)\) are the local attributes of \(p\) and are denoted by \(p.a\). Each local attribute \(p.a\) induces a local attribute occurrence of \(p\). The set of all local attribute occurrences of \(p\) is \(O_{\text{loc}}(p) = A_{\text{loc}}(p)\).
2.5. Higher-Order Attribute Grammars

The set $A_{ata}(p)$ consists of the attributable attributes associated to production $p$. They are denoted by $p.x$ (as local attributes of $p$ are). Each attributable attribute $p.x$ induces an attributable attribute occurrence of $p$. Therefore, we have $O_{ata}(p) = A_{ata}(p)$. Attributable attributes have a (inductive) type. The function $T$ works on attributable attributes as well. For each $x \in A_{ata}(p), (T_x) \in T$ is the set of all possible values of $x$. A possible
value for an attributable attribute is a (higher-order) tree. These set of values are defined by non-terminal symbols. As a result, for each $p \in P$ and for each $x \in A_{ata}(p), (T_x) \in N$. Furthermore, every attribute $a \in A_{koni}(X)$ induces a generated attribute occurrence denoted by $p.x.a$. The sets of attribute occurrences induced by an attributable attribute are defined as follows:

\[
O_{inhata}(p.x) = \{p.x.a \mid T(p.x) = X \land a \in A_{inh}(X)\}
\]

\[
O_{synata}(p.x) = \{p.x.a \mid T(p.x) = X \land a \in A_{syn}(X)\}
\]

\[
O_{oneata}(p.x) = O_{inhata}(p.x) \cup O_{synata}(p.x)
\]

and the generated attribute occurrences associated with production $p$ are defined as

\[
O_{gen}(p) = \bigcup_{p.x \in O_{ata}(p)} O_{oneata}(p.x)
\]

In higher-order attribute grammars the set of attribute occurrences of production $p$ is $O_p(p) = O_{loc}(p) \cup O_{ntocp}(p) \cup O_{ata}(p) \cup O_{gen}(p)$. The sets of input and output occurrences of $p$ are now defined as follows:

\[
O_{inp}(p) = O_{inh}(lhs(p)) \cup \bigcup_{1 \leq i \leq |p|} O_{syn}(\langle p, i \rangle) \cup \bigcup_{p.x \in O_{ata}(p)} O_{synata}(p.x)
\]

\[
O_{out}(p) = O_{syn}(lhs(p)) \cup \bigcup_{1 \leq i \leq |p|} O_{inh}(\langle p, i \rangle) \cup \bigcup_{p.x \in O_{ata}(p)} O_{inhata}(p.x)
\]

The attribute equations of a production $p E_p$, its induced dependency graph $DP_p$, its sets of defined and used occurrences $O_{def}(p)$ and $O_{use}(p)$, respectively, are similar to the definitions presented in Definition 2.5. We do not repeat them. The notion of Bochmann normal form and equivalent grammars are also similar.

We say that a higher-order attribute grammar $HAG = \langle G, A, E \rangle$ is complete if and only if $G$ is complete and the following three conditions hold for every production $p \in P$:

- $O_{def}(p) = O_{loc}(p) \cup O_{out}(p) \cup O_{ata}(p)$
- $\forall \{(\ldots, \alpha_i, \ldots, \alpha_j, \ldots) = f \cdots, (\ldots, \alpha_k, \ldots) = g \cdots\} \in E_p \alpha_i \neq \alpha_j \neq \alpha_k$
- for every $((\ldots, p.x, \ldots) = f \cdots) \in E_p$, the type of $f$ is $f:: \cdots \rightarrow (\cdots, T(p.x, \cdots)$

The completeness property does not guarantee that it is always possible to compute a meaning to every abstract syntax tree of the grammar: indeed circular dependencies may occur. If they do not occur for any tree with appropriate instances of its attributable attributes, the grammar is called well-defined.

Definition 2.10 (Well-Defined Higher-Order Attribute Grammar) A higher-order attribute grammar $HAG = \langle G, A, D \rangle$ is well-defined, if the attribution rules $A \cup D$ are such that for each attributed tree $T$ generated by $G$, the values of the attribute instances within $T$ are effectively computable.
2.5.1 Higher-Order Attribute Grammar Specification

In order to show the properties of the higher-order attribute grammars we shall proceed to define a higher-order variant for the BLOCK attribute grammar. But, before doing that, let us recall that the the concrete grammar of BLOCK (presented in Fragment 1) differs slightly from the abstract one (presented in Fragment 2). As we explained before, a mapping between both grammars has to be defined. So, we extend our attribute grammar with a new semantic domain to map the concrete into the abstract grammar. Obviously, we wish to define such a mapping within the attribute grammar formalism. This is exactly one of the features of HAGs.

We shall build up the higher-order variant of the BLOCK AG by adding new fragments, and by reusing fragments of the first-order grammar.

We begin by introducing a higher-order attribute, which defines the abstract syntax tree. The type of this higher-order attribute is the type of the abstract syntax tree, i.e., Its. We associate our first attributable attribute, called ast, to the production applied on the start symbol of the concrete grammar. The declaration of this ata and its instantiation look as follows:

\[
\begin{align*}
\text{Root} & : \text{Prog} \rightarrow '[: Stats ']'
\text{ata} & : \text{Its} \\
ast & = \text{Stats.}\text{ast}
\end{align*}
\]

The above semantic equation defines the initial value of the ata ast as the synthesized attribute ast of Stats (once again we have omitted the identity semantic function).

Next, we define the attribution rules that compute the abstract syntax. Since in this simple example the grammars are very similar, a single synthesized attribute suffices. Thus, we associate the synthesized attribute ast to the non-terminals of the concrete grammar. In this case, the attribute equations just use the constructor functions of the abstract grammar as semantic functions to construct the abstract tree (e.g., NILIts, CONSIts, DECL, USE and BLOCK).

\[
\begin{array}{llll}
\text{Stats} & \langle \uparrow \text{ast} : \text{Its}\rangle \\
\text{OneSts} & : \text{Stats} \rightarrow \text{LstSts} \\
\text{Stats.}\text{ast} & = \text{LstSts.}\text{ast} \\
\text{NoSts} & : \varepsilon \\
\text{Stats.}\text{ast} & = \text{NilIts} \\
\text{AStat} & : \text{LstSts} \rightarrow \text{Stat} \\
\text{LstSts.}\text{ast} & = \text{CONSIts Stat.}\text{ast} \text{NilIts} \\
\text{ConsLstSts} & : \text{Stat ';}\text{LstSts} \\
\text{LstSts}_1.\text{ast} & = \text{CONSIts Stat.}\text{ast} \text{LstSts}_2.\text{ast}
\end{array}
\]

\[\text{Fragment 9: Mapping concrete to abstract syntax.}\]

The non-terminal symbol Its has three inherited attributes: \(A_{\text{inh}}(\text{Its}) = \{\text{lev, env, dcli}\},\)
and two synthesized ones \( A_{syn}(Its) = \{dcl, errs\} \). Therefore, \( ast \) generates five attribute occurrences: \( O_{oneata}(ast) = \{ast.dcli, ast.lev, ast.env, Its.dcl, ast.errs\} \). In order to define a complete HAG, we need to include in the grammar the semantic equations that instantiate the generated inherited attribute occurrences.

\[
\begin{array}{l}
| Prog <\uparrow \text{errs} : \text{Err} > \mid \\
| \text{Root} : Prog \rightarrow [\text{\'Stats\'1}] \mid \\
| \text{ata} \ ast : Its \mid \\
| \text{ast} \ = \ Stats.ast \mid \\
| \text{ast.dcli} \ = \ [] \mid \\
| \text{ast.lev} \ = \ 0 \mid \\
| \text{ast.env} \ = \ ast.dclo \mid \\
| \text{Prog.errs} \ = \ ast.errs \\
\end{array}
\]

\textit{Fragment 10: Abstract syntax as a higher-order attribute.}

Observe that this corresponds to Fragment 8 of the first-order attribute grammar. Such a fragment is replaced by this one in the higher-order variant of the AG.

Let us consider now the semantic function presented in Fragment 7. Such recursive functions can be defined through attribution rules as well. Thus, attribution rules are associated to the production applied on \( Env \), and they replace the semantic functions \( mBIn \) and \( mNBIn \).

\[
\begin{array}{l}
| Env <\downarrow id : \text{Name}, \downarrow \text{lev} : \text{Int}, \uparrow \text{mBIn} : \text{Err}, \uparrow \text{mNBIn} : \text{Err} > \mid \\
| Env \rightarrow \text{NilEnv} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \text{ConsEnv} \ Tuple \ Env \mid \\
| Env \rightarrow \text{NilEnv} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \text{ConsEnv} \ Tuple \ Env \mid \\
\end{array}
\]

\[
\begin{array}{l}
| Env \rightarrow \text{NilEnv} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \text{ConsEnv} \ Tuple \ Env \mid \\
\end{array}
\]

\[
\begin{array}{l}
| Env.mBIn \ = \ [Env.id] \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \text{ConsEnv} \ Tuple \ Env \mid \\
\end{array}
\]

\[
\begin{array}{l}
| Env1.mBIn \ = \ \text{if Tuple.id == Env1.id} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \text{ConsEnv} \ Tuple \ Env \mid \\
\end{array}
\]

\[
\begin{array}{l}
| Env2.id \ = \ Env1.id \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \text{Env2.lev} \ = \ Env1.lev \\
\end{array}
\]

\[
\begin{array}{l}
| Tuple \rightarrow \text{Pair} \ Name \ Int \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \text{Tuple.id} \ = \ Name \mid \\
\end{array}
\]

\[
\begin{array}{l}
| \phantom{\text{NilEnv}} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \phantom{\text{NilEnv}} \mid \\
| \text{Tuple.lev} \ = \ Int \\
\end{array}
\]

\textit{Fragment 11: Semantic functions as higher-order attributes.}

This allows us to use, in the productions \textbf{DECL} and \textbf{USE}, an attributable attribute \textit{table} of type \( Env \) to lookup the environment.
Chapter 2. Definitions and Notations

Fragment 12: Synthesizing the list of errors using higher-order attributes.

This is the typical example where a higher-order attribute replaces a recursive function. Let us compare this fragment with the equivalent first-order one (Fragment 7). Observe that the semantic equations of Fragment 7 must provide the arguments (i.e., the attribute occurrences) of the semantic functions, according to their specific types. In the higher-order variant we inherited the advantages of the AG formalism: we are no longer restricted to such implicit positional argument, which is enforced by the conventional functional style.

The next table summarizes the different variants of the block attribute grammar that can be obtained by fusing different AG fragments presented in this chapter.

<table>
<thead>
<tr>
<th>BLOCK AG</th>
<th>Class</th>
<th>Fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG1: Abstract grammar</td>
<td>First-order</td>
<td>2, 3, 4, 5, 6, 7, 8</td>
</tr>
<tr>
<td>AG2: Concrete grammar</td>
<td>Higher-order</td>
<td>1, 2, 9, 4, 6, 7, 10, 9</td>
</tr>
<tr>
<td>AG3: Concrete grammar</td>
<td>Higher-order</td>
<td>1, 2, 3, 4, 5, 6, 7, 10, 9, 11, 12</td>
</tr>
</tbody>
</table>

2.5.2 Reducing Higher-Order Attribute Grammars

A Higher-order attribute grammar can be transformed into an equivalent first-order attribute grammar. This transformation is known as Higher-Order Attribute Grammar Reduction [Vog93]. After reducing a higher-order attribute grammar, it is possible to use standard attribute grammar algorithms to handle higher-order ones as well.

The key idea of the reduction process is to replace every higher-order attribute by a non-terminal symbol. That is, every attributable attribute \( a \) of type \( X \) is replaced by non-terminal \( X \). The attribute occurrences generated by \( a \) are then replaced by the attribute occurrence of \( X \). A higher-order tree, however, can only be decorated after it has been constructed. So, the generated synthesized attributes of an attributable attribute should contribute neither directly nor indirectly to the computation of the higher-order tree. Therefore, we also introduce a local attribute that captures the dependencies of the attributable attribute itself. The following definition is taken from [Pen94].

Definition 2.11 (Reducing Higher-Order Attribute Grammars) Let \( H \) be a higher-order attribute grammar. The reduced attribute grammar \( H' \) is obtained by the following transformations on every production \( p \) of \( H \):
• For every \( p.x \in O_{\text{oneata}}(p) \) a non-terminal \( T.p.x \) is appended to \( p \), say at position \( i_x \), and a local attribute \( p.l_x \) is associated to \( p \).

• Every attributable attribute \( p.x \in O_{\text{oneata}}(p) \) is removed.

• Every attribute occurrence \( \langle p, x, a \rangle \) in any equation \( E(p) \) is replaced by \( \langle p, i_x, a \rangle \).

• Every synthesized attribute occurrence \( \langle p, i_x, a \rangle \) is explicitly made dependent on \( p.l_x \), for example by adding the arc \( p.l_x \rightarrow \langle p, i_x, a \rangle \) to \( DP(p) \).

2.6 Parsing with Combinators

Traditionally, attribute grammar systems use parser generator tools to implement a parser for the language under consideration. Such tools accept as input the context-free grammar of the AG and produce a program that implements the parsing process of the language. In this section we show how, in a purely functional setting, we can write functions that look like the context-free grammar itself. That is, we use a small set of parsing functions that can be used to construct recursive descendent parsers [ASU86]. The notation used for those functions is very similar to the Backus Naur formalism, and consequently, writing the parsing functions is as simple as writing the corresponding context-free grammar. The parsing functions are known as parser combinators [Fok95, HM98].

Before we define the parser for the BLOCK language, we briefly discuss the basic combinators used in its construction. The combinators used in this example have the following types:

\[
\begin{align*}
symbol & : : \text{Eq } s \Rightarrow s \rightarrow \text{Parser } s s \\
succeed & : : r \rightarrow \text{Parser } s r \\
(<|>) & : : \text{Parser } s r \rightarrow \text{Parser } s r \rightarrow \text{Parser } s r \\
(<*>) & : : \text{Parser } s (a \rightarrow r) \rightarrow \text{Parser } s a \rightarrow \text{Parser } s r \\
(<$>) & : : (a \rightarrow r) \rightarrow \text{Parser } s a \rightarrow \text{Parser } s r
\end{align*}
\]

The \( \text{symbol} \) combinator consumes a symbol \( s \) of the input whenever the head of the input is equal to \( s \). The symbols to be parsed must be tested for equality as indicated by the predicate \( \text{Eq} \) in its type. The \( \text{succeed} \) combinator parser function that does not consume any symbol and always returns a given, fixed value, i.e., it always succeeds. The \( \text{alternative} \) combinator \( <|> \) takes two parsers and “tries” to parse the input using both alternatives. The \( \text{sequence} \) combinator \( <*> \) also takes two parsers, say \( p \) and \( q \), and parses anything that \( p \) and \( q \) would, if placed in succession. The result of this combinator is obtained by applying the result of the first parser to the result of the second one. Finally, the last combinator applies a function, the so-called semantic function, to the result of a parser.

We present parser combinators not only because they elegantly model parsers in a purely functional language, but because they also illustrate the concepts of higher-order
functions, polymorphic functions and lazy evaluation. Familiarity with these concepts will aid in understanding the attribute evaluators discussed in Chapter 4.

2.6.1 Parsing the BLOCK Language

Let us now write the parser for the BLOCK language. Parsing is usually preceded by the lexical phase that transforms the argument string into a sequence of basic symbols of the language. The lexical phase can be avoided by defining suitable combinators. For example, we can easily define parser combinators to handle comments, keywords and the use of space between tokens. This is the approach taken in [HM98]. In order to have a clear separation between the parsing and the lexical phase, and to follow the traditional modular approach of attribute grammar systems, we choose to implement the scanner and the parser as separate functions. These functions are usually produced by lexical and parser tools. Next, we show how they can be easily written in a functional programming setting.

The Scanner

Before we present the function scanner let us define the data type Token. This type defines the set of values to be passed to the parser. Instead of passing the literal value of each basic symbol to the parser, a more efficient implementation is obtained if the scanner classifies the basic symbols into classes [WC93]. Such classification is then used by the parser. As a result, we define a constructor of type Token for each class of basic symbols. For literal symbols, classification is the only information of interest. For pseudo-terminals, however, we need to define the type of the information they pass to the next phases. The type Token is then defined as follows:

```haskell
data Token = TkDCL -- 'decl'
            | TkUSE -- 'use'
            | TkSC -- ','
            | TkOB -- '['
            | TkCB -- ']
            | TkID Name -- identifier
```

where Name is the type of the identifiers of BLOCK. It was defined as a pseudo-terminal in the BLOCK CFG. Thus, its type must be provided by the lexical analyser. We define this type as a list of characters.

type Name = [Char]

We are now in the position to define the function scanner.
2.6. Parsing with Combinators

The **block** Parser Combinator

Before we can proceed to define the combinator parsing for the **block** language, we need to define the type of the syntax tree constructed by parsing. Observe that the scanner deals thoroughly with the literal symbols of the concrete grammar. For parsing, their occurrence is the only concern. For attribute evaluation, their existence is not a concern at all: they do not play any semantic role. Usually, literal symbols are not included in the concrete tree constructed by the parser. That is, they are “extracted” from the concrete tree. Consequently, the type of the concrete tree constructed by the parser can be easily defined by the following Haskell data type:

```haskell
data Prog = Root Stats
data Stats = OneSts LstSts | NoSts
data LstSts = AStat Stat | ConsLstSts Stat LstSts
data Stat = CDecl Name | CUse Name | CBlock Stats
```

Note that the concrete grammar of Fragment 1, with the extracted literal symbols, would define exactly the same set of values. Recall that the non-terminals are regarded as types as well. In Section 4.4 we will show how these data types are automatically derived from the **block** attribute grammar.

We are now in a position to define the parser for the **block** language. Let us start by the production applied on the root symbol. We have to denote the sequencing: `[' Stats ']'`. The literate symbol `[' '(' ')']` is passed by the scanner as symbol TkOB (TkCB). Thus, we can easily express this production using combinator `<*>` and `symbol` as follows:

```haskell
symbol TkOB <*> stats <*> symbol TkCB
```

where `stats` is the parsing function for `Stats`. Its semantics are: recognizes the token TkOB, then (combinator `<*>`) recognizes the statements (parser `stats`), then (again `<*>`) recognizes TkCB. Now, we proceed by adding a semantic function that extracts the two literal symbols and builds the concrete tree. This semantic function accepts three arguments, corresponding to the three children of the production. It ignores the first and third, i.e., the literal symbols, and applies the constructor function `Root` to the second:
the result of parsing \textit{Stats}. This parsing function and the remain functions that form the parser of BLOCK are presented next.

\begin{verbatim}
prog = sfRoot <$> symbol TkOB <$> stats <$> symbol TkCB
  where sfRoot a b c = ROOT b

stats = ONSTS <$> lststs
  <|> succeed NoSTS

lststs = sfConsLststs <$> stat <$> symbol TkSC <$> lststs
  <|> AStat <$> stat
  where sfConsLststs a b c = CONSlstSTS a c

stat = sfDcl <$> symbol TkDCL <$> ident
  <|> sfUse <$> symbol TkUSE <$> ident
  <|> sfBlk <$> symbol TkOB <$> stats <$> symbol TkCB
  where sfDcl a b = CDecl b
    sfUse a b = CUse b
    sfBlk a b c = CBLOCK b
\end{verbatim}

Program 2: The combinator based parser for the BLOCK language.

where the function \textit{ident} is a specific parser combinator that “extracts” the information associated to a token \texttt{TkID}, \textit{i.e.}, the identifier’s name. It is defined as follows:

\[
\text{ident} \ [ ] = [ ]
\]
\[
\text{ident} (x : xs) = \text{case } x \text{ of}
  \text{(TkID } n) \to [(n, xs)]
  - \to [ ]
\]

The parser is now the parser function \textit{prog}. Thus, we write

\[
\text{parser} :: \ [\text{Token}] \to [(\text{Prog}, [\text{Token}])]
\]
\[
\text{parser} :: \ prog
\]

To parse a sentence is just to apply the function \textit{parser} to the result of the scanner. So, we define the function \textit{runparser} as the composition of both functions:

\[
\text{runparser} :: \ [\text{Char}] \to [(\text{Prog}, [\text{Token}])]
\]
\[
\text{runparser} = \text{parser} \cdot \text{scanner}
\]

This function yields an empty list whenever a sentence \(s\) is not “parsable”, \textit{i.e.}, \(s \notin \mathcal{L}(CG_{\text{block}})\).

\[
\text{parsable} :: \ [\text{Char}] \to \text{Maybe Prog}
\]
\[
\text{parsable} \ s = \text{case } \text{runparser} \ s \text{ of}
  [ ] \to \text{NOTHING}
  (x : xs) \to \text{case } x \text{ of}
    (t, [ ]) \to \text{JUST } t
    - \to \text{NOTHING}
\]
Chapter 3

Multiple Traversal Functions

Summary

This chapter presents multiple traversal programs written in a purely functional style. Strict multiple traversal programs and lazy circular programs are presented.

Strict and lazy implementations for attribute grammars are presented as well. The ordered attribute grammar scheduling algorithm, the visit-sequence paradigm and the binding-tree approach are defined.

There are significant advantages in structuring our programs as multiple traversal programs. Considered in isolation, each of the traversals may be relatively simple. Consequently, they are easier to write and are potentially more reusable. By separating many distinct phases, it becomes possible to focus on a single task, rather than attempting to do many things at the same time. Furthermore, there are algorithms that rely on a multiple traversal strategy because context information must first be collected before it can be used. That is, information flows from one traversal to a following one.

Consider, for example, the problem of replacing all numbers of a list by its minimum number. A two traversal algorithm is required: a first traversal to compute the minimum number of the list, and a second one to replace all its elements by the minimum number previously computed. Compiler writers also face the challenge of organizing their compiler as a multiple traversal algorithm. In this case, multiple traversals are practically inevitable since contextual information computed in one traversal must be passed to the following ones. A typical example is the construction and use of an environment (e.g., the symbol table): it is constructed during the name analysis task and it is needed in later phases, e.g., the type checking task.

In this chapter we analyse several techniques to define multiple traversal programs in a purely functional setting. We present several programs that rely on a multiple traversal strategy and we discuss their implementation under a strict and lazy evaluation model.
3.1 Multiple Traversal Functional Programs

In order to present multiple traversal functional programs let us analyse in detail the well-known "repmin" problem. The formulation of the problem, taken from Bird [Bir84b] who originally introduced it, is as follows: consider the problem of transforming a tree, for example $T_1$

$$T_1 = (\text{Fork} (\text{Tip} 3) (\text{Fork} (\text{Tip} 2) (\text{Tip} 4)))$$

into a second tree, identical in shape to the original one, but with all the tip values replaced by the minimum tip value. For example, the previous tree $T_1$ is transformed into the tree:

$$T_2 = (\text{Fork} (\text{Tip} 2) (\text{Fork} (\text{Tip} 2) (\text{Tip} 2)))$$

In a strict and purely functional solution to this problem the tree $T_1$ is traversed twice: once to find the minimum tip value and a second time to carry out the tip replacement.

Such trees are simple binary leaf-trees. A binary leaf-tree is either a tip that contains a value or a node comprising a left and a right subtree. The type of the tree can easily be defined by the following HASKELL data type:

```haskell
data Tree a = Fork (Tree a) (Tree a) | Tip a
```

In our first solution to the repmin problem we will use this data structure to "glue" the two required functions: first, the function $tmin$ computes the minimum value of the tree. After that, the function $replace$ uses the shape of this tree and the minimum value previously computed to construct the required tree result. The straightforward solution, given in [Bir84b], is as follows:

```
tmin :: Tree Int → Int
    tmin (Tip n) = n
    tmin (Fork l r) = min (tmin l) (tmin r)

replace :: Tree Int → Int → Tree Int
    replace (Tip _) m = Tip m
    replace (Fork l r) m = Fork (replace l m) (replace r m)
```

Program 3: The repmin program.

This is a typical example where the abstract syntax tree "glues" multiple traversal functions. For this simple program the solution is clean and concise. For more complex programs, however, the strict and purely functional solution may involve the definition, construction, traverse and destruction of a large number of intermediate data structures.
As we shall see, this occurs when the syntax tree does not “match” the different traversal functions. That is, a different tree data type may be required for every traversal of a multiple traversal program. Consequently, we may have to define a set of different tree types, one for each traversal function! That is to say that, we may have to write a lot of redundant intermediate data structures.

Consider, for example, that we wish to construct a compiler that makes several traversals over the syntax tree. Each traversal computes values that are assigned to the nodes in the tree. In a purely functional solution, each traversal function would have to construct a new tree with the additional values in its nodes. Thus, we would have to define a tree type per traversal!

Let us analyse the repmin problem in more detail. The programmer with a keen eye for possible optimizations will realize that the function replace does not require the complete knowledge of the original tree, but, instead, it requires the knowledge of its shape only, and its minimum value, obviously. Observe that in the definition of replace we have used the wildcard symbol to abstract from the values stored in the tips, because they do not play any role in the second traversal of repmin. So, we can use a simpler tree, e.g., a binary shape-tree, to glue the two traversal functions. A binary shape-tree is a binary tree with no elements in its nodes, as defined next:

```haskell
data STree = SFork STree STree |
            STip
```

Now, we have to rewrite the functions tmin and replace to handle the shape-tree: the function tmin computes the minimum tip value and builds the shape-tree that represents the shape of the original tree. The function replace destructs the shape-tree and builds the desired tree result.

```haskell
tminst :: Tree Int -> (STree, Int)
tminst (Tip n) = (STip, n)
tminst (Fork l r) = (SFork nl nr, min ml mr)
where (nl, ml) = tminst l
      (nr, mr) = tminst r

transform :: Tree Int -> Tree Int
transform tree = streplace t m
where (t, m) = tminst tree
```

Program 4: The repmin program using a shape tree as the intermediate data structure.
data structure is used to “glue” multiple traversal functions.

We return now to our running example: the BLOCK language. We wish to construct a purely functional implementation for the semantic analyser of BLOCK. This language does not require a declare-before-use discipline, and, as we have explained in Section 2.3.3, it induces a program that traverses every block twice: a first traversal function has to collect context information, i.e., the environment, and then, a second traversal function uses that information to compute the desired result, which is the list of errors in this case. In order to not complicate our solution too much we shall focus the implementation on the abstract syntax of BLOCK. The abstract grammar shown in Fragment 2 is defined in HASKELL by the following data types:

\[
\begin{align*}
\text{data } P & = R \quad \text{Its} \\
\text{data } \text{Its} & = \text{ConsIts } \text{It } \text{Its} \\
& \quad \mid \text{NilIts}
\end{align*}
\]

This abstract syntax tree is the “glue” of the two traversal functions. We call the function that builds the environment \( \text{buildEnv} \), and the function that returns the errors \( \text{compErr} \). The function \( \text{buildEnv} \) takes three arguments: the syntax tree, the initial collection of declarations and the lexical level, and returns the total environment. Recall that the environment is an association list from identifiers to lexical levels. So, the lexical level is needed for the construction of the environment. The function \( \text{compErr} \) takes two arguments: the syntax tree and the total environment, and returns the list of errors. So, we would expect the following types for these two functions:

\[
\begin{align*}
\text{buildEnv} & : : \text{Its } \rightarrow \text{Env } \rightarrow \text{Int } \rightarrow \text{Env} \\
\text{compErr} & : : \text{Its } \rightarrow \text{Env } \rightarrow \text{Err}
\end{align*}
\]

, and the function applied on the root of the abstract syntax tree is to be defined as follows:

\[
\begin{align*}
\text{block} & : : P \rightarrow \text{Err} \\
\text{block} \ (R \ \text{its}) & = \text{errs} \\
\text{where} & \ \text{env} = \text{buildEnv} \ \text{its } ] / 0 \\
& \ \text{errs} = \text{compErr} \ \text{its } \ \text{env}
\end{align*}
\]

Before we proceed to code the remaining functions, let us first analyse the function \( \text{compErr} \). This function has to synthesize the correct list of errors occurring on each block, i.e., it has to detect duplicated declarations and invalid uses (the AG specification of this function is presented in Fragment 3). We start by analysing the simplest case: the detection of invalid uses.

To detect whether the use of an identifier, say \( n \), is valid or not, the function \( \text{compErr} \), simply needs to look up for \( n \) in the total environment, i.e., in its argument. So, no problem

---

1In Section 4.3 we will return to the problem of mapping the concrete tree, which is constructed by the combinator parser (Section 2.6.1), into its abstract representation.

2In Section 4.3 we will automatically derive these type constructors from the AG specification.
arises in this case. However, there is a problem when detecting duplicate declarations: the function \texttt{compErr} needs to look up the declared identifier, say \texttt{n}, on the list of declarations that were collected until the declaration of \texttt{n} is reached, and not in the total environment. Such a collection of identifiers is available in the first traversal only. As a consequence, the value of the collected list of declarations and the associated value of the lexical level have to be passed from the first traversal to the second one. This is a typical example where we need to assign values to the nodes of the original abstract syntax tree. In other words, we need to define a specific intermediate data structure to explicitly pass such values from the first function to the second one and, in this way, "gluing" both traversal functions. The algebraic data type declaration of this tree is presented next. We use superscripts to denote the traversal function for which the type constructor is intended to. Thus, \texttt{Its} denotes the (tree) type constructor for the second traversal and \texttt{ConsIts} and \texttt{NilIts} are the data constructors of that type.

\begin{table}
\begin{align*}
\texttt{data Its}^2 &= \texttt{ConsIts}^2 \texttt{It}^2 \texttt{Its}^2 \\
&\quad | \texttt{NilIts}^2 \\
\texttt{data It}^2 &= \texttt{DECL}^2 \texttt{Name Env Int} \\
&\quad | \texttt{USE}^2 \texttt{Name} \\
&\quad | \texttt{BLOCK}^2 \texttt{Its}^1 \texttt{Int}
\end{align*}
\end{table}

The constructor \texttt{DECL}^2 includes the types of the list of declarations (type \texttt{Env}) and the type of the level (type \texttt{Int}), \textit{i.e.}, the values that have to be passed from the first to the second traversal function. The remaining constructors are similar to the original tree data constructors. The exception is the constructor \texttt{BLOCK}^2; it includes the constructor type \texttt{Its}^1 (which defines the body of the inner block) and the primitive type \texttt{Int}, which is the type of the block level. The body of the inner block is defined by the tree constructor type for the first traversal, \textit{i.e.}, constructor type \texttt{Its}^1. Note that an inner block is processed only in the second traversal of its outermost block since the initial collection of identifiers of the inner block is the total environment of the outermost one (see the dependency graph of production \texttt{BLOCK} in Figure 2.2). As a result, the tree type for the second traversal of a (outermost) block has to contain references to the tree constructor for the first traversal. The level of a block also has to be passed explicitly from the first to the second traversal, since it is available in the first traversal to \texttt{BLOCK} and it is passed down to its body, in the second one.

The type of function \texttt{buildEnv} and \texttt{compErr} are now defined as follows:

\begin{align*}
\texttt{buildEnv} &:: \texttt{Its}^1 \to \texttt{Env} \to \texttt{Int} \to (\texttt{It}^2, \texttt{Env}) \\
\texttt{compErr} &:: \texttt{Its}^2 \to \texttt{Env} \to \texttt{Err}
\end{align*}

where \texttt{Its}^1 is a type synonymous with type \texttt{Its}. In \texttt{HASKELL} we would write \textit{type} \texttt{Its}^1 = \texttt{Its}. The functions are now glued by the intermediate tree. They look as follows:

\begin{align*}
\texttt{block} &:: \texttt{P} \to \texttt{Err} \\
\texttt{block} (\texttt{R its}) &= \texttt{errs} \\
\texttt{where} &\quad (\texttt{newtree, env}) = \texttt{buildEnv its} \| 0 \\
\texttt{errs} &= \texttt{compErr newtree env}
\end{align*}
Program 5: The BLOCK multiple traversal function.

Functional programs are known for having simple and concise solutions. Occasionally, however, using a functional language may lead to complex and extensive solutions. As the previous examples have shown, constructing multiple traversal programs in a strict and purely functional setting can be both complex and inefficient.

Firstly, for large programs relying on a large number of traversals over a data structure, a possible large set of redundant intermediate data structures may have to be defined in order to glue the different traversal functions. Furthermore, the construction, traverse, and destruction of such intermediate data structures may degrade the efficiency of such functional programs. For example, we have developed several programs that perform multiple traversals over data structures which would be extremely complex to develop in a functional language. In [SAS98] we have presented a pretty printing combinator library whose algorithm relies on a four traversal strategy. In the strict implementation of such combinators we would have to define four data types to glue the traversal functions: one per traversal. The attribute grammar system we developed, the Lrc system [KS98], performs eleven traversals over the abstract syntax tree. Its strict and functional implementation would require the definition of eleven redundant data types! Obviously, we did not write all the functional implementation of such programs, neither did we write all those redundant type definitions. Instead, we have derived such strict functional programs from attribute grammars. In Chapter 4 we will present new techniques to derive strict functional implementations from attribute grammars.

Secondly, it can be extremely complex to determine which intermediate values are needed in ensuing traversals and how such traversals are glued together. That is to say that, it can be extremely difficult to schedule statically the computations of the functional program.
Let us consider again the two traversal program for the BLOCK language presented in Program \(5\). In this very simple example it was not difficult to determine which values had to be passed explicitly from the first to the second traversal. However, the program we constructed can be very inefficient. Observe that the whole list of declarations has to be retained from the first to the second traversal, for all instances of data constructor Decl! This may result in a huge memory leak since a possibly large number of copies of that list may have to be retained for the second traversal. A more efficient program is obtained if we compute the error in the first traversal and pass its value to the second one, exactly as defined by our original algorithm. When writing Program \(5\) we were “misled” by the type of the functions: we had scheduled the computations according to their types.

In more complicated examples, like the pretty printing library and the LRC system, it would be extremely difficult to statically define the intermediate data types and to define an efficient schedule of the computations.

In the next section, we present a different style to write multiple traversal programs avoiding the definition of all redundant intermediate data structure. This style is known as functional circular programming.

### 3.2 Circular Programs

Let us consider again the repmin problem. The solutions presented thus far make two traversals over the tree: one performed by \(tmin\) to find the minimum value and one performed by \(replace\) to make the replacements. A two traversal algorithm is not essential to solve the repmin problem. An alternative solution can, on a single traversal, compute the minimum tip value and, at the same time, store the addresses of the tip nodes in a linear list. After that, the program inserts the minimum value in each of the nodes in the list. The basic idea of this optimization is shown in next the figure.

In an imperative setting, this single traversal algorithm could easily be implemented, although it would require a major revision of the original program. In a purely functional setting, how can we achieve this single traversal program? The answer is to use the so-called circular programs. Circular programs are proposed by Bird [Bir84b] as an elegant and efficient technique to eliminate multiple traversals of data structures.

Circular programs, as the name indicates, are characterized by having what appears to be a circular definition: arguments in a function call depend on results of that same call. That is, they contain definitions of the form: \((\ldots, x, \ldots) = f \ldots x \ldots\). The single traversal repmin program, presented in [Bir84b], is as follows:

---

3To help the reader find the programs being referred to, page numbers are given as subscripts.
Chapter 3. Multiple Traversal Functions

repmin :: Tree Int → Int → (Tree Int, Int)
repmin (Tip n) m = (Tip m, n)
repmin (Fork l r) m = (Fork t1 t2, min m1 m2)
  where (t1, m1) = repmin l m
        (t2, m2) = repmin r m

transform :: Tree Int → Tree Int
transform t = nt
  where (nt, m) = repmin t m

Program 6: The repmin circular program.

A single traversal is obtained because a single function, i.e., repmin, is applied to the (tree) argument of transform. Although the circular definitions seems to induce both cycles and non-termination of those programs, the fact is that using a lazy language, the lazy evaluation machinery may still be able to determine, at runtime, the right order to evaluate such circular definitions. Actually, circular programs are a famous example which nicely exploits and demonstrates the power of lazy evaluation.

Circular programs provide a concise technique to implement multiple traversal programs under a lazy language. They eliminate the need of defining, constructing and traversing redundant intermediate data structures that glue multiple traversal functions. Observe that the circular repmin program is not only the circular version of the original repmin (Program 3.11), but also the circular version of repmin that uses the shape-tree (Program 3.15). In this case, the intermediate shape-tree is not included in the circular program. Thus, circular programs provide also an interesting way to achieve deforestation [Wad90]. Circular programs, however, can be very hard to design and difficult to understand due to the counterintuitive circular definitions. Specially if we consider algorithms relying on a large number of traversals. In the latter case, a circular implementation of such algorithms would contain function calls with many arguments, depending on many results of the same call. Furthermore, the advantages of a compositional style of programming are lost within the style of circular programming: circular programs have to be written and understood as a whole, and, as a consequence, they loose the clarity and high level modularity of the compositional style. Finally, the main drawback of circular programs is that they are prone to induce incorrect solutions, which lead to circular programs that do contain a cycle and do not terminate when executed [Bir84b].

In order to overcome these drawbacks, different techniques to obtain circular programs are suggested by several authors [Bir84b, Joh87, KS87]. Bird starts with the straightforward multiple traversal programs and uses a source-to-source transformation to derive circular programs. Johnsson and Kuiper and Swierstra start to express multiple traversal programs as attribute grammars and then, they simple map attribute grammars into circular programs. In Section 3.5, we will present the attribute grammar approach. Next we
3.2. Circular Programs

briefly describe Bird’s transformation technique.

Bird’s transformation technique is based on the combination of three different techniques: tupling, fold/unfold transformation and circular programming. To present this transformation we show the derivation of repmin given in [Bir84b].

*Tupling*: given the common recursive pattern of functions replace and tmin, we tuple them into one function repmin, which does the same work as the previous two functions.

\[
\text{repmin } t \ m = (\text{replace } t \ m , t\text{min } t)
\]

*Fold/unfold transformation*: now, the function repmin is synthesized from the original one by a standard application of the fold/unfold method. We have to consider two cases:

1. \[
\text{repmin(Tip } n) \ m = (\text{replace } (\text{Tip } n) \ m , t\text{min } \text{Tip } n))
= (\text{Tip } m, n)
\]
2. \[
\text{repmin(Fork } l \ r) \ m = (\text{replace } (\text{Fork } l \ r) \ m , t\text{min } \text{Fork } l \ r))
= (\text{Fork } (\text{replace } l \ m) (\text{replace } r \ m), \text{min } (t\text{min } l) (t\text{min } r))
= (\text{Fork } t1 \ t2, \text{min } m1 \ m2)
\quad \text{where } (t1, m1) = \text{repmin } l \ m
\quad (t2, m2) = \text{repmin } r \ m
\]

*circular programming*: finally, circular programming is used to couple the two components of the result value of repmin to each other. Consequently, we obtain the circular definition of transform:

\[
\text{transform } t = nt
\quad \text{where } (nt, m) = \text{repmin } t \ m
\]

As Bird mentions in his paper, this transformation does not guarantee termination. The problem is not induced by the circular transformation of the last step, as we would expect, but by the fold/unfold transformation. It is well known that the fold/unfold transformation preserves partial correctness only: whenever the transformed program terminates, it is guaranteed to return the same results as the original program. The repmin program terminates because its second argument is demanded only when the first argument, i.e., the final tree, is required to be output. It is possible to synthesize circular programs that do not terminate, simply because such arguments are demanded too early. See [Bir84b] for examples where the transformation yields incorrect programs. Nevertheless, the formal way to establish that circular programs terminate is by fixed-point induction [Bir84b].

Another disadvantage of this transformation is that it can only be applied to functions that exhibit a common pattern of recursion. This is required by the tupling step. As a result, this transformation does not directly apply to the second version of repmin (Program 4) because its functions use two different recursive patterns.
3.3 Catamorphisms

Functional programmers usually structure their multiple traversal programs by recursion on an algebraic data type. A function is defined for each data type constructor, specifying how such constructor is to be evaluated. This style of structuring recursive programs can be captured by the so-called catamorphisms \cite{Mal90} or generalized fold operators \cite{MJ95}. For example, many functions on lists simply recurse over a list. Such a common pattern of recursion is usually captured in a fold operator, e.g., `fold` in ML or `foldr` in Haskell. Fold operators express common recursion over recursive data types. They provide an elegant and concise way of writing traversal functions. Moreover, catamorphisms satisfy a number of laws that support calculating with programs \cite{BJJM98}. Using such laws, efficient programs can be calculated from inefficient “specification” programs.

Let us start by presenting catamorphisms. A catamorphism \( \text{cata}^T(f_1, \ldots, f_n) \) can be viewed as a function that replaces every constructor \( C_i \) in a data structure of type \( T \) with a corresponding function \( f_i \). In other words, a catamorphism replaces constructors by functions. A catamorphism is naturally derived from a recursive data type. The catamorphisms for binary leaf-trees and shape-tree are presented next.

\[
\begin{align*}
\text{foldBTree} &: (a \to b, b \to b \to b) \to \text{Tree} a \to b \\
\text{foldBTree} \quad (\text{tip, fork}) &= \text{fold} \\
\text{where} \quad \text{fold} (\text{Fork} \ l \ r) &= \text{fork} (\text{fold} \ l)(\text{fold} \ r) \\
\text{fold} (\text{Tip} \ i) &= \text{tip} \ i \\
\text{foldSTree} &: (a, a \to a \to a) \to \text{STree} \to a \\
\text{foldSTree} \quad (\text{stip, sfork}) &= \text{fold} \\
\text{where} \quad \text{fold} (\text{SFork} \ l \ r) &= \text{sfork} (\text{fold} \ l)(\text{fold} \ r) \\
\text{fold} (\text{STip}) &= \text{stip}
\end{align*}
\]

Program 7: Catamorphisms on tree structures.

Just as with the fold on lists, we can now define many functions in terms of folding binary trees with \( \text{foldBTree} \), instead of using explicit recursion. The function \( t\text{min} \), for example, is known as the catamorphism \( \text{cata}^{\text{Tree}}(\text{id}, \text{min}) \). Thus, the \text{repm} in problem can be defined by means of catamorphisms. The function \( t\text{min} \) and \( \text{replace} \) are defined as follows:

\[
\begin{align*}
\text{tmin} &= \text{foldBTree} \ (\text{id}, \text{min}) \\
\text{replace} \ m &= \text{foldBTree} \ (\lambda x \to \text{Tip} \ m, \text{Fork})
\end{align*}
\]

The advantage of expressing a function by means of a catamorphism (or folds) is that the definition becomes non-recursive, and therefore much easier to manipulate. Using the shape-tree as the intermediate data structure, we have the following catamorphic functions:

\[
\begin{align*}
\text{tminst} &= \text{foldBTree} \ (\ \ m \ \to \ (\text{STip} \ , \ m) \\
\text{, \ \ l \ \to \ (\text{SFork} \ (\text{fst} \ l) \ (\text{fst} \ r) \ , \ \text{min} \ (\text{snd} \ l) \ (\text{snd} \ r) \ ) \\
\text{streplace} \ m &= \text{foldSTree} \ (\text{Tip} \ m, \text{Fork})
\end{align*}
\]

3.3.1 Deriving a Circular Program

In this section we show how circular programs can be obtained using catamorphisms. It is basically Bird’s approach, but instead of focusing the transformation on tupling and
the fold/unfold transformation, we focus our transformation on well-known techniques to manipulate catamorphisms. The last step of Bird transformation is used to obtain the circular definition.

Let us define a variant of the repmin problem: the repminrror. The repminrror is similar to the repmin. It computes a new tree whose tip nodes contain the minimum value of the original tree. The single difference is the shape of the resulting tree. The repminrror requires that the shape of the resulting tree is the mirror shape of the original tree. That is, the two children of a fork node have to switch positions. Program $\text{8}$ is the straightforward multiple traversal program.

In order to demonstrate our techniques we have defined a very inefficient program: the mirroring of the tree is computed in the first traversal of the original tree, i.e., it works as the intermediate tree which glues both traversals. This inefficiency could be eliminated by constructing the mirror tree during the replacement.

We proceed to derive a circular program for repminrror. We start by defining the above functions as catamorphisms. This transformation can be automatically obtained by using the techniques described in [LS95].

Let us analyze the body of function transform. The two calls to $tmin$ and $mirror$ do not interfere with each other. Moreover, they are now defined by a catamorphism over the same data type, and, as a consequence, we may fuse both functions using the banana-split theorem [BJJM98]. The tupling analysis described in [HITT97] can be used to determine which catamorphisms to tuple.

Observe that this definition of mirrormin is similar to the function tminst expressed as a catamorphism. In this case, however, the banana-split theorem has just calculated the tupled catamorphism for us. Let us now abstract from the constructors that contribute to the result of mirrormin. As a result we get:
\[
mirrormin \ (\text{tip}, \text{fork}) = \text{foldBTree} \quad (\begin{array}{ll} \text{x} & \rightarrow & (\text{tip} \ x, \ x) \\ \text{l} \ r & \rightarrow & (\text{fork} \ (\text{fst} \ r) \ (\text{fst} \ l), \ \text{min} \ (\text{snd} \ l) \ (\text{snd} \ r)) \end{array})
\]

and, \text{transform} becomes:

\[
\text{transform} \quad t = \text{replace} \ m \ mt \\
\text{where} \quad (mt, m) = \mirrormin \ (\text{Tip}, \text{Fork}) \ t
\]

By definition of \text{replace} we have:

\[
\text{transform} \quad t = \text{foldBTree} (\begin{array}{l} x \rightarrow \text{Tip} \ m, \text{Fork} \end{array}) \ mt \\
\text{where} \quad (mt, m) = \mirrormin \ (\text{Tip}, \text{Fork}) \ t
\]

Function \text{transform} explicitly shows that \text{mirrormin} builds a data structure that is consumed by the catamorphism \text{foldBTree}. The reader familiar with fusion and hylomorphisms [MJ95, BJJM98], would expect that such a redundant intermediate data structure could be eliminated using standard fusion laws, or by using the cata/build rule [LS95]. The \text{cata/build} rule, for example, formalises the intuition that first building an intermediate data structure using a set of constructors \text{C}_i, and then replacing each constructor by a function \text{f}_i, yields the same result as building the resulting value using function \text{f}_i directly. The standard \text{cata/build} rule, however, does not apply in this case, because \text{mirrormin} computes the minimum value of the tree, besides constructing the intermediate mirroring tree. Thus, we need a kind of (state) monadic \text{cata/build} rule here, but we have not found such a rule in the literature [Par98]. Although we cannot apply standard calculation rules to eliminate the intermediate tree, we can achieve such an elimination by constructing a circular program, that is, by just applying Bird’s circular transformation. The resulting circular program is presented next.

\[
\text{transform} \quad t = nt \\
\text{where} \quad (nt, m) = \mirrormin \ (\begin{array}{l} x \rightarrow \text{Tip} \ m, \text{Fork} \end{array}) \ t
\]

Under what circumstances can we expect that this transformation yields a well-formed and correct program? Firstly, the semantics of the functional language have to be based on a lazy evaluation mechanism. With lazy evaluation the (pseudo) circular structure can be built and \text{mirrormin} can be ( lazily) evaluated, without computing the value of the argument \text{m} until it is really required. Secondly, the circular definition cannot introduce any \textit{real} circularity, \textit{i.e.}, no value can be defined in terms of itself. This is the case in the above program. It can be easily proved by fixed-point induction, exactly as it is done in Bird’s paper [Bir84b]. Next, we show how the lazy evaluation mechanism works for \( t = \text{Fork} \ (\text{Tip} \ 3) \ (\text{Tip} \ 2) \).
3.4 Attribute Grammars in a Functional Setting

At each step, the lazy engine only expands the part of the expression that is required to continue the derivation. As we can see in this derivation, there is nothing really circular in the resulting program: each value is defined in terms of other values, and no value is defined in terms of itself. Note that this is the expected behaviour since we start the transformation with a correct and non-circular program. However, under a call by value semantics the value of the argument \( m \) of function \( \text{mirrormin} \) is demanded before the value of this function can be computed. As a consequence, in a strict functional language, the program above does not terminate.

3.4 Attribute Grammars in a Functional Setting

It should now be clear why writing programs which rely on a multiple traversal strategy in a functional language can lead to both complex and unclear solutions: under a strict functional language a possible large number of redundant data structures may have to be defined. Under a lazy functional language, elegant and concise circular programs can be used and no intermediate data structure has to be defined. This conciseness, however, comes at a price: the compositional style of programming in a functional setting is lost. Furthermore, circular programs can be extremely complicated to write and the programmer may easily write an incorrect program, \( e.g., \) a program that does not terminate. As we have shown in the previous section, circular programs can be correctly derived from their (strict) multiple traversal counterparts. Unfortunately, such an approach does not avoid the need of writing redundant data structures and defining the scheduling of computations.

Multiple traversal functional programs can be efficiently and, also, easily written within the attribute grammar formalism. The writer needs not to concern himself with partitioning the program into a number of traversals, as in the strict functional approach. The order of evaluation is derived automatically, using standard attribute grammar techniques. Con-
sequently, no scheduling and no intermediate data structures have to be defined. Indeed, attribute grammars can be viewed as a particular style of writing lazy (circular) programs [Joh87, KS87, Aug93, SSKP96, SA98]. However, attribute grammars have simple and obvious solutions where the corresponding circular program would have both complex and counterintuitive solutions.

In order to show these properties of attributes grammars let us define the \textit{repmin} problem within the AG formalism.

\begin{align*}
\text{Tree} & \quad <\downarrow \text{min} : \text{Int} > \\
\text{Tree} = & \quad \text{Tip} \quad \text{Int} \\
& \quad | \quad \text{Fork} \quad \text{Tree} \quad \text{Tree} \\
& \quad | \quad \text{Tree}_1.\text{min} = \text{min} \text{Tree}_2.\text{min} \quad \text{Tree}_3.\text{min} \\
\text{R} & \quad <\downarrow \text{new} : \text{Tree} > \\
\text{R} = & \quad \text{Root} \quad \text{Tree} \\
\text{Tree}.\text{m} & \quad = \quad \text{Tree}.\text{min} \\
\text{R}.\text{new} & \quad = \quad \text{Tree}.\text{new}
\end{align*}

\textbf{Fragment 13:} The repmin attribute grammar.

From this specification different functional implementations can be derived. As we shall see later, all the functional \textit{repmin} programs, \textit{i.e.}, \textit{Program 3}, \textit{Program 4} and \textit{Program 6}, can be automatically derived from this AG. Observe that, the counterintuitive circular definition of the circular \textit{repmin} program corresponds in the AG formalism to a normal dependency from the synthesized attribute \textit{Tree}.\text{min} to the inherited attribute \textit{Tree}.\text{m} of the same non-terminal symbol \textit{Tree}. Attribute grammars provide the necessary abstraction to specify circular definitions. Consequently, designing circular programs becomes a lot simpler if one considers them as the representation (in a lazy functional language) of an attribute grammar [KS87, Joh87, SSKP96]. Furthermore, standard attribute grammar techniques can be used to statically compute an order of evaluation. Such order guarantees that the correspondent circular program is correct and always terminates when executed.

Lazy evaluators, and circular programs in particular, can be directly used as the functional implementation of attribute grammars. Under lazy evaluation only the attribute instances that do contribute to assign the meaning to a particular sentence are computed. In this case, the termination problem with recursive evaluation of circular attribute grammars becomes less severe: static cycles induced by the attribute equations are dynamically eliminated by the lazy machinery. As a consequence, a lazy implementation of a circular attribute grammar, or ill-defined AG, may be able to assign viable semantics to sentences of the language. The solution is only partial, though. Such an implementation does not guarantee termination for the lazy decoration of all possible attributed trees which are assigned by the circular AG. See [Paa95] for a simple and detailed example where lazy evaluators do not terminate when executed with a sentence of the circular attribute grammar under
consideration.

A more general approach to handle a larger class of cyclic dependencies is based on fixed-point evaluation [Far86, Jon90, Alb91a, Paa95]. This method is based on an iterative computation scheme which approximates the least fixed point of every circularly-defined attribute instance. Let $\alpha$ be such an attribute instance. The basic idea of this method is to compute the least fixed point of $\alpha$ by starting from an initial special bottom value $\perp$, and by successively approximating the value of $\alpha$ until it is not changed anymore. That is, the least fixed point has been reached. To allow approximations of the least fixed-point to be associated with every attributed instance, the attribute grammar framework has to be extended: the domains of $\alpha$ must be a complete partial order, having $\perp$ as the smallest element. Furthermore, to guarantee that the execution of every attribute equation improves the value of the attribute instance being computed it is required that the semantic function, within the attribute equation, is monotonic [Alb91a].

Both lazy and fixed-point evaluators are partial solutions only, because they are able to process a subclass of circular attribute grammars [Paa95]. Our goal is to derive correct functional programs from attribute grammars. That is, we wish to be able to statically analyse an attribute grammar and derive a functional program that is correct and always terminates when executed with all possible sentences of the underlying AG. Non-termination is induced by circular dependencies. Thus, we may use well-known algorithms to statically detect circularities within attribute grammars and reject the circular ones. Such approach loses some generality because some (circular) grammars for which viable semantics could be assigned by lazy evaluators are rejected. Although less general, this approach has the main advantage of moving the detection of circularities from the dynamic to the static realm. As a result, real circularities that do cause non-termination of lazy evaluators are always detected statically. That is to say, the resulting lazy evaluators are correct and they do terminate. We call such lazy evaluators, well-formed lazy evaluators or, when expressed as circular programs, well-formed circular programs.

However, to derive strict functional programs the order of the computations has to be statically defined by a scheduling algorithm. There are several scheduling algorithms for attribute grammars [Kas80, Pen94, NGI+99]. These algorithms apply to a proper sub-class of non-circular attribute grammars, the so-called L-Ordered Attribute Grammars [Kas80]. Nevertheless, non-circular and L-ordered attribute grammars are large enough to express all grammars of practical importance. Figure 3.1 shows the relation between attribute grammars and functional programs.

Attribute grammars and lazy programs are closely related. Consequently, we may easily map an (not necessarily well-defined) attribute grammar directly into a circular program, under a lazy programming language. In this case, we have to prove that the resulting program is correct. The way to establish that a circular program is correct is to look at the partial approximations of the program as defined by the fixed point theory [Bir84b], or, to use standard attribute grammar techniques [SSK99]. In both cases, we need to use techniques which are also common practice within the attribute grammar formalism!
Attribute grammar techniques can also be used to transform a lazy circular program into a strict multiple traversal program [SSK99]. As a result, the programmer is no longer restricted to a lazy language in order to write such concise circular programs. In [SSKP96] we have presented a technique to derive automatically the strict solutions from the lazy ones. We call this approach strictification of circular programs, and it can be applied to circular programs for which we can statically define an order of evaluation. In terms of attribute grammars, we would say that the underlying attribute grammar is ordered.

3.4.1 The Circularity Test

The basic idea behind constructing and analysing attribute dependency graphs is to compute a single attribute dependency representation that holds for all the possible attributed trees assigned to the grammar. As a consequence, this graph makes it possible to analyse statically the evaluation properties of the attribute grammar, without considering any concrete tree of the language. Thus, we do not restrict our computations to a single derivation tree and a single node of this tree, but, on the contrary, we consider any node of any possible attributed tree. For every $X \in V$ we compute a dependency graph $IS\_SET(X)$ describing how the synthesized attributes of $X$ depend on the inherited attributes of $X$, in every context where $X$ may occur. That is, we define a set of dependency patterns that reflect how the information flows from the inherited to the synthesized attributes at every node $N$ labelled by $X$. We call these graphs the Inherited-Synthesized (IS) dependency graphs. The next definition of the circularity test is taken from [Alb91b], with slight changes in the notation adopted.

Let us consider the production $p : X_0 \rightarrow X_1 X_2 \ldots X_n$ applied on $X_0$. Suppose that for every $X_i$, with $1 \geq i \leq n$, we have an inherited-synthesized dependency graph $IS\_SET(X_i)$ that records the dependencies between the attributes of $A_{nont}(X_i)$. Then, we would like to compute the dependency set $IS\_SET(X_0)$ on $A_{nont}(X_0)$ reflecting all possible dependency patterns on $A_{nont}(X_0)$. This is achieved in three steps: first we paste the dependencies $D_1 \ldots D_n$, with $D_i \in IS\_Set(X_i)$, into $DP(p)$. After that, we compute a transitive closure of the resulting graph. Finally, we project that graph to extract the
Before we define the $IS\_Set(X)$, let us first define how we paste and project dependency graphs. By $DP(p)[D_1, D_2, \ldots, D_n]$ we denote the directed graph that is obtained from $DP(p)$ by pasting the graphs $D_i$ into $DP(p)$. This graph is constructed as follows: we add an arc from attribute occurrence $\langle p, i, a \rangle$ to $\langle p, i, b \rangle$ whenever there is an arc from $X_i.a$ to $X_i.b$ in $D_i$, with $1 \leq i \leq n$. Pasting is the process that takes the dependencies of attributes and converts them into dependencies on attribute occurrences. Projection does the opposite: given a dependency graph $d$, the vertices of which are $O_{pr}(p)$, and an index of a non-terminal occurrence $i$, $0 \leq i \leq |p|$, $proj(d, i)$ extracts the dependencies for $\langle p, i \rangle$ and converts them into dependencies for $X_i$. This process is defined as follows:

$$proj(d, i) = \{X_i.a \rightarrow X_i.b | \langle p, i, a \rangle \rightarrow \langle p, i, b \rangle \in d\}.$$ 

We are now in a position to define the set of IS dependency graphs for every $X \in V$.

$$IS\_Set(X) = \{ proj(DP(p)[D_1, D_2, \ldots, D_n]^+, 0) \mid p \in P \land lhs(p) = X \land |p| = n \land D_i \in IS\_Set(X_i) \}$$

Observe that for each $X \in V$, $IS\_SET(X)$ is finite since $A_{nont}(X)$ is finite. After computing the $IS\_Set(X)$ for all grammar symbols $X \in V$, the circularity test is straightforward: it simply detects whether the pasting of the IS dependency graphs into the graphs of productions $DP(p)$ results in a cycle graph or not. If no pasted graph contains a cycle, the attribute grammar is non-circular. This rule is defined as follows: for each $p : X_0 \rightarrow X_1 X_2 \ldots X_n$ and for each tuple $(d_1, \ldots, d_n) \in IS\_Set(X_1) \times \cdots \times IS\_Set(X_n)$ the dependency graph $d = DP(p)[d_1, \ldots, d_n]$ is constructed. If $d$ contains a cycle then the grammar is circular.

The circularity test can be used to statically prove that the BLOCK AG is non-circular and, thus, well-defined. As a result, a well-formed lazy evaluator can be derived for our AG. Since we also wish to derive a strict functional evaluator for BLOCK, in Section 3.6.2 we will use Kastens’ ordered algorithm [Kas80] to prove the well-definedness of the grammar. The next section presents the mapping from attribute grammars into lazy circular evaluators.

### 3.5 Attribute Grammars as Circular Programs

Attribute grammars can be easily mapped into lazy circular programs [Joh87, KS87]. This section presents the mapping given in [KS87].

For each non-terminal $X \in N$ a function $evalX$ is derived. The arguments of $evalX$ are the tree (of type $X$) and an additional argument for each attribute in $A_{inh}(X)$. Its result is a tuple of the synthesized attributes of $X$, $A_{syn}(X)$. The function has the following signature:

$$evalX : X \rightarrow (T \ inh_1) \rightarrow \cdots \rightarrow (T \ inh_k) \rightarrow (T \ syn_1, \ldots, T \ syn_l)$$

with $A_{inh}(X) = \{ inh_1, \ldots, inh_k \}$ and $A_{syn}(X) = \{ syn_1, \ldots, syn_l \}$. 
For every production $P : X_0 \rightarrow X_1 \ldots X_s$ an alternative function definition for $eval_{X_0}$ is defined. It is selected by using pattern matching on $P$. The pattern variables are the right-hand side of $P$ prefixed with “t”. The alternative function definition looks as follows:

$$eval_{X_0}(p_{tX_1} \ldots p_{tX_s}) inh_1 \ldots inh_k = (syn_1, \ldots, syn_l)$$

where body

The body of $eval_{X_0}$ is the translation of the attribute equations of $P$, $E(P)$. It is constructed as follows: for every semantic equation defining an attribute occurrence $X_q.a = f(\ldots)$, $0 \leq q \leq s$, in $E(P)$, an equation $a_q = f(\ldots)$ is generated. Attribute occurrences $X_r.a$ in $f(\ldots)$ are replaced by $a_r$, $0 \leq r \leq s$. For every non-terminal symbol $X_j \in N$, $1 \leq j \leq n$, a definition $(syn_1, \ldots, syn_k) = eval_{X_j} p_{tX_j} inh_1 \ldots inh_l$, with $A_{inh}(X_j) = \{inh_1, \ldots, inh_k\}$ and $A_{syn}(X_j) = \{syn_1, \ldots, syn_l\}$ is generated.

Eval $[460]$ presents the lazy attribute evaluator obtained by mapping the BLOCK attribute grammar into a circular program according to the above rules. Note that the BLOCK AG is well-defined (the proof will be given in Section 3.6.2) and as a consequence the circular program is well-formed.

Eval $[1]$ presents the lazy attribute evaluator obtained by mapping the BLOCK attribute grammar into a circular program according to the above rules. Note that the BLOCK AG is well-defined (the proof will be given in Section 3.6.2) and as a consequence the circular program is well-formed.

This attribute evaluator contains two circular definitions: as we have explained before, these counterintuitive definitions are induced by the two normal dependencies from the synthesized attribute $dcl0$ to the inherited $env$ of symbol $Its$ (see Figure 2.2).
3.6 Attribute Grammars as Multiple Traversal Programs

In the previous section we have discussed how attribute grammars can be implemented under a lazy attribute evaluation model. We shall now discuss another attractive property of attribute grammars: their efficient implementation under a strict attribute evaluation model. A strict model of attribute evaluation is attractive for two main reasons: firstly, because we obtain very efficient implementations in terms of memory and time consumption. Secondly, because a rather simple strict attribute evaluator can be derived from an attribute grammar. Furthermore, such an evaluator is not restricted to a lazy semantics execution model. Indeed, it can be correctly executed under both a strict and a lazy execution model.

The strict model of evaluating attribute grammars relies on a multiple traversal attribute evaluator. Such a strict evaluator performs several traversals over the abstract syntax tree in order to assign a meaning to that tree. There are standard techniques that schedule the attribute computations into different traversals of the evaluator. Traditional implementations of this strict model of attribute evaluation rely on side effects to obtain efficient (imperative) implementations. On the contrary, we aim at constructing programs that do not rely on any side effect. In other words, we aim at deriving strict and purely functional attribute evaluators.

3.6.1 L-Ordered Attribute Grammars

This section discusses classes of attribute grammars for which strict attribute evaluators can be derived. That is, attribute grammars whose evaluation order is determined statically, based on the attribute dependencies defined in the grammar.

The algorithms that compute the attribute evaluation order establish the number of visits and an interface for every non-terminal \( X \) of the attribute grammar. We denote the interface of non-terminal \( X \) by \( \text{Interface}(X) \) and it is defined as follows:

\[
\text{Interface}(X) = [(inh_1, syn_1), \ldots, (inh_n, syn_n)]
\]

with \( inh_i \subseteq A_{inh}(X) \) and \( syn_i \subseteq A_{syn}(X) \). Thus, \( \text{Interface}(X) \) specifies, for every visit to \( X \), which inherited attributes in \( A_{inh}(X) \) are used and which synthesized attributes in \( A_{syn}(X) \) are computed. Roughly speaking, it fixes the types for every of the traversal functions for term \( X \).

\( \text{Interface}(X) \) induces a partial order on the attributes \( A(X) \). The largest class of attribute grammars for which strict attribute evaluators can be derived is the class of partitionable attribute grammars. Informally, an attribute grammar is partitioned if for each non-terminal there is an interface, such that in any context of the non-terminal its attributes are computable in an order which is included in the partial order induced by the interface. Let \( PO(X_i) \) be the partial orders induced by \( \text{Interface}(X_i) \). An attribute
grammar is partitionable if for every production $p : X_0 \to X_1 X_2 \ldots X_n$ the graph \(DP(p)[PO((p,0)), PO((p,1)), \ldots, PO((p,|p|))]\) is non-circular. In this case we say that the interfaces are compatible. In the literature one finds a slightly different class of attribute grammars, the so-called L-ordered attribute grammars.

**Definition 3.1 (L-Ordered Attribute Grammar)** An attribute grammar is a L-ordered attribute grammar if there exist total orders \(TO(X)\) for every non-terminal \(X\) such that for every production \(p \in P\) the graph \(DP(p)[TO((p,0)), TO((p,1)), \ldots, TO((p,|p|))]\) is cycle free.

The total orders \(TO(X)\) on \(A(X)\) are easily converted into interfaces: cut them into maximal segments of inherited and synthesized attributes. In [EF82], Engelfried and Filè proved that deciding whether an attribute grammar is partitionable or not is a NP-complete problem. Kastens’ [Kas80] defined a subclass of partitionable attribute grammars, the so-called ordered attribute grammars, that can be checked by an algorithm that depends polynomially in time on the size of the attribute grammar.

We shall continue by presenting Kastens’ scheduling algorithm. After that, we discuss the visit-sequence paradigm. Next, we discuss inter-traversal attribute dependencies. Finally, we shall present the binding-tree based attribute evaluators, i.e., purely functional and strict attribute evaluators.

### 3.6.2 Ordered Attribute Grammars

In this section we briefly present Kastens’ ordering algorithm [Kas80]. The basic idea of this algorithm is the following: for each symbol \(X \in V\) of a given \(AG\) a partial order \(DS(X)\) over the attributes \(A_{nont}(X)\) is computed. It determines an evaluation order for the attributes of \(X\), applicable in any context where \(X\) may occur. As a result, an element \(X.a \to X.b \in DS(X)\) indicates that the value of attribute instance \(a\) must be evaluated before \(b\) in any node that is an instance of \(X\) and for every tree assigned to the grammar \(AG\).

The existence of such an order is a sufficient but not necessary condition for the well-definedness of attribute grammars. Note that Kastens’ ordering algorithm makes a worst case assumption by merging all (indirect) dependencies on attributes of a non-terminal, in any context the non-terminal may occur, into a single dependency graph. This pessimistic approach, however, is crucial for L-ordered grammars: it must always be possible to compute the attributes of \(X\) in the order specified by \(DS(X)\), irrespective of the actual context of \(X\).

The next definition is taken from Kastens’ original paper [Kas80] except for some differences in notation. For a version more oriented towards implementation of this algorithm we refer the reader to [RT89, Pen94].

**Step 1:** \(DP = \bigcup_{p \in P} DP(p)\), is the relation of direct dependencies between attribute
occurrences induced by the attribution rules of productions. The AG is not ordered if \( DP \) is cyclic. This circularity is called \textit{type I circularity}. No attribute grammar in Bochmann normal form has type I circularity \cite{RT89}. Recall that in Figure 2.2 we have (graphically) shown the relation \( DP \) of the \textsc{block} AG.

\textbf{Step 2}: \( IDP = \bigcup_{p \in P} IDP(p) \), is the relation of induced dependencies between attribute occurrences. \( IDP \) projects indirect dependencies into dependencies between attribute occurrences as follows: every dependency between attributes of one occurrence of a grammar symbol, say \( X \), induces a dependency between corresponding attributes of all occurrences of \( X \). Formally it is defined as follows:

\[
IDP(p) = DP(p) \cup \{ \langle p, i, a \rangle \rightarrow \langle p, j, b \rangle | \{ \langle p, i, a \rangle, \langle p, i, b \rangle \} \subseteq O_{pr}(p) \\
\wedge \langle p', j, a \rangle \rightarrow \langle p', j, b \rangle \in IDP^+ \\
\wedge \langle p, i \rangle =_N \langle p', j \rangle \}
\]

The AG is not ordered if \( IDP \) is cyclic. This circularity is called \textit{type II circularity}. Figure 3.2 shows the \( DP \), and \( IDP \) graphs induced by the \textsc{block} AG.

The relation \( IDS = \bigcup_{X \in N} IDS(X) \) defines the \textit{Induced Dependencies} among attributes of \textit{Symbols}:

\[
IDS(X) = \{ X.a \rightarrow X.b | \langle p, i, a \rangle \rightarrow \langle p, i, b \rangle \in IDP \wedge \langle p, i \rangle =_N X \}
\]

The induced dependencies of the symbols for the \textsc{block} attribute grammar are:
Step 3: determines the “interfaces” for the non-terminal symbols. That is, it statically establishes the number of visits to a non-terminal symbol $X$ and for each of those visits it defines which inherited attributes of $X$ are used to compute which synthesized attributes of $X$. Several orders are possible. Kastens’ algorithm maximizes the size of the interfaces so that the number of visits is minimized. In order to compute such interfaces we define successively

$$
A_{X,1} = A_{syn}(X) - \{ X.a \mid X.a \mapsto X.b \in IDS^+ \}
$$

$$
A_{X,2n} = \{ X.a \mid X.a \in A_{inh}(X) \land \forall X.b : X.a \mapsto X.b \in IDS^+ \Rightarrow \exists m < 2n : X.b \in A_{X,m} \}
$$

$$
A_{X,2n+1} = \{ X.a \mid X.a \in A_{syn}(X) \land \forall X.b : X.a \mapsto X.b \in IDS^+ \Rightarrow \exists m < 2n + 1 : X.b \in A_{X,m} \}
$$

where the sets $A_{X,k}$, with $1 \leq k \leq m$ form a disjoint partition of $A_{nont}(X)$. The algorithm uses a “backward” sort, hence, the evaluation order corresponds to a decreasing order of index $k$. Thus, the subsets are in such a way that $A_{X,k}$ contains the attributes which contribute directly to the computation of attributes in $A_{X,k-1}$. The interfaces of non-terminal $X$ are defined as follows:

$$\text{Interface}(X) = [(A_{X,m}, A_{X,m-1}), \ldots, (A_{X,2}, A_{X,1})]$$

This is the crucial step of Kastens’ algorithm and it is this that makes the algorithm polynomial. Many partial orders comply with a $IDS$ relation, but step 3 fixes a particular choice: the one that maximizes the interfaces. This may be an unfortunate choice.

Having computed the disjoint partitions of $A_{nont}(X)$ for each $X \in V$, the graphs $DS(X)$ are defined as follows:

$$DS(X) = IDS(X) \cup \{ X.a \mapsto X.b \mid X.a \in A_{X,k} \land X.b \in A_{X,k-1} \land 2 \leq k \leq m \}$$

We are now ready to give the definition of ordered attribute grammar.

**Definition 3.2 (Ordered Attribute Grammar)** An attribute grammar is an Ordered Attribute Grammar (OAG) if for every production $p \in P$, the graph $DP(p)[DS(\langle p, 0 \rangle), DS(\langle p, 1 \rangle), \ldots, DS(\langle p, |p| \rangle)]$ is cycle free.
If the constructed graphs are circular, the grammar is rejected, although circularities also arise for some non-circular attribute grammars. On the contrary, if they are not circular, they can be topologically sorted in order to determine a total order on the attribute occurrences of a production. This order can be interpreted as a sequence of abstract computations to be performed on that production. We will return to this subject in the next section. A circularity can originate from two sources. Either the grammar is not L-ordered and no interface exist, or it is L-ordered, but step 3 selected a non-compatible interface. In this case, one could try to enforce a different disjoint partition of $A_{\text{non}(X)}$ by adding artificial dependencies. Such circularity is called type III circularity. Attribute grammars whose type III circularity can be eliminated are called arranged orderly attribute grammars [Kas80].

Let us now prove that the BLOCK AG is an ordered attribute grammar. First, we define the sets $A_{X,k}$ of disjoint partitions of attributes of all symbols $X$ of BLOCK.

\[
\begin{align*}
A_{P,1} &= \{ P.\text{errs} \} \\
A_{\text{Its},1} &= \{ \text{Its}.\text{errs} \} \\
A_{\text{Its},2} &= \{ \text{Its}.\text{env} \} \\
A_{\text{Its},3} &= \{ \text{Its}.\text{dclo} \} \\
A_{\text{Its},4} &= \{ \text{Its}.\text{dcli}, \text{Its}.\text{lev} \} \\
A_{\text{It},1} &= \{ \text{It}.\text{errs} \} \\
A_{\text{It},2} &= \{ \text{It}.\text{env} \} \\
A_{\text{It},3} &= \{ \text{It}.\text{dclo} \} \\
A_{\text{It},4} &= \{ \text{It}.\text{dcli}, \text{Its}.\text{lev} \}
\end{align*}
\]

Next, we compute the partial orders $DS(X)$ over the attributes of $A_{\text{non}(X)}$. In this case, no extra dependency is induced by the partitions $A_{X,k}$. As a result we have:

\[
\begin{align*}
DS(P) &= DS(P) \\
DS(\text{Its}) &= IDS(\text{Its}) \\
DS(\text{It}) &= IDS(\text{It})
\end{align*}
\]

Consequently, the dependency graphs shown in Figure 3.2 contain the dependencies of productions $DP$ pasted with $DS$ as stated in the definition of ordered attribute grammars. All the dependency graphs are cycle free. So, the BLOCK AG is ordered. We have the following partitions for the non-terminal symbols of the grammar:

\[
\begin{align*}
\text{Interface}(P) &= \{ \{\}, \{P.\text{errs}\} \} \\
\text{Interface}(\text{Its}) &= \{ \{\text{Its}.\text{dcli}, \text{Its}.\text{lev}\} , \{\text{Its}.\text{dclo}\} , \{\text{Its}.\text{env}\} , \{\text{Its}.\text{errs}\} \} \\
\text{Interface}(\text{It}) &= \{ \{\text{It}.\text{dcli}, \text{It}.\text{lev}\} , \{\text{It}.\text{dclo}\} , \{\text{It}.\text{env}\} , \{\text{It}.\text{errs}\} \}
\end{align*}
\]

It is worthwhile to note that the scheduling algorithm just broke up the circular definitions of the circular program ($Eval_{[60]}$) into two partitions (or traversals); the first specifies that given attributes $\text{lev}$ and $\text{dcli}$ of non-terminals $\text{Its}$ and $\text{It}$ it is possible to compute attribute $\text{dclo}$. The second specifies that given $\text{env}$ its is possible to compute $\text{errs}$. Note that this is exactly the scheduling we have defined in Section 3.1.

### 3.6.3 Chained Scheduling Algorithm

Chained Scheduling Algorithm [Pen94] is a variant of Kastens’ ordered scheduling algorithm. Chained scheduling uses the same first three steps to determine the interfaces
of the non-terminals. The two algorithms differ in the remaining steps. The difference is in the process to sort topologically the attribute occurrences of productions, according to the partitions computed in step 3. Since a partial order is being converted into a total one, several orders can be defined. As a result of changing the scheduling order, the lifetime of the attribute occurrences may be altered, which may have a large impact in the performance of the attribute evaluator. We will return to this point in the next section.

Chained scheduling was designed to improve the behaviour of functional attribute evaluators. It chooses such an evaluation order that every attribute occurrence of production $p$ is computed as early as possible. This scheduling order complies with the interfaces of the non-terminal symbols occurring in $p$. Kastens’ scheduling algorithm schedules the attribute occurrences of production $p$ according to the partitions computed in step 3. That is, an attribute occurrence, say $\alpha$, is scheduled to be computed in the partition (or traversal) where $\alpha$ is needed.

In the next section we present visit-sequences, which are the output of both algorithms. We shall briefly compare the algorithms when scheduling the evaluation order for the BLOCK AG. A detailed analysis of both scheduling algorithms can be found in ([Pen94],Chapter 5).

### 3.6.4 The Visit-Sequence Paradigm

Traditionally, the result of the attribute grammar scheduling algorithm is a sequence of abstract computations that have to be performed by a multiple traversal attribute evaluator. Such abstract computations are usually called visit-sequences. They are constructed according to the following idea: for every production $p \in P$ a fixed sequence of abstract computations is associated. They abstractly describe which computations have to be performed in every visit of the evaluator to a particular type of nodes in the tree. Such nodes are the instances of $p$.

Three kinds of abstract computations or instructions are used: eval $(x)$ that computes attribute $x$, visit $(n,v)$ that descends to node $n$ for the $v$th time and suspend $(v)$ that returns from visit $v$ to the father node. In a visit-sequence evaluator, the number of visits to a non-terminal $X$ is fixed: it corresponds to the number of elements in Interface$(X)$. We denote the number of visits of nonterminal $X$ by $v(X)$. Furthermore, each visit $i$ to a $X$, with $1 \leq i \leq v(X)$, has a fixed interface: the element $(A_{X,k}, A_{X,k-1})$ in position $i$ of sequence Interface$(X)$. This interface consists of a set of inherited attributes of $X$ that may be used during the visit $i$ and another set of synthesized attributes that are guaranteed to be computed by the visit $i$. We denote these two sets by $A_{inh,v}(X,i)$ and $A_{syn,v}(X,i)$, where $A_{inh,v}(X,i) = A_{X,k}$ and $A_{syn,v}(X,i) = A_{X,k-1}$.

The visit-sequence of a production is usually presented as a list of the three basic instructions. Visit-sequences, however, are the input of our techniques to derive purely functional attribute evaluators. Thus, they are divided into visit-sub-sequences $vss(p,i)$, containing the instructions to be performed on visit $i$ to the production $p$, with $p$ applied on $X$ and $1 \leq i \leq v(X)$. The suspend instructions are used as breakpoints. In order to simplify the presentation of our algorithms, visit-sub-sequences are also annotated with
define and usage attribute directives. Every visit-sub-sequence \( \text{vss}(p, i) \) is annotated with the interface of visit \( i \) to \( X \). Therefore \( \text{vss}(p, i) \) is annotated with \( \text{inh} (\alpha) \) and \( \text{syn} (\beta) \), where \( \alpha (\beta) \) is the list of the elements of \( A_{\text{inh},v}(X, i) \) and \( A_{\text{syn},v}(X, i) \). Every instruction \( \text{eval} (\alpha) \) is annotated with the directive \( \text{uses} (bs) \) that specifies the list of attribute occurrences used to evaluate \( \alpha \), i.e., the occurrences that \( \alpha \) depends on. The instruction \( \text{visit} (i, v) \) causes child \( i \) of production \( p \), say \( X_i \), to be visited for the \( v \)th time. The visit uses the attribute occurrences of \( A_{\text{inh},v}(X_i, v) \) as arguments and returns the attribute occurrences of \( A_{\text{syn},v}(X_i, v) \). Thus \( \text{visit} (i, v) \) is annotated with \( \text{inp} (is) \) and \( \text{out} (os) \) where \( is \) is the list of the elements of \( A_{\text{inh},v}(X_i, v) \) and \( os \) is the list of elements of \( A_{\text{syn},v}(X_i, v) \).

Visit-Sequences present the structured and annotated visit-sub-sequences derived for the BLOCK \( AG_1 \), obtained with the chained scheduling algorithm.

| plan R | begin 1 inh( ) eval (Iits.lev) uses( ) eval (Iits.dcli) visit (Iits.1) inp(Iits.lev, Iits.dcli) out(Iits.dclo) eval (Iits.env) uses(Iits.dclo) visit (Iits.2) inp(Iits.env) out(Iits.errs) eval (P.errs) uses(Iits.errs) end 1 syn(P.errs) | plan CONSITs | begin 1 inh(Iits1.lev, Iits1.dcli) eval (Iits.dcli) uses(Iits1.dcli) eval (Iits.lev) uses(Iits1.lev) visit (Iits.1) inp(Iits.dcli, Iits.lev) out(Iits.dclo) eval (Iits.1) uses(Iits.dclo) eval (Iits.2) uses(Iits.dcli) eval (Iits.1) uses(Iits1.lev) eval (Iits.2) uses(Iits1.lev) visit (Iits.2) inp(Iits2.dcli, Iits2.lev) out(Iits2.dclo) eval (Iits2.dcli) uses(Iits2.dclo) eval (Iits2.1) uses(Iits2.1) eval (Iits2.errs) uses(Iits2.errs) end 1 syn(Iits2.dclo) | plan BLOCK | begin 1 inh(It.lev, It.dcli) eval (It.lev) uses(It.lev) eval (It.1) uses(It.1) visit (It.1) inp(It.1.dcli, It.lev) out(It.1.dclo) eval (It.1.dcli) uses(It.1.dclo) eval (It.1) uses(It.1) eval (It.2) uses(It.2) visit (It.2) inp(It.2.errs) out(It.2.errs) eval (It.2.errs) uses(It.2.errs) end 2 syn(It.2.errs) |

Visit-Sequences 1: The annotated visit-sub-sequences induced by grammar \( AG_1 \).
The boxed attribute occurrences correspond to occurrences that are defined in one visit-sub-sequence and used in a different one. An implementation of this visit-sequences has to have a special mechanism to handle such occurrences: they induce values that have to be passed between different traversals of the evaluator.

The visit-sequences produced with the chained algorithm are similar to the ones produced by Kastens’ algorithm. The single exception is instruction `eval (It.errs)` of production `Decl`. The Chained algorithm schedules this instruction to the first sub-sequences, since all the attribute occurrences used to compute its value are available in the first visit to production `Decl` (see the `uses` annotation). On the contrary, Kastens’ algorithm schedules this instruction to the second sub-sequence of `Decl`, according to the partitions computed by step 3 of the algorithm. That is, the instruction is schedule to the visit where the attribute is needed. In this case, the attribute occurrences `It.dcli` and `It.lev`, and the grammar symbol `Name` (i.e., the arguments of the semantic function defining `It.errs`) must be retained for the second visit. This may result in a very inefficient evaluator, since at all nodes that are instances of `Decl` the accumulated list of declarations has to be retained for the second visit. We have seen this “problem” before: in Program 3.48 we have also scheduled (by-hand) the computation of that value for the second traversal of the evaluator. In that occasion, we were misled by the types of the functions.

### 3.6.5 Inter-Traversal Attribute Dependency

Attribute evaluators can be implemented as a set of procedures using standard attribute grammar techniques [Kat84, Kas91b]. Traditional implementations of such procedures have side-effects. Attributes, for example, may be stored in global variables or in the nodes of the original abstract syntax tree [Kas91b].

In a purely functional implementation of attribute grammars, where no side-effects are allowed, attributes can neither be stored in global variables nor in the abstract syntax tree nodes. Attributes are only the arguments and the results of the attribute evaluator’s functions. As a consequence, the main problem in strict and purely functional attribute evaluation is to handle attribute instances that are computed in one traversal, and are used in a following one. That is, how can an intermediate attribute value be used in different traversals of the evaluator? These dependencies, known as Inter-Traversal Attribute Dependencies (ITAD), are represented in Figure 3.3.

Inter-traversal attribute dependencies are defined next.

**Definition 3.3 (Inter-Traversal Attribute Dependency)** A node $N$ of an abstract syntax tree has an inter-traversal attribute dependency between traversal $v$ and $w$, with $v < w$, if there is an attribute instance which value is evaluated during traversal $v$ to $N$ and which is used during visit $w$ to $N$.

A straightforward imperative attribute evaluator handles those dependencies easily: attribute values needed later are stored in the tree nodes. A later traversal can directly
access those attributes. In this approach, the tree data types are extended with references to such attribute instances. Hence, these references remain present during the attribute evaluation task as long as the tree node exists. In [Kas91b], Kastens presents an algorithm which removes as many attributes from the tree as possible.

In a purely functional setting, an attribute value needed in a future traversal must be passed as one of the results of a preceding traversal. Thus, an explicit mechanism to pass attribute values between traversals must be defined. In the next section, we present the binding-tree paradigm where the inter-traversal attribute values are handled by a specific purpose data structure: a binding-tree.

### 3.6.6 The Binding-Tree Paradigm

The binding-tree paradigm is a strict and purely functional technique to implement multiple traversal attribute evaluators [Pen94]. This technique is based on the visit-sequence paradigm: the visit-sub-sequences induce a set of traversal functions which are “glued” by intermediate data structures. The traversal functions are called visit-functions and the intermediate data structures are the binding-trees.

The binding-trees are used with the single purpose of passing the inter-traversal attribute values between different traversals of the attribute evaluator. Binding-trees contain precisely the values that should be explicitly passed from one traversal to following ones. Such trees are constructed in one traversal – values that should be passed are included – and destructed in a following one, so that the values are available for use. The visit-functions and the binding-tree data types are automatically induced by a binding analysis of the visit-sub-sequences [Pen94]. Figure 3.4 shows a graphical representation of this approach.

Pennings developed this approach in the context of incremental evaluation of higher-order attribute grammars, where incrementality is achieved via the memoization of calls to (side-effect free) visit-functions. In order to obtain a better incremental behaviour, Pennings uses the so-called factorized binding-trees. To explain this technique, let us consider a three traversal evaluator which computes in the first traversal a value, that is used in the
second and third one. This value has to be passed in a binding-tree among the traversal functions, and to pass it again in another binding-tree from the second to the third traversal, accumulating also the values that might have to be passed from the second to the third. Aiming at better incrementality, Pennings uses a different approach. The first traversal returns two binding-trees: passing the same value from the first to the second and from the first to the third. Then, another binding-tree passes the values from the second to the third. That is, a (factorized) binding-tree is used to pass a value from the traversal that creates it to every traversal that uses it. In other words, the number of binding-trees is quadratic in the number of traversals.

As a result of the binding-tree mapping, a visit-function \( \text{visit}_v X \) is constructed for every terminal \( X \) and every visit \( v \) such that \( 1 \leq v \leq v(X) \). The arguments of the visit-function are the subtree labelled \( X \), the inherited attributes of visit \( v \) and the binding trees computed in previous visits that are destructed by visit \( v \). The results are the binding-trees for following visits and the synthesized attributes of visit \( v \). The first visit does not inherit any binding-tree, and the last one \( n \) does not synthesize them. The introduction of binding-trees is reflected in the types of the visit-functions.

\[
\begin{align*}
\text{visit}_1 X &:: X \rightarrow "A_{\text{inh}, v}(X, 1)" \rightarrow (X^{1-2}, \ldots, X^{1-n}, "A_{\text{syn}, v}(X, 1)") \\
\text{visit}_k X &:: X \rightarrow "A_{\text{inh}, v}(X, k)" \rightarrow X^{1-k} \rightarrow \cdots \rightarrow X^{k-1-k} \rightarrow (X^{k-k+1}, \ldots, X^{k-n}, "A_{\text{syn}, v}(X, k)") \\
\text{visit}_n X &:: X \rightarrow "A_{\text{inh}, v}(X, n)" \rightarrow X^{1-n} \rightarrow \cdots \rightarrow X^{n-1-n} \rightarrow "A_{\text{syn}, v}(X, n)"
\end{align*}
\]

where the quotes around the set of inherited and synthesized attributes of a particular visit should be interpreted as the types of the elements of those sets. Superscripts are used to denote the origins and the targets of binding-trees. Thus, \( X^{v \rightarrow w} \) denotes the constructor type of a binding-tree with origin in traversal \( v \) (\( 1 \leq v \leq v(X) \)) and target in traversal \( w \) (\( v < w \leq v(X) \)). The original syntax tree does not change during the different traversals of the evaluator. Attribute values are passed between traversals, as arguments and as results of visit-functions, or within binding-trees. No attribute is stored in the tree nodes.
Binding-trees may be empty, and most of them probably are. As a final step of the binding analysis, an empty test for bindings is performed: binding-trees that do not bind any values are eliminated. Binding-trees are terms with a structure much like the tree that is being traversed. Thus, binding-trees induce additional data types: one per binding-tree.

Let us consider the `block` example. According to Visit-Sequences [67], the binding-tree approach induces a single binding-tree type constructor. This type is defined as follows:

\[
\text{data } \text{Its}^{1\rightarrow2} = \begin{cases} \text{NilIts}^{1\rightarrow2} \\
\text{ConsIts}^{1\rightarrow2} \text{ It}^{1\rightarrow2} \text{ Its}^{1\rightarrow2} \\
\text{Use}^{1\rightarrow2} \\
\text{Decl}^{1\rightarrow2} \text{ (It.errs)} \\
\text{Block}^{1\rightarrow2} \text{ (Its.lev)} \end{cases}
\]

The types of the attributes `It.err` and `Its.lev`, which have an inter-traversal attribute dependency, are included in the appropriate data constructors of the binding-tree type. The visit-functions get an additional argument/result which is the binding tree. We have the following visit-function types:

\[
\begin{align*}
\text{visit}1\text{Its} &:: \text{ Its} \rightarrow \text{ Env} \rightarrow \text{ Int} \rightarrow (\text{Its}^{1\rightarrow2}, \text{Env}) \\
\text{visit}2\text{Its} &:: \text{ Its} \rightarrow \text{ Env} \rightarrow \text{ Its}^{1\rightarrow2} \rightarrow \text{Err} \\
\text{visit}1\text{P} &:: (R \text{ tIts}) = \text{ errs}_1 \\
\text{visit}2\text{Its} &:: (\text{NilIts}) \text{ dcli} \text{ lev} = (\text{NilIts}^{1\rightarrow2}, \text{dcli})
\end{align*}
\]

Next, we present the complete binding-tree based attribute evaluator.

\[
\text{data } \text{Its}^{1\rightarrow2} = \begin{cases} \text{NilIts}^{1\rightarrow2} \\
\text{ConsIts}^{1\rightarrow2} \text{ It}^{1\rightarrow2} \text{ Its}^{1\rightarrow2} \\
\text{Use}^{1\rightarrow2} \\
\text{Decl}^{1\rightarrow2} \text{ Err} \\
\text{Block}^{1\rightarrow2} \text{ Int} \end{cases}
\]

\[
\begin{align*}
\text{visit}1\text{Its} &:: (\text{ConsIts} \text{ tIt} \text{ tIts}_2) \text{ dcli} \text{ lev} = (\text{ConsIts}^{1\rightarrow2} \text{ tIt}^{1\rightarrow2} \text{ tIts}_2^{1\rightarrow2}, \text{dcli}_2) \\
&\quad \text{ where } (\text{tIts}_2^{1\rightarrow2}, \text{dcli}_2) = \text{visit}1\text{Its} \text{ tIt} \text{ dcli} \text{ lev} \\
&\quad \text{and } (\text{tIts}_2^{1\rightarrow2}, \text{dcli}_1) = \text{visit}1\text{Its} \text{ tIts} \text{ dcli}_1 \text{ lev}_1 \\
&\quad \text{errs}_1 = \text{visit}2\text{Its} \text{ tIts} \text{ dcli}_1 \text{ tIts}_2^{1\rightarrow2} \\
&\text{visit}2\text{Its} &:: (\text{ConsIts} \text{ tIt} \text{ tIts}_2) \text{ env} (\text{ConsIts}^{1\rightarrow2} \text{ tIt}^{1\rightarrow2} \text{ tIts}_2^{1\rightarrow2}) = \text{errs} \\
&\quad \text{ where } \text{errs} = [] \\
&\quad \text{errs}_1 = \text{visit}2\text{Its} \text{ tIt} \text{ env} \text{ tIts}_2^{1\rightarrow2} \\
&\quad \text{errs}_2 = \text{visit}2\text{Its} \text{ tIts}_2 \text{ env} \text{ tIts}_2^{1\rightarrow2} \\
&\text{visit}1\text{P} &:: (R \text{ tIts}) = \text{ errs}_1 \\
&\quad \text{ where } \text{lev}_1 = 0 \\
&\text{visit}2\text{Its} &:: (\text{NilIts}) \text{ dcli} \text{ lev} = (\text{NilIts}^{1\rightarrow2}, \text{dcli}) \\
&\quad \text{ where } \text{errs} = [] \\
\end{align*}
\]

\[
\begin{align*}
\text{visit}1\text{It} &:: (\text{Decl} \text{ tName}) \text{ dcli} \text{ lev} = (\text{Decl}^{1\rightarrow2} \text{ dcli}, \text{dcli}) \\
&\quad \text{ where } \text{dcli} = (\text{Pair} \text{ tName} \text{ lev}) : \text{dcli} \\
&\quad \text{errs} = (\text{Pair} \text{ tName} \text{ lev}) \text{ 'mNBIn' dcli} \\
\text{visit}1\text{It} &:: (\text{Use} \text{ tName}) \text{ dcli} \text{ lev} = (\text{Use}^{1\rightarrow2}, \text{dcli}) \\
&\quad \text{ where } \text{errs} = (\text{Pair} \text{ tName} \text{ lev}) \text{ 'mBln' env} \\
\text{visit}2\text{It} &:: (\text{Block} \text{ tIts}) \text{ dcli} \text{ lev} = (\text{Block}^{1\rightarrow2} \text{ dcli}, \text{dcli}) \\
&\quad \text{ where } \text{errs}_1 = \text{visit}2\text{Its} \text{ tIts} \text{ env} \text{ lev}_1 \\
&\quad \text{errs}_2 = \text{visit}2\text{Its} \text{ tIts} \text{ env} \text{ lev}_1 \\
\text{Eval 2:} \text{ The block binding-tree based attribute evaluator for grammar } AG_1.
\]
The binding-tree data constructors contain the derived types for attributes \( \text{It.err} \) and \( \text{Its.lev} \). In Chapter 6 we will discuss the incremental evaluation of purely functional attribute evaluators and this evaluator will be analysed in great detail. An abstract syntax tree and a binding tree constructed during the decoration of a simple sentence will be presented in Figure 6.3.

Let us return to the \textit{repmin} problem and briefly discuss its attribute grammar (AG \[13\]). As we have explained before, the dependency from a synthesized to an inherited attribute (i.e., \( \text{Tree.min} \rightarrow \text{Tree.m} \)) induces a second traversal to term \( \text{Tree} \). Note, however, that no inter-traversal attribute dependency occurs, because no value has to be passed from the first to the second traversal of \textit{repmin}. Thus, the induced binding-tree \( \text{Tree}^{1 \rightarrow 2} \) is an empty binding-tree, which is eliminated by the binding analysis. As a result, the functional solution presented in Program 3.44 is the functional attribute evaluator obtained with the binding-tree approach.
Chapter 4

Deforested Attribute Evaluators

Summary

This chapter discusses deforested attribute evaluators. Visit-tree based attribute evaluators, λ-attribute evaluators and deforested lazy evaluators are defined. The deforestation of higher-order attribute grammars and attribute grammar based systems are also presented.

In a purely functional setting, programs are often constructed as a set of simple and clear components, which are “glued” together by using intermediate data structures. Compilers are a typical example of programs designed in this way: a function (i.e., the parser) constructs a syntax tree that later is consumed by another function (i.e., the attribute evaluator).

Attribute grammars are also designed and implemented as a set of simple components, with the abstract syntax tree as the intermediate structure. The parser constructs the syntax tree and the attribute evaluator decorates that tree. Traditionally, the attribute evaluator walks over the tree in order to decorate it with attribute values. Several walks, also called traversals, are typically performed to decorate a syntax tree. The syntax trees are, once again, the data structures that glue together the different traversals of the evaluator. They not only guide the traversals of the evaluator, but they also convey information between them. That is, they store values of attributes that are needed on different traversals of the evaluator. Consequently, in a strict and purely functional setting, additional intermediate data structures may have to be used in order to pass such values between different traversal functions. This means that the introduction, and consequent construction and destruction, of a possibly large set of additional data structures may degrade execution time dreadfully.

This chapter presents a new technique to implement attribute grammars as a set of strict, side-effect free functions. The attribute evaluators are completely deforested, i.e.,
no explicit intermediate data structure has to be defined, constructed, nor traversed. Furthermore, a parser may directly call the attribute evaluator functions and no syntax tree has to be constructed. Moreover, all the attribute instances are handled in a canonical way: they just show up as arguments and results of visit-functions.

4.1 Functional Multiple Traversal Attribute Evaluators

This section presents new techniques for deriving efficient strict multiple traversal evaluators. Our techniques are based on an attribute lifetime analysis of the attribute grammar. Such an analysis is described in the next section. After that, we present two new techniques to implement attribute grammars in a strict, functional setting: the visit-tree based attribute evaluators (Section 4.1.2) and the deforested attribute evaluators (Section 4.1.3).

4.1.1 Attribute Grammar Lifetime Analysis

Our techniques to derive purely functional attribute evaluators from an (higher-order) attribute grammar are based on a static analysis of the visit-sequences produced by the AG scheduling algorithms. This static analysis performs a lifetime analysis of attribute occurrences and grammar symbols. We need to distinguish two cases: the values that are used within eval instructions and the non-terminal symbols that are visited. Note that in the higher-order formalism, grammar symbols (terminal and non-terminal symbols) may be used in the attribute equations as normal values. Consequently, they can also occur as normal values used by eval instructions.

We start by analysing the lifetime of attribute occurrences and grammar symbols within the eval instructions. It is known, from the visit-sub-sequences, in which visit each attribute occurrence is defined and in which visit(s) it is used. Thus, we introduce two predicates def and use. The predicate def\((p, a, v)\) denotes whether attribute \(a\) of production \(p\) is defined in visit \(v\). Likewise, use\((p, a, v)\) denotes whether attribute occurrence \(a\) of production \(p\) is used in visit \(v\):

\[
\begin{align*}
def(p, a, v) & = \text{eval}(a) \in vss(p, v) \lor \text{inh}(\ldots, a, \ldots) \in vss(p, v) \\
& \lor \text{out}(\ldots, a, \ldots) \in vss(p, v) \\
\text{use}(p, a, v) & = \text{uses}(\ldots, a, \ldots) \in vss(p, v) \lor \text{syn}(\ldots, a, \ldots) \in vss(p, v) \\
& \lor \text{inp}(\ldots, a, \ldots) \in vss(p, v)
\end{align*}
\]

Next, we extend these predicates to work on grammar symbols. The grammar symbols are not defined by attribute equations, but at parse-time. So, we assign visit number 0 to the parser. The predicate def is extended as follows:

\[
def(p, a, 0) = a \in V
\]

and the predicate use is overloaded to work on grammar symbols, too. Thus, use\((p, \langle p, k \rangle, v)\) denotes also whether the grammar symbol \(\langle p, k \rangle\) is used in visit \(v\) to \(p\), or not.
We can now define the predicate \textit{alive} on attribute occurrences and grammar symbols. An attribute or grammar symbol of a production \( p \) is \textit{alive} at visit \( i \), when it is defined in a previous visit and it is used in visit \( i \) or later. For each production \( p \) and for each of its visits \( i \), with \( 1 \leq i \leq v(\langle p, 0 \rangle) \), we define the set \( \text{alive}(p, i) \) which contains the \textit{live} attribute occurrences and grammar symbols on visit \( i \). It is defined as follows:

\[
\text{alive}(p, i) = \{ a \mid \text{def}(p, a, k) \land \text{use}(p, a, j) \land k < i \leq j \} \\
\cup \{ \langle p, k \rangle \mid \text{use}(p, \langle p, k \rangle, j) \land 0 \leq i \leq j \}
\]

The second step of our analysis is to perform a lifetime analysis on the visits to non-terminal symbols of the grammar. Let \( \text{alive.visits}(p, c, v) \) denote the list of visits to child \( c \) of production \( p \), which have to be performed in visit-sub-sequence \( v \) to \( p \) or later. This list is defined as follows:

\[
\text{alive.visits}(p, c, v) = [\text{visit}(c, i) \mid \text{visit}(c, i) \in \text{vss}(p, j), v \leq j \leq v(\langle p, 0 \rangle)]
\]

For each visit-sub-sequence, say \( s \), of production \( p \) we determine the earliest visit to each of the children of \( p \). The earliest visit \( v \) to a child of \( p \) in sub-sequence \( s \) is the earliest visit occurring in sub-sequence \( s \) or the earliest visit occurring in the remaining sub-sequences of \( p \). Thus, we define the function \( \text{inspect}(p, v) \) which takes the head of the list defined by \( \text{alive.visits} \), for all non-terminal symbols of production \( p \).

\[
\text{inspect}(p, v) = \{ \text{head alive.visits}(p, c, v) \mid \text{alive.visits}(p, c, v) \neq [] \land \langle p, c \rangle \in \text{rhs}(p) \}
\]

Before we proceed to present our functional evaluators, let us analyse first the Visit-Sequences \[\text{(\ref{eq:visits-1})}\] derived for the block grammar \( AG_1 \). We focus in the productions where inter-traversal attribute dependencies occur, \textit{i.e.}, productions \textbf{Block} and \textbf{Decl}. In both productions one attribute occurrence (\textit{i.e.}, \textit{Its.lev} and \textit{It.errs}) is defined in the first sub-sequence and used in the second one. Thus, we have the following two \textit{alive} sets:

\[
\begin{align*}
\text{alive(BLOCK, 1)} &= \{ \} \\
\text{alive(BLOCK, 2)} &= \{ \langle \text{Block, 1, lev} \rangle \} \\
\text{alive(DECL, 1)} &= \{ \langle \text{Name, 1} \rangle \} \\
\text{alive(DECL, 2)} &= \{ \langle \text{Decl, 0, errs} \rangle \}
\end{align*}
\]

In production \textbf{DECL} the pseudo-terminal symbol \textit{Name} is used in the first sub-sequence, and, by definition, was defined in visit 0. So, it is alive in visit 1. In production \textbf{Use} this symbol is used for the last time in sub-sequence 2. So, it is alive in both visits.

\[
\begin{align*}
\text{alive(USE, 1)} &= \{ \langle \text{Name, 1} \rangle \} \\
\text{alive(USE, 2)} &= \{ \langle \text{Name, 1} \rangle \}
\end{align*}
\]

Let us now analyse the lifetime of the non-terminal symbols. In the visit-sequences of productions \textbf{DECL}, \textbf{USE} and \textbf{NILITS} no \textit{visit} instruction occurs. As a result, the set \textit{inspect} for the both visits is an empty set. The single child of production \textbf{BLOCK}, \textit{i.e.}, non-terminal \textit{Its}, is visited twice, both visits in the second visit to \textbf{BLOCK}. So, we have:
and for the remaining productions we have:

\[
\begin{align*}
\text{alive visits}(\text{R}, 1) &= [\text{visit } (\text{Its}, 1), \text{visit } (\text{Its}, 2)] \\
\text{alive visits}(\text{ConsIts}, 1) &= [\text{visit } (\text{Its}, 1), \text{visit } (\text{Its}, 2)] \\
\text{alive visits}(\text{ConsIts}, 2) &= [\text{visit } (\text{Its}, 2)] \\
\text{inspect}(\text{R}, 1) &= \{\text{visit } (\text{Its}, 1)\} \\
\text{inspect}(\text{ConsIts}, 1) &= \{\text{visit } (\text{Its}, 1)\} \\
\text{inspect}(\text{ConsIts}, 2) &= \{\text{visit } (\text{Its}, 2)\}
\end{align*}
\]

4.1.2 Visit-Tree based Attribute Evaluators

This section presents a new approach to pure and strict functional implementation of attribute grammars [SKS97a]. This new approach is called visit-tree based attribute evaluators and it uses additional data structures, called visit-trees, to explicitly pass ITAD values between traversals. Basically, the visit-tree based attribute evaluator mimics the imperative approach: attribute values defined in one traversal and used in the following ones are stored in a new tree, the so-called visit-tree. Such values have to be preserved in the (visit-)tree from the traversal that creates them until the last traversal that uses them. Each traversal builds a new visit-tree for the next traversal, with the additional ITAD values stored in its nodes. The functions that perform the subsequent traversals find the values they need, either in their arguments or in the nodes of the (visit-)tree, exactly as in the imperative approach.

A set of visit-tree types is defined, one per traversal. A visit-tree for one traversal, say \( v \), is specialized for that particular traversal of the evaluator: it contains only the attribute values which are really needed in traversal \( v \) and the following ones. The visit-trees are constructed dynamically, \( i.e. \), during attribute evaluation. Consequently, the underlying data structure which guides the attribute evaluator is not fixed during evaluation. This dynamic construction/destruction of the visit-trees allows for an important optimization: subtrees that are not needed in future traversals are discarded from the visit-trees concerned. As result, any data no longer needed, no longer is referenced. Figure 4.1 presents a graphical representation of the visit-tree approach.

In order to present visit-tree based evaluators, we start by informally analysing a very simple production. Consider the following simplified visit-sub-sequences for production \( X = \text{Prod } Y Z \) (the annotations \( \text{inp} \) and \( \text{out} \) of the visit instructions are omitted since they are not relevant for this example):
4.1. Functional Multiple Traversal Attribute Evaluators

Figure 4.1: Visit-Tree based Attribute Evaluator: each traversal function \( t_i \) constructs a visit-tree \( T_{i+1} \) which stores ITAD values needed in traversals \( j \), with \( i \leq j \leq n \) (black rectangles). Subtrees no longer needed are discarded (this is represented in the right bottom subtree). The last traversal \( n \) does not construct any visit-tree.

\[
\text{begin 1} \quad \text{inh}(X.\text{inh}_1) \\
\text{visit} \quad (Y, 1) \\
\text{eval} \quad \ldots, \\
\quad \text{uses}(X.\text{inh}_1, \ldots), \\
\text{visit} \quad (Y, 2) \\
\text{eval} \quad (X.\text{syn}_1) \\
\quad \text{uses}(\ldots), \\
\text{end 1} \quad \text{syn}(X.\text{syn}_1)
\]

Observe that the inherited attribute \( X.\text{inh}_1 \) must be explicitly passed from the first visit to \( X \) (where it is defined) to the second one (where it is used). The non-terminal \( Y \) is visited twice, with both visits occurring in the first visit to \( X \).

In the visit-tree approach, the value of attribute \( X.\text{inh}_1 \) is passed to the second traversal in one visit-tree. This visit-tree is one result of the first traversal of the evaluator. So, we have to define two tree data types: one for the first traversal, and another for the second one. These data types, called *visit-tree data types*, are defined next. We use superscripts to distinguish the traversal they are intended to.

\[
\text{data } X^1 = \text{Prod}^1 \ Y^1 \ Z^1 \\
\text{data } X^2 = \text{Prod}^2 \ (\tau \ X.\text{inh}_1) \ Z^1
\]

In the first visit to \( X \), the non-terminal \( Y \) is visited twice. The visit-tree data type \( X^1 \) includes a reference to the first visit (the visit-tree type \( Y^1 \)) only. The second visit-tree is constructed when visiting the first one (see the visit-functions below). Thus, a reference to the earliest visit suffices. Non-terminal \( Y \) is not used in the second visit to \( \text{Prod} \). So, no reference to \( Y \) has to be included in \( Y^2 \). Consequently, subtrees labelled \( Y \) are discarded from the second traversal of the evaluator.

The visit-trees are arguments and results of the visit-functions. Thus, the visit-functions must explicitly destruct/construct the visit-trees. Next, we present the functions induced
by these visit-sub-sequences.

\[
\text{visit}_1X \ (\text{Prod}^1 \ tY^1 \ tZ^1) \ \text{inh}_1 = (\text{Prod}^2 \ \text{inh}_1 \ tZ^1, \text{syn}_1)
\]

where
\[
\begin{align*}
(\ldots) &= \text{visit}_1Y \ tY^1 \ldots \\
\text{syn}_1 &= \ldots
\end{align*}
\]

\[
\text{visit}_2X \ (\text{Prod}^2 \ \text{inh}_2 \ tZ^1) \ \text{inh}_2 = (\text{syn}_2)
\]

where
\[
\begin{align*}
(\ldots) &= \text{visit}_1Z \ tZ^1 \ldots \\
\text{syn}_2 &= f \ \text{inh}_1 \ldots
\end{align*}
\]

In the first traversal the attribute \(\text{inh}_1\) is computed, and a visit-tree node \(\text{Prod}^2\) (which stores this attribute value) is constructed. In the second traversal, that node is destructed and the value of \(\text{inh}_1\) is ready to be used. Observe also that the visit-function \(\text{visit}_1Y\) returns the visit-tree for the next traversal (performed by \(\text{visit}_2Y\)).

**Deriving Visit-Tree based Attribute Evaluators**

This section presents the formal derivation of visit-tree based attribute evaluators. To simplify the presentation we start by considering first-order attribute grammars only. In Section 4.3 we will discuss higher-order attribute grammars.

The visit-sub-sequences are mapped into visit-tree based visit-functions as follows: for each visit \(v\) of a non-terminal symbol \(X\), with \(1 \leq v \leq v(X)\), a visit-tree data type \(X^v\) is defined as follows:

\[
data X^v = "\text{constructors}(X, v)"
\]

where \(\text{constructors}(X, v)\) is the set of data constructors of the visit-tree data type \(X^v\). The quotes around this set should be interpreted as the sum of its elements. This set of data constructors is derived from visit-sub-sequences of the productions applied on \(X\) as follows: for each production \(P\), and for each visit \(v\) to \(X\), a constructor \(P^v\) is derived. The set \(\text{constructors}(X, v)\) is the union of those individual constructors.

\[
\text{constructors}(X, v) = \bigcup_{\text{P applied on } X} \{ P^v \ \text{T} \ a_1 \cdots (\text{T} \ a_m) \ (\text{T} \ vt_1) \cdots (\text{T} \ vt_n) \}
\]

for all \(a_i \in \text{alive}(P, v)\) and for all \(vt_i \in \text{inspect}(P, v)\). The function \(\text{T}\) is overloaded to work on visit instructions, too. The type induced by a visit instruction is defined as follows: \(\text{T visit}(\langle P, k \rangle, w) = X^w\).

Let us define next the formal derivation of the visit-functions. For each traversal \(v\) of a non-terminal symbol \(X\), a visit-function \(\text{visit}_vX\) is generated. Its arguments are a visit-tree of type \(X^v\), and an additional argument for each attribute in \(A_{\text{inh}_v}(X, v)\).
result is a tuple, whose first element is the visit-tree for the next traversal and the other elements are the attributes in $A_{\text{syn},v}(X,v)$. The visit-functions have a signature of the following form:

$$
\text{visit}_i X :: X \rightarrow (T \text{ inh}_1) \rightarrow \cdots \rightarrow (T \text{ inh}_k) \rightarrow (X^{v+1}, T \text{ syn}_1, \ldots, T \text{ syn}_l)
$$

with $A_{\text{inh},v}(X,v) = \{\text{inh}_1, \ldots, \text{inh}_k\}$ and $A_{\text{syn},v}(X,v) = \{\text{syn}_1, \ldots, \text{syn}_l\}$. The visit-function that performs the last traversal has a different signature since it does not construct any visit-tree. So, we have:

$$
\text{visit}_n X :: X^n \rightarrow (T \text{ inh}_1) \rightarrow \cdots \rightarrow (T \text{ inh}_k) \rightarrow (T \text{ syn}_1, \ldots, T \text{ syn}_l)
$$

Every visit-function $\text{visit}_i X$ is defined by a set of alternatives, one per production $P : X \rightarrow X_1 \ldots X_s$. The alternatives are selected by pattern matching on the visit-tree constructor $P^i$. Each visit-function alternative is of the following form:

$$
\text{visit}_i X \quad (P^i < \text{pat-vars}(P,i)>) \quad <\text{inherited}(i)> =
(P^{i+1} < \text{pat-vars}(P,i+1)>, <\text{synthesized}(i)>)
$$

where $\text{body}(i)$

The function which performs the last traversal looks as follows:

$$
\text{visit}_n X \quad (P^n < \text{pat-vars}(P,n)>) \quad <\text{inherited}(i)> = (<\text{synthesized}(i)>)
$$

where $\text{body}(n)$

The code fragments defining the inherited and synthesized attributes look as follows:

$$
<\text{inherited}(i)> = \text{inh}_1 \text{ inh}_2 \ldots \text{inh}_k
$$

$$
<\text{synthesized}(i)> = \text{syn}_1, \text{syn}_2, \ldots, \text{syn}_l
$$

for all $\text{inh}_i \in A_{\text{inh},v}(X,j)$ and for all $\text{syn}_i \in A_{\text{syn},v}(X,j)$. The fragment $<\text{pat-vars}(P,j)>$ denotes the pattern variables. It is defined as follows:

$$
<\text{pat-vars}(P,j)> = (\text{var } a_1) \ldots (\text{var } a_m) \ tX^v_k \ldots tX^w_r
$$

for all $a_i \in \text{alive}(P,j)$ and with $\{\text{visit } (\langle P,k,v \rangle, \ldots, \text{visit } (\langle P,r,w \rangle, w)\} = \text{inspect}(P,j)$. The function $\text{var}$ associates a pattern variable with every attribute occurrence or grammar symbol in $\text{alive}$. For every attribute occurrence $\langle P,k,a \rangle \in \text{alive}(P,j)$ and $k > 0$, we have $\text{var } \langle P,k,a \rangle = a_k$. That is, attribute occurrences are distinguished by labelling the respective attributes with the position of their non-terminal occurrences in the production. The exceptions are the attribute occurrences of the left-hand side of $P$. They are not labelled and we have $\text{var } \langle P,0,a \rangle = a$. For every grammar symbol occurrence $\langle P,k \rangle \in \text{alive}(P,j)$ we have $\text{var } \langle P,k \rangle = tX^v_k$. That is, the syntactically referenced symbols and visit-trees are prefixed with character “t” (denoting that those values are terms).
The body body\((j)\) of each alternative function definition of visit\(_j\)\(X\) is generated according to the instructions of the visit-sub-sequence \(vss(P, j)\). Every attribute equation of the form
\[
\text{eval } ((P, k, a)) \\
\text{uses}(\text{args}) \in vss(P, j)
\]
defining an attribute occurrence \(\langle P, k, a \rangle = f \ \text{args}\) of production \(P\), induces an equation
\[
a_k = f \ \text{args}
\]

Attribute occurrence \(\langle P, r, a \rangle\), with \(r > 0\) occurring in \(\text{args}\), is replaced by \(a_r\) and \(\langle P, 0, a \rangle\) is replaced by \(a\). Syntactic reference \(\langle X, r \rangle\) occurring in \(\text{args}\) is replaced by \(tX_r\). Local attribute occurrences are copied literally to the body of the respective visit-function.

Every visit instruction visit\((\langle P, c \rangle, w)\) defining the visit \(w\) to child \(c\) of \(P\), i.e., to the instance of the nonterminal \(X_c\), induces a recursive visit-function call. Two cases have to be distinguished:

If \(w < v(X_c)\) then, the visit-function call returns the visit-tree for the next traversal. The following equation is generated:
\[
(tX_c^{w+1}, \text{syn}_{1}, \ldots, \text{syn}_{k}) = \text{visit}_w X_c tX_c^w \ \text{inh}_{1} \ldots \text{inh}_l
\]

If \(w = v(X_c)\) then, only the synthesized attributes of \(vss(P, w)\) are computed by the visit-function call.
\[
(\text{syn}_{1}, \ldots, \text{syn}_{k}) = \text{visit}_w X_c tX_c^w \ \text{inh}_{1} \ldots \text{inh}_l
\]

with \(A_{\text{inh}_w}(X_c, w) = \{\text{inh}_{1}, \ldots, \text{inh}_{k}\}\) and \(A_{\text{syn}_w}(X_c, w) = \{\text{syn}_{1}, \ldots, \text{syn}_{l}\}\).

Having defined the mapping from visit-sub-sequences into visit-tree based visit-functions, we can now derive visit-tree based attribute evaluators. Next, we consider the evaluator for the block grammar \(AG_1\). We start by deriving the visit-tree data types. Non-terminal \(P\) has a single visit and non-terminals \(Its\) and \(It\) have two visits each. Thus, five data types are constructed: \(P^1\), \(Its^1\), \(Its^2\), \(It^1\) and \(It^2\). Let us define the constructors that have to handle the inter-traversal dependencies:

\[
\text{constructors}(\text{It}, 1) = \{\text{DECL}^1 \ Name, \ \text{USE}^1 \ Name, \ \text{BLOCK}^1 \ Its^1\}
\]
\[
\text{constructors}(\text{It}, 2) = \{\text{DECL}^2 (T \ errs), \ \text{USE}^2 \ Name, \ \text{BLOCK}^2 (T \ lev_1) \ Its^2\}
\]

the remaining sets of constructors are easily obtained. The two induced visit-tree data types are presented next. To simplify our notation and make evaluators easier to read, we omit the traversal number in the case of the first traversal.
The visit-trees data types $\text{Its}^2$ and $\text{It}^2$ are specialized for the second traversal of the strict evaluator: they contain the types of the attributes that are really needed in the second traversal. No reference to pseudo-terminal $\text{Name}$ is included in the data constructor $\text{DECL}^2$: it does not play any role in the second traversal of the evaluator.

We can now define the visit-functions, one for each traversal of a non-terminal symbol. Let us derive the visit-function alternative for the most intricate production: the production $\text{BLOCK}$. First, we define the types of the two visit-functions induced by the two traversals of non-terminal $\text{It}$.

\[
\text{visit}_1\text{It} :: \text{It} \to \text{Env} \to \text{Int} \to (\text{It}^2, \text{Env}) \\
\text{visit}_2\text{It} :: \text{It}^2 \to \text{Env} \to \text{Err}
\]

and the visit-functions are constructed as follows:

\[
\text{visit}_1\text{It} \quad (\text{BLOCK} < \text{pat-vars(\text{BLOCK}, 1)}>) \quad \text{deli lev} = \\
\quad (\text{BLOCK}^2 < \text{pat-vars(\text{BLOCK}, 2)}>, \text{dclo}) \\
\quad \text{where body}(1)
\]

\[
\text{visit}_n\text{It} \quad (\text{BLOCK}^2 < \text{pat-vars(\text{BLOCK}, 2)}>) \quad \text{env} = (\text{err}) \\
\quad \text{where body}(2)
\]

where the fragments $\text{pat-vars}$ are:

\[
< \text{pat-vars(\text{BLOCK}, 1)}> = t\text{Its}^1 \\
< \text{pat-vars(\text{BLOCK}, 2)}> = \text{lev} \text{tIts}^1
\]

The two bodies of visit-functions are derived from the corresponding visit-sub-sequences:

\[
< \text{body}(1) > = \text{lev} = \text{lev} + 1 \\
\text{dclo} = \text{deli} \\
< \text{body}(2) > = \text{deli}_1 = \text{env} \\
\text{(tIts}^2, \text{dclo}_1) = \text{visit}_1\text{Its}\text{tIts}^1\text{deli}_1 \\
\text{errs}_1 = \text{visit}_2\text{Its}\text{tIts}^2\text{dclo}_1 \\
\text{errs} = \text{errs}_1
\]

Next, we present the complete visit-tree based visit-functions constructed for the BLOCK AG. Copy equations induced by copy rules were trivially removed from the attribute evaluator.
Eval 3: The block visit-tree based attribute evaluator for grammar $AG_1$.

It is worthwhile to note how the AG scheduling algorithm statically breaks up pseudo-circular definitions into multiple traversal functions. For example, the circular definition occurring in function $evalIts$ of Program 1

$$(dclo_1, errs_1) = evalIts tIts deli_1 lev_1 delo_1$$

is now unravelled, and it can be strictly evaluated by the two visit-function calls occurring in the body of $visit_1P$, as follows:

$$errs_1 = uncurry visit_2Its (visit_1Its tIts^1 deli_1 lev_1)$$

The visit-tree based attribute evaluators have the following properties:

- The number of visit-trees is linear in the number of traversals of the evaluator.
- No additional arguments/results in the visit-functions are used to explicitly pass attribute values induced by ITADs between traversals.
- The visit-functions are strict in all their arguments, as a result of the order computed by the AG ordered scheduling algorithm.
- Efficient memory usage: data not needed is no longer referenced. References to grammar symbols and attribute instances can efficiently be discarded as soon as they have played their semantic role.
4.1. Functional Multiple Traversal Attribute Evaluators

- Standard function memoization techniques can be used to achieve incremental evaluation.
- The attribute evaluators are easy to read and understand.

4.1.3 Deforested Strict Attribute Evaluators

The visit-tree based attribute evaluator approach requires the definition of a possible large set of “glue” data types. One visit-tree data type has to be defined per traversal of the evaluator. Furthermore, the visit-trees must be constructed, traversed and destructed explicitly during attribute evaluation. As a result, the efficiency of the attribute evaluator can be affected by the construction and traversal of such trees.

This section presents a new technique for attribute evaluation: deforested attribute evaluators. Such evaluators, called \( \lambda \)-attribute evaluators, do not explicitly define, construct nor traverse any intermediate data structure. The \( \lambda \)-attribute evaluators consist of a set of partially parameterized visit-functions, each performing the computations of one traversal of the evaluator. Every visit-function returns, as one result, a (partial parameterized) visit-function that performs the next traversal of the evaluator. Performing this traversal corresponds to totally parameterize the visit-function and, once again, returning the function for the next traversal. The main idea is that for each visit-sub-sequence we construct a function that, besides mapping inherited to synthesized attributes, also returns the function that represents the next visit. Any state information needed in future visits is passed on by partially parameterising a more general function. The single exception is the final visit-function which returns synthesized attributes only. This technique is represented in Figure 4.2.

\[
\begin{align*}
\lambda_1 & \rightarrow \lambda_2 \times \text{syn}_1 \\
& \rightarrow \cdots \rightarrow \lambda_n \times \text{syn}_{n-1} \\
& \rightarrow \text{syn}_n
\end{align*}
\]

Figure 4.2: \( \lambda \)-Attribute Evaluator: Each traversal function builds the computations for the next traversal.

In order to introduce deforested evaluators, we start by analysing a simple production, like we did for the visit-tree evaluators. Thus, we consider again the production \( X = \text{PROD} Y Z \) and the visit-sub-sequences presented previously (see page 77). Next, we present the two deforested visit-functions derived from the two sub-sequences. Note that, they are themselves the evaluator. So, no tree data types have to be defined.
Chapter 4. Deforested Attribute Evaluators

\begin{align*}
\lambda_{Prod^1} & \quad \lambda_{Y_1} \lambda_{Z_1} \in\text{inh}_1 = (\lambda_{Prod^2} \in\text{inh}_1 \lambda_{Z_1}, \text{syn}_1) \\
\text{where} & \quad (\lambda_{Y_2}, \ldots) = \lambda_{Y_1}, \\
& \quad (\ldots) = \lambda_{Y_2} \\
& \quad \text{syn}_1 = \ldots
\end{align*}

\begin{align*}
\lambda_{Prod^2} & \quad \text{inh}_1 \lambda_{Z_1} \in\text{inh}_2 = (\text{syn}_2) \\
\text{where} & \quad (\ldots) = \lambda_{Z_1}, \\
& \quad \text{syn}_2 = f \in\text{inh}_1 \\
& \quad \lambda_{Y_2} \quad \text{partial parameterized in the} \\
& \quad \text{first traversal and totally} \\
& \quad \text{parameterized in the second one.}
\end{align*}

The visit-functions \( \lambda_{Y_1} \) and \( \lambda_{Z_1} \) define the computations of the first traversal of non-terminal symbols \( Y \) and \( Z \). The attribute occurrence \( X.x \) is passed from the first to the second traversal as a hidden result of \( \lambda_{Prod^1} \) in the form of an extra argument to \( \lambda_{Prod^2} \). Note that no reference to visits for non-terminal symbol \( Y \) are included in \( \lambda_{Prod^2} \) since all the visits to \( Y \) occur in the first visit to \( P \). Observe also that the function \( \lambda_{Z_1} \) is directly passed to the second visit to \( X \), where the first visit to \( Z \) is performed.

**Deriving \( \lambda \)-Attribute Evaluators**

We describe now the formal derivation of the \( \lambda \)-attribute evaluators for first-order attribute grammars. In Section 4.3, we will discuss the higher-order variant. For each production \( P : X \rightarrow X_1 \ldots X_s \) applied on \( X \), and for each traversal \( v \) of \( X \), one deforested visit-function \( \lambda_{P^v} \) is derived. The arguments of this visit-function are:

1. The attribute occurrences and grammar symbols that are alive at visit \( v \), \( \text{alive}(P, v) \),

2. The deforested visit-functions derived for the right-hand side symbols of \( P \) that are inspected in traversal \( v \) or later, \( i.e., \text{inspect}(P, v) \), and

3. The inherited attributes of traversal \( v \), \( i.e., \text{A}_{\text{inh}, v}(X, v) \).

The result is a tuple of which the first element is the partial parameterized function for the next traversal and the other elements are the synthesized attributes, \( i.e., \text{A}_{\text{syn}, v}(X, v) \). Thus, the deforested visit-function has the following signature:

\[
\lambda_{P^v} : \quad < \text{type}_{\text{pp} \_ \text{args}}(P, v) > \rightarrow (\text{T } \text{inh}_1) \rightarrow \cdots \rightarrow (\text{T } \text{inh}_k) \rightarrow \\
(\text{T } \lambda_{P^{v+1}}, \text{T } \text{syn}_1, \ldots, \text{T } \text{syn}_l)
\]

with \( \{\text{inh}_1, \ldots, \text{inh}_k\} = \text{A}_{\text{inh}, v}(X, v) \), \( \{\text{syn}_1, \ldots, \text{syn}_l\} = \text{A}_{\text{syn}, v}(X, v) \). The fragment < type_{pp_args}(P, v) > denotes the type of the elements in alive(P, v) and in inspect(P, v). This fragment is defined as follows:

\[< \text{type}_{\text{pp} \_ \text{args}}(P, v) > = (\text{T } a_1) \rightarrow \cdots \rightarrow (\text{T } a_m) \rightarrow (\text{T } v_1) \rightarrow (\text{T } v_n)\]

for all \( a_i \) such that \( a_i \in \text{alive}(P, v) \) and for all \( v_t \) such that \( v_t \in \text{inspect}(P, v) \). The function \( \text{T} \) is overloaded to work on visit instructions, as defined for the visit-tree evaluators.
The visit-function which performs the last traversal of non-terminal \( X \) does not return any partial parameterized visit-function. Its signature is:

\[
\lambda_{P} : < \text{type}_{\text{pp}}\_\text{args}(P, n) > \rightarrow (T \text{inh}_1) \rightarrow \cdots \rightarrow (T \text{inh}_k) \rightarrow (T \text{syn}_1, \ldots, T \text{syn}_l)
\]

Every visit \( v \) to a non-terminal symbol \( X \) induces a new type \( X^v \). It is defined as follows:

\[
\text{type } X^v = (T \text{inh}_1) \rightarrow \cdots \rightarrow (T \text{inh}_k) \rightarrow (X^{v+1}, T \text{syn}_1, \ldots, T \text{syn}_l)
\]

with \( A_{\text{inh},v}(X, v) = \{\text{inh}_1, \ldots, \text{inh}_k\} \) and \( A_{\text{syn},v}(X, v) = \{\text{syn}_1, \ldots, \text{syn}_l\} \). The last visit induces a different type:

\[
\text{type } X^n = (T \text{inh}_1) \rightarrow \cdots \rightarrow (T \text{inh}_k) \rightarrow (T \text{syn}_1, \ldots, T \text{syn}_l)
\]

Let us now define the code of the visit-function \( \lambda_{P} \):

\[
\lambda_{P} : < \text{par}\_\text{par}(P, i) > < \text{inherited}(i) > =
\]

\[
(\lambda_{P+1} < \text{par}\_\text{par}(P, i + 1) > , < \text{synthesized}(i) >)
\]

where \( < \text{body}(i) > \)

and the visit-function which performs the last traversal is:

\[
\lambda_{P} : < \text{par}\_\text{par}(P, n) > < \text{inherited}(i) > = (< \text{synthesized}(i) >)
\]

where \( < \text{body}(n) > \)

where the code fragments defining the inherited and synthesized attributes are:

\[
< \text{inherited}(i) > = \text{inh}_1 \text{inh}_2 \ldots \text{inh}_k
\]

\[
< \text{synthesized}(i) > = \text{syn}_1, \text{syn}_2, \ldots, \text{syn}_l
\]

The code fragment \( < \text{par}\_\text{par}(P, j) > \) denotes the partial parameterisation of the next visit-function.

\[
< \text{par}\_\text{par}(P, j) > = (\text{var } a_1) \ldots (\text{var } a_m) \lambda_{X_1^v} \ldots \lambda_{X_l^v}
\]

for all \( a_i \in \text{alive}(P, j) \) and with \( \{\text{visit } ((P, k), v), \ldots, \text{visit } ((P, r), w)\} = \text{inspect}(P, j) \).

The generation of the body \( < \text{body}(i) > \) is very similar to the visit-tree approach discussed previously. \texttt{eval} instructions induce exactly the same equations and, so, we will not repeat them here. We describe the mapping of \texttt{visit} instructions only. Every visit
Chapter 4. Deforested Attribute Evaluators

instruction \textit{visit} \((\langle P, c \rangle, w)\) defining the visit \(w\) to child \(c\) of \(P\), \textit{i.e.}, to the instance of the nonterminal \(X_c\), induces a recursive deforested visit-function call. Once again, two cases have to be distinguished:

If \(w < v(X_c)\) then, the visit-function call returns the partial parameterized function for the next traversal. The following equation is generated:

\[
(\lambda_{X_c}^{w+1}, syn_{-1}, \ldots, syn_{-j}) = \lambda_{X_c}^{w} \ inh_{-1} \ldots inh_{-j}
\]

If \(w = v(X_c)\) then, only the synthesized attributes are computed by the function call.

\[
(syn_{-1}, \ldots, syn_{-j}) = \lambda_{X_c}^{w} \ inh_{-1} \ldots inh_{-j}
\]

with \(\{inh_{-1}, \ldots, inh_{-j}\} = A_{inh,v}(\langle P, c \rangle, w)\) and \(\{syn_{-1}, \ldots, syn_{-j}\} = A_{syn,v}(\langle P, c \rangle, w)\).

Let us return to the \textit{block} grammar \(AG_1\) and derive the deforested visit-functions for production \textit{BLOCK}. Two visits are performed to this production. As a consequence, two deforested visit-function \(\lambda_{\text{Block}_1}\) and \(\lambda_{\text{Block}_2}\) are generated. Let us start by deriving their types. According to the static analysis performed in Section [4.1.1] these functions have the following types:

\[
\begin{align*}
\lambda_{\text{Block}_1} &:: \text{Its}_1 \rightarrow \text{Env} \rightarrow \text{Int} \rightarrow (\text{It}_2, \text{Env}) \\
\lambda_{\text{Block}_2} &:: \text{Int} \rightarrow \text{Its}_1 \rightarrow \text{Env} \rightarrow \text{Err}
\end{align*}
\]

where \(\text{Its}_1\) and \(\text{It}_2\) are the types derived for the two visits to non-terminals \(\text{Its}\) and \(\text{It}\). They are defined as follows:

\[
\begin{align*}
type \text{Its}_1 &= (\text{Env} \rightarrow \text{Int} \rightarrow (\text{Its}_2, \text{Env})) \\
type \text{Its}_2 &= (\text{Env} \rightarrow \text{Err}) \\
type \text{It}_1 &= (\text{Env} \rightarrow \text{Int} \rightarrow (\text{It}_2, \text{Env})) \\
type \text{It}_2 &= (\text{Env} \rightarrow \text{Err})
\end{align*}
\]

Once we have defined the types, we proceed by deriving the deforested visit-functions. The following fragments of the two visit-functions are induced by the visit-sub-sequences:

\[
\begin{align*}
\lambda_{\text{Block}_1} &< \text{par}\ par(\text{BLOCK},1) > lev
dcli = ((\lambda_{\text{Block}_2} < \text{par}\ par(\text{BLOCK},2) >, dcli) \\
\text{where} & < \text{body}(1) >
\end{align*}
\]

\[
\begin{align*}
\lambda_{\text{Block}_2} &< \text{par}\ par(\text{BLOCK},2) > env = \text{errs} \\
\text{where} & < \text{body}(2) >
\end{align*}
\]

where the fragments < \text{par}\ par > are:

\[
< \text{par}\ par(\text{BLOCK},1) > = \lambda_{\text{Its}_1} \\
< \text{par}\ par(\text{BLOCK},2) > = \text{lev}_1 \lambda_{\text{Its}_1}
\]

These fragments define how the attribute occurrence \(\text{It.lev}\) is passed from the first to the second traversal of the evaluator: it is simply a hidden argument and a hidden result of deforested visit-function \(\lambda_{\text{Block}_1}\) and \(\lambda_{\text{Block}_2}\), respectively.
The bodies of the visit-functions are derived from the corresponding visit-sub-sequences:

\[
<\text{body}(1)> = lev_1 = lev + 1 \\
dcli = dcli
\]

\[
<\text{body}(2)> = dcli_1 = env \\
(\lambda_{Its^2}, dclo_1) = \lambda_{Its^1} dcli_1 lev_1 \\
errs_1 = \lambda_{Its^2} dclo_1 \\
errs = errs_1
\]

Next, we present the complete \(\lambda\)-attribute evaluator derived for the block grammar \(AG_1\) (some copy rules were, once again, trivially removed from the evaluator’s code).

\[
\lambda_{It}\text{ts}^1 = errs_1 \\
\text{where}\ lev_1 = 0 \\
dcli_1 = [] \\
(\lambda_{Its^2}, dclo_1) = \lambda_{Its^1} dcli_1 lev_1 \\
errs_1 = \lambda_{Its^2} dclo_1
\]

\[
\lambda_{ConsIts^2} \cdot \lambda_{Its^2} dcli lev = (\lambda_{ConsIts^2} \cdot \lambda_{Its} \cdot \lambda_{Its^2} \cdot dclo) \\
\text{where}\ (\lambda_{Its^2}, dclo_1) = \lambda_{Its} dcli lev \\
(\lambda_{Its^2}, dclo) = \lambda_{Its^1} dclo_1 lev
\]

\[
\lambda_{NilIts^1} \cdot dcli lev = (\lambda_{NilIts^2}, dcli) \\
\lambda_{ConsIts^2} \cdot \lambda_{Its^2} env = errs \\
\text{where}\ errs_1 = \lambda_{Its^2} env \\
ehrs_2 = \lambda_{Its^2} env \\
enrs = errs_1 ++ errs_2
\]

\[
\lambda_{NilIts^2} \cdot env = errs \\
\text{where}\ errs = []
\]

Eval 4: The block \(\lambda\)-attribute evaluator for grammar \(AG_1\).

As a result of our techniques, all visit-functions have become combinators, i.e., they do not refer to global variables. The attribute evaluator is no longer defined by the single function derived for the root symbol of the AG, but, instead, by the set of deforested visit-functions. The type of the visit-function induced by the production applied to the root symbol is:

\[
\lambda_{RootP^1} :: \text{Its}^\dagger \rightarrow \text{Err}
\]

where \(\text{Its}^\dagger\) is now a function type.

As a result of generating Haskell code we inherit many useful properties of this language. The attribute evaluator Eval 4, for example, is completely polymorphic. Nothing in this evaluator is defined concerning the type of the identifiers of the language. The identifiers are provided by an external lexical analyser. They can be a sequence of characters, a single character or even a numeral. The deforested evaluator can be reused
in all those cases, provided that the semantic functions \( mBIn \) and \( mNBI_\) are defined on that type, too.

This approach has the following properties:

- The \( \lambda \)-attribute evaluators have the tendency to be more polymorphic.
- The evaluators are \textit{data type independent} and, thus, new semantics can be easily added: for example, new productions can be incorporated to a compiler, without having to change the evaluator. This property will be explained in Section 4.5.
- Attribute instances needed in different traversals of the evaluator are passed between traversals as results/arguments of partially parameterized visit-functions. No additional data structure has to be explicitly defined to handle them, like trees [Kas91b, PSV92, SKS97b], stacks and queues [odAS91] or “cactus stacks” [JPJ +90]. The runtime stack does the job for us.
- The resulting evaluators are higher-order attribute evaluators. The arguments of the evaluators visit-functions are other AE visit-functions.
- All the attributes are handled in a canonical way: they just show up as arguments and results of deforested visit-functions.
- No pattern matching is needed to detect the production applied to the node the evaluator is visiting.
- The visit-functions are strict in all their arguments, as a result of the order computed by the AG ordered scheduling algorithm.
- Efficient memory usage: data not needed is no longer referenced. References to grammar symbols and attribute instances can efficiently be discarded as soon as they have played their semantic role.
- The code of the attribute evaluator is shorter because no data structures are defined.

### 4.2 Deforestation of Lazy Attribute Evaluators

The deforestation approach is orthogonal to the lazy implementation of attribute grammars. That is, the lazy attribute evaluator can be partially parameterized with the abstract syntax tree, as well. The resulting attribute evaluator is a \textit{deforested circular program}.

The mapping from attribute grammars into deforested circular programs is as follows: for every production \( P : X_0 \rightarrow X_1 \ldots X_s \) a deforested function \( \lambda_P \) is generated. The arguments of \( \lambda_P \) are the deforested functions induced by symbols on the right-hand-side of \( P \) and the inherited attributes of \( X_0 \). The result is a tuple of the synthesized attributes of \( X_0 \). The deforested function \( \lambda_P \) has the following signature:
4.3. Deforestation of Higher-Order Attribute Grammars

\[ \lambda_P :: X_1 \to \cdots \to X_s \to (T \text{ inh}_1) \to \cdots \to (T \text{ inh}_k) \to (T \text{ syn}_1, \ldots, T \text{ syn}_l) \]

with \( A_{\text{inh}}(X_0) = \{\text{inh}_1, \ldots, \text{inh}_k\} \) and \( A_{\text{syn}}(X_0) = \{\text{syn}_1, \ldots, \text{syn}_l\} \).

Every non-terminal symbol \( X \in N \) induces a new type. It is defined as follows:

\[ \text{type } X = (T \text{ inh}_1) \to \cdots \to (T \text{ inh}_k) \to (T \text{ syn}_1, \ldots, T \text{ syn}_l) \]

For each production \( P \), a deforested function \( \lambda_P \) is generated, as follows:

\[ \lambda_P \lambda_{X_1} \ldots \lambda_{X_s} \text{ inh}_1 \ldots \text{ inh}_k = (\text{syn}_1, \ldots, \text{syn}_l) \]

where body

The \textit{body} is the translation of the attribute equations for \( P \). It is very similar to the mapping presented in Section 3.5. We will not repeat it here.

We present next the deforested lazy attribute evaluator for block \( AG_1 \).

\[ \lambda_R = \lambda_{Its} = \text{err}_s \]
\[ \text{where } \quad \text{lev}_1 = 0 \]
\[ \text{dcli}_1 = [] \]
\[ (\text{dcl}_0, \text{err}_s) = \lambda_{Its} \text{ dcli}_1 \text{ lev}_1 \text{ dcl} \]

\[ \lambda_{Cons} = \lambda_{Its} \text{ dcli lev env} = (\text{dcl}_0, \text{err}_s) \]
\[ \text{where } \quad \text{dcli} = (\text{PAIR } \text{tName } \text{lev}) : \text{dcl} \]
\[ \text{errs} = (\text{PAIR } \text{tName } \text{lev}) \quad ‘\text{mNBIn}’ \text{ dcl} \]

\[ \lambda_{Use} = \lambda_{Its} \text{ dcli lev} = (\text{dcli}, \text{err}_s) \]
\[ \text{where } \quad \text{errs} = \text{tName} \quad ‘\text{mBIn}’ \text{ env} \]

\[ \lambda_{Block} = \lambda_{Its} \text{ dcli lev env} = (\text{dcli}, \text{err}_s) \]
\[ \text{where } \quad \text{lev} = \text{lev} + 1 \]
\[ (\text{dcl}_0, \text{err}_s) = \lambda_{Its} \text{ env lev} \text{ dcl} \]

\[ \text{Eval 5: The block deforested circular attribute evaluator for grammar } AG_1. \]

The type of the visit-function induced by the production applied to the root symbol is:

\[ \lambda_R :: \text{Its} \to \text{Err} \]

4.3 Deforestation of Higher-Order Attribute Grammars

A higher-order attribute grammar may have several higher-order attributes (i.e., higher-order trees). Thus, an attribute evaluator for HAG may contain a large number of higher-order trees: one higher-order tree per instance of one higher-order attribute. So, a large
number of trees may be constructed, traversed and destructed during attribute evaluation. Consequently, the efficiency of the attribute evaluator may be affected by such dynamic manipulation of higher-order trees.

A more efficient attribute evaluator is obtained if those higher-order trees are deforested. Observe that higher-order attributes are intermediate data structures (i.e., the “glue”) which are constructed by the attribute evaluator during decoration. Consequently, we can easily adapt our deforestation techniques to eliminate higher-order trees as well. The basic idea is that visit-functions that decorate a higher-order attribute are partially parameterized with the higher-order trees. As a result, the evaluator directly builds the computation defined by the higher-order attribute, instead of constructing the higher-order tree in the first place. Before we present the deforestation of higher-order attribute grammars, we begin by discussing the functional evaluation of higher-order attribute grammars.

4.3.1 Evaluation of Higher-Order Attribute Grammars

A lazy or a strict attribute evaluator for higher-order attribute grammars can be obtained by using standard attribute grammar techniques. As we have explained in Section 2.5.2, the key idea is to reduce the higher-order attribute grammar into an equivalent first-order one. After that, standard algorithms can be applied to the first-order grammar. The implementation of attribute grammars as lazy attribute evaluators, for example, can be immediately obtained from the first-order grammar. That is to say, higher-order attribute grammars can be easily implemented as (deforested) circular programs by using the mappings presented in Sections 3.5 and 4.2.

Higher-order attribute grammars can be efficiently implemented as strict, multiple traversal attribute evaluators, too. This section briefly describes how such attribute evaluators are obtained from a HAG specification. After that, we show how to deforestate such evaluators.

The strict implementation of HAG is based on the visit-sequence paradigm, exactly as for first-order attribute grammars. Visit-sequences for a higher-order attribute grammar $G$ can be obtained by first reducing $G$ into a plain grammar $G'$, then computing visit-sequences $vss'$ for $G'$, and, finally, translating $vss'$ to visit-sequences $vss$ for $G$.

In order to handle HAG, the visit-sequences are extended to express two things: a visit and the instantiation of an attributable attribute. Thus, a visit instruction of the form visit $(p.x, v)$ denotes that attributable attribute $p.x$ is visited for the $v$th time. The instruction eval $(x)$ denotes the computation of the value of attribute $x$ and the instantiation of $x$ if $x$ is an attributable attribute.

To present the functional implementation of HAGs let us consider the first higher-order variant of the BLOCK grammar $AG_2$, where a single higher-order attribute is defined: the attributable attribute Root.ast. After reducing $AG_2$, we obtain the visit-sequences presented in Visit-Sequences 2.11. We show the annotated visit-sub-sequences for production Root, where the ata Root.ast is defined, and we also present the productions defining the
4.3. Deforestation of Higher-Order Attribute Grammars

The visit-sequences induced for the productions of the abstract grammar (Fragment 2) are exactly the same as the ones presented in Visit-Sequences 1. We do not repeat them here.

<table>
<thead>
<tr>
<th>visit OneSts</th>
<th>eval OneSts</th>
<th>eval OneSts</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{inh}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
</tr>
<tr>
<td>(( \text{Stat} ), ( \text{LstSts} ))</td>
<td>( \text{eval}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{uses}(\text{ast}) )</td>
<td>( \text{use}(\text{ast}) )</td>
<td>( \text{use}(\text{ast}) )</td>
</tr>
<tr>
<td>(( \text{ast} ))</td>
<td>( \text{eval}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{end} 1 )</td>
<td>( \text{syn}(\text{ast}) )</td>
<td>( \text{syn}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{plan OneSts} )</td>
<td>( \text{plan NoSts} )</td>
<td>( \text{plan ConsLstSts} )</td>
</tr>
<tr>
<td>( \text{begin} 1 )</td>
<td>( \text{begin} 1 )</td>
<td>( \text{begin} 1 )</td>
</tr>
<tr>
<td>( \text{inh}(\text{ast}) )</td>
<td>( \text{inh}(\text{ast}) )</td>
<td>( \text{inh}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{visit}(\text{ast}) )</td>
<td>( \text{visit}(\text{ast}) )</td>
<td>( \text{visit}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{eval}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{use}(\text{ast}) )</td>
<td>( \text{use}(\text{ast}) )</td>
<td>( \text{use}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{end} 1 )</td>
<td>( \text{syn}(\text{ast}) )</td>
<td>( \text{syn}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{plan NoSts} )</td>
<td>( \text{plan NoSts} )</td>
<td>( \text{plan NoSts} )</td>
</tr>
<tr>
<td>( \text{begin} 1 )</td>
<td>( \text{begin} 1 )</td>
<td>( \text{begin} 1 )</td>
</tr>
<tr>
<td>( \text{inh}(\text{ast}) )</td>
<td>( \text{inh}(\text{ast}) )</td>
<td>( \text{inh}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{eval}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{use}(\text{ast}) )</td>
<td>( \text{use}(\text{ast}) )</td>
<td>( \text{use}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{end} 1 )</td>
<td>( \text{syn}(\text{ast}) )</td>
<td>( \text{syn}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{plan NoSts} )</td>
<td>( \text{plan NoSts} )</td>
<td>( \text{plan NoSts} )</td>
</tr>
<tr>
<td>( \text{begin} 1 )</td>
<td>( \text{begin} 1 )</td>
<td>( \text{begin} 1 )</td>
</tr>
<tr>
<td>( \text{inh}(\text{ast}) )</td>
<td>( \text{inh}(\text{ast}) )</td>
<td>( \text{inh}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{eval}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
<td>( \text{eval}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{use}(\text{ast}) )</td>
<td>( \text{use}(\text{ast}) )</td>
<td>( \text{use}(\text{ast}) )</td>
</tr>
<tr>
<td>( \text{end} 1 )</td>
<td>( \text{syn}(\text{ast}) )</td>
<td>( \text{syn}(\text{ast}) )</td>
</tr>
</tbody>
</table>

Visit-Sequences 2: The annotated visit-sub-sequences induced by grammar \( AG_2 \).

As expected, the non-terminals of the concrete grammar are visited exactly once, in which visit the abstract syntax tree is synthesized. Note that no inherited attributes were associated with those symbols, and consequently, they can be evaluated in a purely bottom-up strategy. This feature will become important in the next section.

Let us analyse in more detail the visit-sequences computed by the chained scheduling algorithm for production \( \text{Root} \). First, the subtree rooted \( \text{Stat} \) is visited (i.e., the concrete tree) and attribute \( \text{Stats.ast} \) is computed. Next, the attributable attribute \( \text{ast} \) is instantiated with that tree (i.e., the abstract syntax tree) and its generated attributes are computed. After that, the higher-order tree \( \text{ast} \) is visited twice to synthesize attribute \( \text{ast.errs} \), i.e., the list of errors. This list is the synthesized value of the visit-sequence. As expected, the visit-sequences induced by productions \( \text{OneSts} \) and \( \text{NoSts} \) are “normal” visit-sequences: no special instructions are induced, since no \( \text{atas} \) are defined in those productions. The same holds for the productions of the abstract grammar.

The expanded visit-sub-sequences can easily be mapped into visit-tree based attribute evaluators. However, there are three cases which slightly differ from the mapping presented in Section 4.1.2: the instantiation of an \( \text{ata} \) (referenced with \( I \) in the previous visit-
sequences), the instantiation of its inherited attributes (II) and a visit to an ata (III). The first case is easily solved since atas are viewed as local attributes of productions: they are literally copied to the body of the visit-function. To solve the second case, we extend our techniques to handle generated attribute occurrences. Every attribute equation of the form

\[
\text{eval } (P.x.a) \\
\text{uses} (\text{args})
\]

defining a generated attribute occurrence \(P.x.a = f \ \text{args}\), induces an equation \(a_x = f \ \text{args}\). Attribute occurrence \(P.y.b\) occurring in \(\text{args}\) is replaced by \(a_y\). In other words, generated attribute occurrences are replaced by variables which are labelled with the name of the attributable attribute.

Finally, we consider the third case. Every visit instruction \(\text{visit } (P.x, v)\) defining the visit \(v\) to ata \(P.x\) of type \(X\), i.e., \(X = (T \ P.x)\), induces a recursive visit-function call as follows:

\[
(x^{v+1}, \text{syn}_1 x, \ldots, \text{syn}_k x) = \text{visit}_v X x^v \ \text{inh}_1 x \ldots \text{inh}_k x
\]

\[A_{\text{inh}_{v}}(X, v) = \{ \text{inh}_1, \ldots, \text{inh}_k \}\] and \[A_{\text{syn}_{v}}(X, v) = \{ \text{syn}_1, \ldots, \text{syn}_l \}\]. That is, terms of type \(X\) are visited for the \(v\)th time.

We are now ready to derive the visit-tree based visit-function from the previous visit-sub-sequences. As a result of our attribute grammar lifetime analysis we have the following sets:

\[
\begin{align*}
\text{alive}(\text{ROOT}, 1) & = \{ \} & \text{inspect}(\text{ROOT}, 1) & = \{ \text{visit}(\text{Stats}, 1) \} \\
\text{alive}(\text{ONESTS}, 1) & = \{ \} & \text{inspect}(\text{ONESTS}, 1) & = \{ \text{visit}(\text{LstSts}, 1) \} \\
\text{alive}(\text{NOSTS}, 1) & = \{ \} & \text{inspect}(\text{NOSTS}, 1) & = \{ \} \\
\text{alive}(\text{ASTAT}, 1) & = \{ \} & \text{inspect}(\text{ASTAT}, 1) & = \{ \text{visit}(\text{Stat}, 1) \} \\
\text{alive}(\text{CONSLESTSTS}, 1) & = \{ \} & \text{inspect}(\text{CONSLESTSTS}, 1) & = \{ \text{visit}(\text{Stat}, 1), \text{visit}(\text{LstSts}, 1) \} \\
\text{alive}(\text{CDECL}, 1) & = \{ \text{Name} \} & \text{inspect}(\text{CDECL}, 1) & = \{ \} \\
\text{alive}(\text{CUSE}, 1) & = \{ \text{Name} \} & \text{inspect}(\text{CUSE}, 1) & = \{ \} \\
\text{alive}(\text{CBLOCK}, 1) & = \{ \} & \text{inspect}(\text{CBLOCK}, 1) & = \{ \text{visit}(\text{Stats}, 1) \}
\end{align*}
\]

Note that, although in the single sub-sequence of production \(\text{ROOT}\) two visit instructions are associated to ata \(\text{ROOT.ast}\), no reference is included in \(\text{inspect}(\text{ROOT}, 1)\) because \(\text{ROOT.ast} \not\in \text{rhs}(\text{ROOT})\). The derived visit-tree data types are:

\[
\text{data } \text{Prog} = \text{ROOT} \ 	ext{Stats} \\
\text{data } \text{Stats} = \text{ONESTS} \ 	ext{LstSts} | \text{NOSTS} \\
\text{data } \text{LstSts} = \text{ASTAT} \ 	ext{Stat} | \text{CONSLESTSTS} \ 	ext{Stat} \ 	ext{LstSts} \\
\text{data } \text{Stat} = \text{CDECL} \ 	ext{Name} | \text{CUSE} \ 	ext{Name} | \text{CBLOCK} \ 	ext{Stats}
\]

These two data types are exactly the types we have defined in Section 2.6.1. The visit-tree data types derived for the productions of the abstract grammar were derived in Section 4.1.2. So, we do not repeat them here.
Next, we present the visit-functions derived for non-terminals $Prog$ and $Stats$. We omit the visit-functions $\text{visit}_1LstSts$ and $\text{visit}_1Stat$ which can be trivially derived from the visit-sequences. Furthermore, the visit-functions derived for non-terminals $Its$ and $It$ are the ones presented in Evaluator $Eval^{382}$.

\begin{align*}
\text{visit}_1Prog (\text{Root } tStats) &= \text{errs} \\
\text{where} & \quad ast_1 = \text{visit}_1Stats tStats \\
& \quad ast = ast_1 \\
& \quad lev_{ast} = 0 \\
& \quad dcli_{ast} = [] \\
& \quad (ast^2, dclo_{ast}) = \text{visit}_1Its ast dcli_{ast} lev_{ast} \\
& \quad env_{ast} = dclo_{ast} \\
& \quad errs_{ast} = \text{visit}_2Its ast^2 env_{ast} \\
& \quad errs = errs_{ast}
\end{align*}

and the types of the two functions are:

\begin{align*}
\text{visit}_1Prog & : Prog \rightarrow \text{Err} \\
\text{visit}_1Stats & : Stats \rightarrow \text{Its}
\end{align*}

Observe that the visit-function $\text{visit}_1Prog$, when executed with the concrete syntax tree represented in Figure 2.1, will compute as an intermediate data structure the correspondent abstract syntax tree (the tree in the right of the same figure). This tree is the result of the call to $\text{visit}_1Stats$. The root non-terminal symbol has one traversal only. As a result, the attribute evaluator for $AG_2$ is defined by the single function derived for that symbol. So, we have:

\begin{align*}
evaluator & : Prog \rightarrow \text{Err} \\
evaluator &= \text{visit}_1Prog
\end{align*}

As expected, the evaluator is a function that takes as argument a term of type $Prog$ and returns a value of type $\text{Err}$.

### 4.3.2 Deriving Deforested Evaluators for HAGs

One problem arises when higher-order attribute grammars are deforested: how to handle “deforested terms” induced by grammar symbols that are syntactically referenced in the attribute equations of the HAG? Observe that, if we use our previous techniques, the equations of the derived deforested attribute evaluators would expect terms, and not functions, as arguments. So, under the deforestation approach, the attribute equations have to be transformed in order to handle partially parameterized visit-functions, instead of terms. That is, the attribute equations and the semantic functions of the HAG have to be deforested.
To simplify the discussion, we assume that all the inductive computations defined over the non-terminal symbols of the HAG are defined through attribution, and not through semantic functions. In other words, attribute equations do not inspect, i.e., do not pattern match terms induced by non-terminal symbols. Let us recall that within the higher-order formalism, semantic functions are redundant and every inductive computation can be defined through attribution.

We proceed now with the definition of the mapping from higher-order attribute grammars into λ-attribute evaluators. We shall focus again on the three cases that differ slightly from the first-order grammar, as referenced in Visit-Sequences $\Sigma$. First, higher-order attributes are not instantiated with higher-order trees, but with their initial deforested visit-functions. So, for every instruction of the form

$$\text{eval (P.x)}$$
$$\text{uses(args)}$$

defining the instantiation of attributable attribute P.x, an equation $x = f \ args$ is induced.

Secondly, every attribute equation of the form

$$\text{eval (P.x.a)}$$
$$\text{uses(args)}$$

defining a generated attribute occurrence $P.x.a = f \ args$, induces an equation $a_x = f \ args$. Attribute occurrence $P.y.b$ occurring in $args$ is replaced by $a_y$.

Thirdly, every visit instruction $\text{visit (P.x, v)}$ defining the visit $v$ to ata $P.x$ of type $X$, i.e., $X = (T_\ P.x)$, induces a recursive visit-function call as follows:

$$\lambda_{\text{attr} + 1, syn_1, \ldots, syn_k} = \lambda_{\text{attr} - inh_1, \ldots, inh_k}$$

with $A_{inh,v}(X, v) = \{inh_1, \ldots, inh_k\}$ and $A_{syn,v}(X, v) = \{syn_1, \ldots, syn_k\}$.

The constructor functions induced by the productions $P$ of the HAG, which occur in the attribute equations of the grammar are deforested too. They are represented by their initial deforested visit-functions. That is, they are replaced by the deforested visit-functions that represents their first visit. So, for every equation of the form

$$\text{eval (a)}$$
$$\text{uses(...)}$$

one equation is generated as follows $a = \text{equation}$. Every occurrence of a constructor $C$ in equation, with $C \in P$, is replaced by $\lambda_{C_1}$.

We are now ready to derive the deforested attribute evaluator for the block HAG $AG_2$. Next, we present the complete λ-attribute evaluator:
4.3. Deforestation of Higher-Order Attribute Grammars

\[
\begin{align*}
\lambda_{\text{Root}} & : \lambda_{\text{Stats}} = \text{errs}_{\text{ast}} \\
\text{where} & \ \
\text{ast}_1 &= \lambda_{\text{Stats}} \\
\lambda_{\text{ast}} & = \text{ast}_1 \\
\text{lev}_{\text{ast}} &= 0 \\
\text{deli}_{\text{ast}} &= \emptyset \\
(\lambda_{\text{ast}2}, \text{dcl}_{\text{ast}}) &= (\lambda_{\text{ast}}, \text{dcl}_{\text{ast}} \text{ lev}_{\text{ast}}) \\
\text{errs}_{\text{ast}} &= \lambda_{\text{ast}2} \text{ dcl}_{\text{ast}} \\
\lambda_{\text{CDecl}} & : t\text{Name} = \text{ast} \\
\text{where} & \ \
\text{ast} &= \lambda_{\text{Decl1}} t\text{Name} \\
\lambda_{\text{ConsIts}} & : \lambda_{\text{Its1}} \lambda_{\text{Its2}} \text{ deli} \text{ lev} = \\
& (\lambda_{\text{ConsIts}} \lambda_{\text{Its2}} \lambda_{\text{Its2}}, \text{ delo}) \\
\text{where} & \ \
(\lambda_{\text{Its2}}, \text{dcl}_{\text{Its1}}) &= \lambda_{\text{Its1}} \text{ deli} \text{ lev} \\
(\lambda_{\text{Its2}}, \text{dcl}_{\text{Its1}}) &= \lambda_{\text{Its2}} \text{ dcl}_{\text{Its1}} \text{ lev} \\
\lambda_{\text{NilIts}} & : \text{deli} \text{ lev} = (\lambda_{\text{NilIts2}}, \text{ deli}) \\
\lambda_{\text{Decl1}} & : t\text{Name} \text{ deli} \text{ lev} = (\lambda_{\text{Decl2}} \text{ errs}, \text{ delo}) \\
\text{where} & \ \
\text{dcl} &= (t\text{Name}, \text{lev}) : \text{ deli} \\
\text{errs} &= (t\text{Name}, \text{lev}) \text{ mNBIn} \text{ dcl} \\
\lambda_{\text{Use1}} & : t\text{Name} \text{ deli} \text{ lev} = (\lambda_{\text{Use2}} t\text{Name}, \text{ deli}) \\
\lambda_{\text{Block1}} & : \text{lev}_1 \lambda_{\text{Its1}} \text{ deli} \text{ lev} = (\lambda_{\text{Block2}} \text{ lev}_1 \lambda_{\text{Its1}} \text{ deli}) \\
\text{where} & \ \
\text{lev}_1 &= \text{lev} + 1
\end{align*}
\]

\[
\begin{align*}
\lambda_{\text{ConsLstIts}} & : \lambda_{\text{Stats}} \lambda_{\text{Stat}} \lambda_{\text{LstStats}} = \text{ast} \\
\text{where} & \ \
\text{ast}_1 &= \lambda_{\text{Stat}} \\
\text{ast}_2 &= \lambda_{\text{LstStats}} \\
\text{ast} &= \lambda_{\text{ConsIts}} \text{ ast}_1 \text{ ast}_2 \\
\lambda_{\text{AStat}} & : \lambda_{\text{Stat}} = \text{ast} \\
\text{where} & \ \
\text{ast}_1 &= \lambda_{\text{Stat}} \\
\text{ast} &= \lambda_{\text{ConsIts}} \text{ ast}_1 \lambda_{\text{NilIts}} \\
\lambda_{\text{CUse}} & : t\text{Name} = \text{ast} \\
\text{where} & \ \
\text{ast} &= \lambda_{\text{Use1}} t\text{Name} \\
\lambda_{\text{ConsIts2}} & : \lambda_{\text{Its1}} \lambda_{\text{Its2}} \text{ env} = \text{errs} \\
\text{where} & \ \
\text{errs}_1 &= \lambda_{\text{Its1}} \text{ env} \\
\text{errs}_2 &= \lambda_{\text{Its2}} \text{ env} \\
\text{errs} &= \text{errs}_1 + \text{errs}_2 \\
\lambda_{\text{NilIts2}} & : \text{env} = \text{errs} \\
\text{where} & \ \
\text{errs} &= [] \\
\lambda_{\text{Decl2}} & : \text{errs} \text{ env} = \text{errs} \\
\text{where} & \ \
\text{errs} &= \text{tName} \text{ mBNIn} \text{ env} \\
\lambda_{\text{Use2}} & : \text{tName} \text{ env} = \text{errs} \\
\text{where} & \ \
\text{errs} &= \text{tName} \text{ mBNIn} \text{ env} \\
\lambda_{\text{Block2}} & : \text{lev}_1 \lambda_{\text{Its1}} \text{ env} = \text{errs}_1 \\
\text{where} & \ \
(\lambda_{\text{Its2}}, \text{dcl}_{\text{Its1}}) &= \lambda_{\text{Its1}} \text{ env} \text{ lev}_1 \\
\text{errs}_1 &= \lambda_{\text{Its2}} \text{ dcl}_{\text{Its1}}
\end{align*}
\]

\[
\text{Eval 6: The BLOCK \lambda-attribute evaluator for grammar } \text{AG}_2.\]

In order to help in understanding this evaluator, we derive the types of the visit-functions. According to Visit-Sequences non-terminals \textit{Stats}, \textit{LstStats} and \textit{Stat} have a single visit. Because these symbols do not have inherited attributes, the single visit simply synthesizes the attribute \textit{ast} of type \textit{Its}. Consequently, the following types are generated:

\[
\begin{align*}
type \text{Stats} &= \text{Its} \\
type \text{LstStats} &= \text{Its} \\
type \text{Stat} &= \text{It}
\end{align*}
\]

Note that the type \text{Its} and \text{It} is the type of attribute \textit{ast} (see the BLOCK AG). In the higher-order mapping such types are labelled with number 1, \textit{i.e.}, the number of the first traversal to be performed on those terms. Informally, the above types define that the first (and single) visit to the concrete tree (non-terminal \textit{Stats}) returns as its result the function that performs the first visit to the abstract tree (non-terminal \textit{Its}). Note also that the types \text{Its} and \text{It} denote a function type.

Let us now focus on the types of the visit-functions derived for the productions of the concrete grammar. The following types follow directly from the \textit{alive} and \textit{inspect} sets
presented in the previous section.

\[
\begin{align*}
\lambda_{\text{OneSts}^1} & :: \text{LstSts}^1 \rightarrow \text{Its}^1 \\
\lambda_{\text{NoSts}^1} & :: \text{Its}^1 \\
\lambda_{\text{ConsLstSts}^1} & :: \text{Stat}^1 \rightarrow \text{LstSts}^1 \rightarrow \text{Its}^1 \\
\lambda_{\text{AStat}^1} & :: \text{Stat}^1 \rightarrow \text{Its}^1 \\
\lambda_{\text{CDecl}^1} & :: \text{Name} \rightarrow \text{It}^1 \\
\lambda_{\text{CUse}^1} & :: \text{Name} \rightarrow \text{It}^1 \\
\lambda_{\text{CBlock}^1} & :: \text{Stats}^1 \rightarrow \text{It}^1
\end{align*}
\]

Observe that this evaluator does not construct the syntax tree when decorating the concrete one, as defined by the underlying HAG. As we can see in the boxed function calls, the deforested evaluator directly calls the visit-functions that represent the decoration of the abstract tree.

In this evaluator the environment and the list of errors are not deforested. Let us recall that in HAG $AG_2$ the environment and the list of errors are not defined through attribution. Thus, their constructor functions are not deforested, since they are not productions of $AG_2$. A different situation occurs when we consider the grammar $AG_3$. In this grammar, the environment and its searching operations are defined through attribution. As a result, when using deforestation both the data structure that represents the environment and the visit-functions that define its searching functions are deforested. Consequently, a more efficient evaluator is obtained, since no redundant “glue” has to be constructed.

### 4.4 Deforested Attribute Grammar System

Traditionally, attribute grammar systems \cite{RT89,JPJ90,GHL92,GE90} construct a syntax tree during the parsing of the source text. This tree is used later as the underlying data structure which guides a multiple traversal attribute evaluator. For some classes of attribute grammars, e.g., one-pass attribute grammars \cite{Alb91a}, the construction of the syntax tree may be avoided and, consequently, the attribute evaluation may be performed in conjunction with the parsing, the so-called parse-time attribute evaluation. In this case, it is the parser which guides the attribute evaluation. Such model has several advantages, namely speed and space requirements. There are methods that make one-pass attribute evaluation during parsing possible \cite{ASU86,Alb91a}. Such methods, however, consider the elimination of the abstract syntax tree for one-pass attribute grammars only.

This section presents a more general method to eliminate the construction of syntax trees and to perform attribute evaluation during parsing. This method is based on the $\lambda$-attribute evaluators presented in Sections 4.1 and 4.3. Parse-time attribute evaluation is obtained as a by-product of our attribute grammar implementation: the parser directly calls the deforested visit-functions which perform the first traversal of the $\lambda$-attribute evaluators.

Before presenting our approach, we start by defining the complete AG system for the {f block} example. The scanner and the parser were defined in Chapter 2. Recall that
function \texttt{parsable} has the following type
\[
\texttt{parsable} :: \texttt{[Char]} \rightarrow \texttt{Maybe Prog}
\]
and the visit-tree based attribute evaluator which decorates terms of type \texttt{Prog}, has type:
\[
\texttt{evaluator} :: \texttt{Prog} \rightarrow \texttt{Err}
\]
Now, we are finally ready to define the desired program \texttt{block}.
\[
\texttt{block \ s} = \text{case } (\texttt{parsable \ s}) \text{ of}\\
(\texttt{Just \ t}) \rightarrow \texttt{evaluator \ t}\\
(\texttt{Nothing}) \rightarrow []
\]

Remember that the definition of the parser function which we have defined for the root symbol \texttt{Prog} of the \texttt{BLOCK AG}:
\[
\texttt{prog} = \texttt{sfRoot } <\$> \texttt{ symbol TkOB } \leftrightarrow \texttt{ stats } \leftrightarrow \texttt{ symbol TkCB}\\
\texttt{where} \quad \texttt{sfRoot \ a \ b \ c} = \texttt{ROOT \ b}
\]
which has type
\[
\texttt{prog} :: \texttt{[Token]} \rightarrow [([\texttt{Prog}], \texttt{[Token]})]
\]

Generally, parsers are automatically derived from the AG specification. Combinator parsers are no exceptions. The previous parser function, for example, can be trivially inferred from the \texttt{BLOCK AG}. The \texttt{alive} sets define which symbols are alive at visit 0: by definition the visit of the parser. Thus we have:
\[
\texttt{alive} (\texttt{ROOT},0) = \{ '^[, \texttt{Stats}, ^]' \}
\]
which defines the symbols of the concrete syntax. The symbols needed by the semantic function are the ones alive in visit 1:
\[
\texttt{alive} (\texttt{ROOT},1) = \{ \}
\texttt{inspect} (\texttt{ROOT},1) = \{ \texttt{visit} (\texttt{Stats},1) \}
\]
As we have explained before, symbols that do not play a role in the semantic analysis of the language are left out from the concrete tree.

The above parser function computes a syntax tree as expressed by its type (boxed type). Using the deforested approach, no tree data type is defined, and, therefore, no tree constructor functions can be used. Under the deforested approach, the parser function derived from the AG includes a call to the deforested visit-function which performs the first traversal. Thus, the equivalent deforested parser function looks as follows:
\[
\texttt{prog'} = \texttt{sfRoot } <\$> \texttt{ symbol TkOB } \leftrightarrow \texttt{ stats } \leftrightarrow \texttt{ symbol TkCB}\\
\texttt{where} \quad \texttt{sfRoot \ a \ b \ c} = \lambda_{\texttt{Root}^t} b
\]
which has type
\[
\texttt{prog'} :: \texttt{[Token]} \rightarrow [([\texttt{Name}], \texttt{[Token]})]
\]
The deforested visit-functions are partially parameterized with the arguments available at parse-time. Those arguments are other visit-functions that are partially parameterized during the parsing of the grammar symbols of the right-hand side of the production. No explicit tree is constructed. This property is reflected in the type of the parser function: function \texttt{prog} returns now the list of errors (boxed type). The complete deforested parser for BLOCK is presented next.

\begin{align*}
\texttt{prog} &= \texttt{sfRoot} \ <$\$> \ symbol \ TkOB \ <$\$> \ stats \ <$\$> \ symbol \ TkCB \\
\text{where} \quad \texttt{sfRoot a b c} &= \lambda_{\text{Root}^1} b \\
\texttt{stats} &= \lambda_{\text{OnSts}^1} \ <$\$> \ lststs \ <$\$> \ succeed \ \lambda_{\text{NoSts}^1} \\
\texttt{lststs} &= \texttt{sfConslststs} \ <$\$> \ stat \ <$\$> \ symbol \ TkSC \ <$\$> \ lststs \\
\text{where} \quad \texttt{sfConslststs a b c} &= \lambda_{\text{ConsLstSts}^1} a c \\
\texttt{stat} &= \texttt{sfDcl} \ <$\$> \ symbol \ TkDCL \ <$\$> \ ident \\
\text{where} \quad \texttt{sfDcl} a b &= \lambda_{\text{CDecl}^1} b \\
\texttt{sfUse} a b &= \lambda_{\text{CUse}^1} b \\
\texttt{sfBlk} a b c &= \lambda_{\text{CBlock}^1} b \\
\end{align*}

\textit{Program 10:} The deforested parser combinators for the BLOCK language.

When parsing a source text, the deforested visit-functions are partially parameterized and, thus, \textit{partially evaluated} at parse-time. They are totally evaluated as soon as they get all required arguments. Some of these functions, however, get all the arguments they need at parse-time, and, consequently, their results may be computed immediately.

Consider, for example, the simple deforested visit-function \(\lambda_{\text{NoSts}^1}\). This function is a constant function: it does not depend on argument values and always returns \(\lambda_{\text{NilIts}^1}\). In other words, function \(\lambda_{\text{NoSts}^1}\) can be totally evaluated at parse-time. The same holds for all the deforested visit-functions defining the concrete syntax tree of BLOCK: all their arguments are available at parse-time. The result of their evaluation is a partial parameterization of the visit-functions defining the abstract syntax of BLOCK. The final computations of these functions, however, have to be suspended during parsing, because they have to wait for the remaining arguments.

Generally, every visit-function derived from a visit-sub-sequence \(i\) which neither has inherited attributes (annotation \texttt{inh}) nor uses its inherited ones, can be evaluated in visit \(i - 1\). It has got all the arguments it needs from the previous visit. This is particularly important when implementing processors that produce code as the the input is being processed, \textit{i.e.}, for implementing online algorithms.
4.5 Extending the Grammar

One key property of deforested attribute evaluators is that they are independent of any particular data type definition. This property makes the evaluators highly reusable and new semantics can easily be added to an existent attribute evaluator.

For example, suppose that we want to extend the BLOCK language with named blocks. That is, the BLOCK AG is extended with the following production:

\[
\text{nameBlk} : \text{Stat} \rightarrow \text{Name} ' [' \text{Stats} ' ]'
\]

In traditional AG implementations, the attribute evaluator would have to be modified, since the type of the abstract syntax tree changes. Our implementation, however, is independent of the abstract tree data type. The attribute evaluator \( \mathit{Eval} \) can be reused, without any modification, to implement the AG extension. The only part of the compiler that has to be modified is the parser function for non-terminal \( \text{Stat} \). Note that the new production must be included, obviously. As for the visit-functions \( \lambda_{\text{NamedBlk}} \), which implement the different visits to the production, they have to be added to the evaluator as a separate module. The new parser fragment looks as follows:

\[
\begin{align*}
\text{stat} & = \text{sfDcl} \quad \text<s> \text{symbol TkDCL} \leftrightarrow \text{ident} \\
\text{<|> sfUse} & = \text<s> \text{symbol TkUSE} \leftrightarrow \text{ident} \\
\text{<|> sfBlk} & = \text<s> \text{symbol TkOB} \leftrightarrow \text{stats} \leftrightarrow \text{TkCB} \\
\text{<|> sfNamedBlk} & = \text<s> \text{ident} \leftrightarrow \text{symbol TkOB} \leftrightarrow \text{stats} \leftrightarrow \text{TkCB}
\end{align*}
\]

\textbf{where}

\[
\begin{align*}
\text{sfDcl} \ a \ b & = \lambda_{\text{CDecl}} \ b \\
\text{sfUse} \ a \ b & = \lambda_{\text{CUse}} \ b \\
\text{sfBlk} \ a \ b \ c & = \lambda_{\text{CBlock}} \ b \\
\text{sfNamedBlk} \ a \ b \ c \ d & = \lambda_{\text{NamedBlk}} \ b \ c
\end{align*}
\]

The signature of the visit-functions \( \lambda_{\text{NamedBlk}} \) must follow the interface, \textit{i.e.} the types, of the non-terminal symbols \( \text{Stat} \) and \( \text{Stats} \).

This property of our AG implementation is particularly important when designing language processors, in a component based style: AG components and the respective evaluators can easily be reused and updated, even when separate analysis and compilation of such components is considered. This is the subject of next chapter.
Chapter 5

Generic Attribute Grammars

Summary

This chapter presents generic attribute grammars: a support for genericity, reusability and modularity for attribute grammars. Generic attribute grammars, flow types and generic attribute evaluators are defined. The semantic compositionality of these attribute grammars components is discussed.

Modern software applications are nowadays designed and implemented as combinations of several generic components, which are physically and conceptually separated from each other. The benefits of such an organization are ease of specification, clearness of description, interchangeability between different “plug-compatible” components, reuse of components across applications and separation of analysis and compilation of components.

Consider, for example, the construction of a compiler for a particular language. Such a compiler may be constructed out of a set of generic components, each of which describes a particular subproblem such as name analysis, type checking, register allocation, etc. Furthermore, each component may describe the properties of a subproblem for a large number of languages, and not just for the language at hand: a single component may describe the name analysis task for block structured languages. Thus, this “off the shelf” component can be reused across applications.

Generally, those components are combined by introducing intermediate data structures that act as the glue, binding the components of the application together: a component constructs (produces) an intermediate data structure that is used (consumed) by other components. In a compiler setting, the abstract syntax tree is the intermediate structure that glues the compiler task components. If one increases the number of components, one gets simpler, more modular and maintainable code, whereas decreasing the number of components leads to greater runtime efficiency since fewer intermediate data structures are used. The long-sought solution for this tension between modularity and reuse on the one hand, and efficiency on the other, is to increase the amount of analysis and the number of
program transformations that the compiler is able to perform. The programmer may write programs as a set of components, confident that the compiler can fuse the components together, removing redundant intermediate data structures.

5.1 Modularity in Attribute Grammars

Attribute Grammars can be split into components in three different ways: (a) The components are the non-terminal symbols of the grammar with the associated productions and semantic rules, (b) the components are formed by the different productions associated with specific non-terminal symbols, (c) the components are the semantic domains which are used in the overall computation. For realistic AGs the first two syntactic approaches lead to huge monolithic grammars that are difficult to write and understand because related properties are described in different components, but can only be understood together. A more general and efficient form of modularity is achieved in (c), where each semantic domain is encapsulated in a single component [Wil78, Kas91a, Hen93, FMY92, LJPR93, KW94, SA98]. Traditional AG systems based on such modular descriptions start by syntactically fusing the modules and in this way constructing an equivalent monolithic AG. An implementation for such an AG is only produced afterwards. Thus, the different semantic aspects are decomposed into modules, but such a decomposition occurs only at a syntactic level. In fact, the modules are fused by the productions of the underlying AG. Consequently, each of the modules/semantic domains cannot be processed in isolation. We call this approach syntactic compositionality. This was the approach taken in the BLOCK attribute grammar: each of the semantic domains is distributed into different fragments, but these AG fragments have to be processed together.

A major disadvantage of this approach is that a single change in one component can render the entire evaluator invalid. Furthermore, it does neither support incremental analysis nor compilation of different modules: the whole implementation must be derived from scratch. Thus, the support for separate analysis and compilation of AG components, as provided by modern programming languages, is sought. Consequently, we wish to define modular attribute grammars where each component is an autonomous entity describing a particular semantic domain of an abstract language. In this way, we wish to process such components independently. So, we aim at semantic compositionality, a style of programming also known as aspect-oriented programming [Kic96, dMPJvW99].

We introduce Generic Attribute Grammars (GENAG), to provide genericity, reusability and modularity, in the context of attribute grammar based systems. A generic attribute grammar is a component which is designed to be easily reused, composed and understood. A generic attribute grammar describes a generic property of an abstract language and has the following properties:

- In a generic attribute grammar some (non-)terminal symbols, from now on called generic symbols, may not be defined within the grammar component, and thus are
considered as a parameter of the grammar. In other words, generic attribute grammars have “gaps” which are filled in later.

- From a generic attribute grammar a *generic attribute evaluator* is derived. Under our evaluation model a generic evaluator is a purely functional, data type free attribute evaluator, as defined in Chapter 4. As we will see, it is this absence of any explicit data type definitions that makes the evaluators (and the grammars) highly modular and reusable.

- In a generic attribute grammar the productions of a non-terminal symbol may be distributed over different GENAG components. Such components can be analysed and compiled separately, thus giving the sought semantic compositionality.

- A generic attribute grammar can be parameterized with the semantic functions used to compute attribute values.

- Finally, deforested attribute evaluators are used in order to remove redundant intermediate data structures which glue the different components of a GENAG system.

### 5.2 Generic Attribute Grammars

This section introduces *generic attribute grammars*. Generic attribute grammars are based on *generic context-free grammars*.

**Definition 5.1 (Generic Context-Free Grammar)** A generic context-free grammar (GCFG) is a triple \( G = (V, P, S) \). \( V = (\Sigma \cup N \cup G) \) is the vocabulary, a finite non-empty set of grammar symbols. \( \Sigma \) is the alphabet, i.e., the set of terminal symbols and \( N \) is the non-empty set of non-terminal symbols. \( G \) is a finite set of generic symbols. \( P \subset N \times V^* \) is a finite set of productions. \( S \in (N \cup G) \) is the start symbol or axiom.

A *generic symbol* is a grammar symbol for which productions may be absent from the current definition. In other words, generic symbols are parameters of the grammar. As we will explain later, a generic symbol can be the root symbol. We denote a generic symbol \( G \in G \) by \( \mathbb{G} \). We say that a generic context free grammar is **complete** if every non-terminal symbol is accessible from the start symbol and can derive a sequence of grammar symbols which contains terminal and generic symbols only. That is, it does not contain any non-terminal symbol.
Definition 5.2 (Complete Generic Context-Free Grammar) A generic context-
free grammar $G = \langle V, P, S \rangle$ is complete if and only if:

$$\forall X \in N \exists \mu, \nu \in V^* \land \delta \in (\Sigma \cup G)^* : S \Rightarrow^* \mu X \nu \Rightarrow^* \delta$$

A complete GCFG defines a generic abstract syntax tree. A generic abstract syntax tree
contains normal leaves and generic leaves, the latter being labelled with a generic symbol.

Definition 5.3 (Generic Abstract Syntax Tree) A generic abstract syntax tree of a
generic context-free grammar $G = \langle V, P, S \rangle$ is a tree defined as follows: every production
$p \in P$, $p : X \rightarrow Y_1 \ldots Y_n$ defines a tree labelled $X$ with subtrees $T_1, \ldots, T_n$ (in that
order), where

- $T_i$ is a normal leaf labelled $Y_i \in \Sigma$.
- $T_i$ is a generic leaf labelled $Y_i \in G$.
- $T_i$ is a generic abstract syntax tree $Y_i \in N$.

A generic attribute grammar is based on a generic context-free grammar that is aug-
mented with attributes, attribute equations and semantic functions.

Definition 5.4 (Generic Attribute Grammar) A generic attribute grammar is a
quadruple $GAG = \langle G, A, D \rangle$. $G = \langle V, P, S \rangle$ is a generic context-free grammar. $A$
is a finite set of attributes, partitioned into sets $A_{nont}(X)$, $A_{gen}(X)$ and $A_{loc}(p)$ for each
$X \in N$, $X \in G$ and $p \in P$. $A_{nont}(X)$ and $A_{gen}(X)$ are further partitioned into sets
$A_{inh}(X)$, $A_{syn}(X)$ and $A_{inhgen}(X)$, $A_{syngen}(X)$, respectively. $D = (T, E, F)$ is the semantic
domain of $AG$. $T$ is a finite set of types, $E$ is a finite set of semantic equations, and $F$ is
a finite set of generic semantic functions.

$A$ and $D$ are known as the generic attribution rules of $GAG$. Every attribute $a \in A$
is associated with either a grammar symbol $X \in N$, a generic symbol $X \in G$ or a
production $p \in P$. The sets $A_{nont}(X)$, $A_{inh}(X)$, and $A_{syn}(X)$ are defined in Definition 2.5.
Similarly, $A_{gen}(X)$ denotes the set of attributes associated with the generic symbol $X \in G$,
respectively. An element $a \in A_{gen}(X)$ is denoted by $X.a$, and it is either inherited if
$a \in A_{inhgen}(X)$ or synthesized if $a \in A_{syngen}(X)$. The definition of local attributes and
local attribute occurrence were presented in Definition 2.5. Thus, we omit these definitions
here. A production $p \in P$, $p : X_0 \rightarrow X_1 \cdots X_n$, with $n \geq 0$, has an attribute occurrence
$\langle p, i, a \rangle$ either if $a \in A_{nont}(X_i)$, with $X_i \in N$, or if $a \in A_{gen}(X_i)$, with $X_i \in G$. Since the
attributes and their occurrences have a type, function \( T \) assigns a type to every attribute (occurrence), exactly as in Definition 2.5.

The sets of attribute occurrences associated with production \( p \) contain the occurrences induced by generic symbols. We define the following sets of attribute occurrences:

\[
\begin{align*}
O_{\text{inh}}(\langle p, i \rangle) &= \{ \langle p, i, a \rangle \mid a \in A_{\text{inh}}(X_i) \cup A_{\text{inhgen}}(X_i) \} \\
O_{\text{syn}}(\langle p, i \rangle) &= \{ \langle p, i, a \rangle \mid a \in A_{\text{syn}}(X_i) \cup A_{\text{syngen}}(X_i) \} \\
O_{\text{nt}}(\langle p, i \rangle) &= O_{\text{inh}}(\langle p, i \rangle) \cup O_{\text{syn}}(\langle p, i \rangle)
\end{align*}
\]

The sets \( O_{\text{ntoccs}}(p) \) and \( O_{\text{pu}}(p) \) (of attribute occurrences of the non-terminal symbols of \( p \) and of \( p \) itself), and the sets \( O_{\text{inp}}(p), O_{\text{out}}(p), O_{\text{def}}(p) \) and \( O_{\text{use}}(p) \) (of input, output, defined and used attribute occurrences of production \( p \)) are overloaded to work on generic symbols, too. They are defined in Definition 2.5 and, once again, we omit their definition here.

With every production \( p \in P \) we associate a set of semantic equations, denoted by \( E_p \) of the form

\[
(\alpha_1, \ldots, \alpha_l) = f \beta_1 \beta_2 \cdots \beta_k
\]

where \( k \geq 0 \) and \( \alpha_i \in O_{\text{out}}(p) \cup O_{\text{loc}}(p) \). Note that \( \alpha_i \) and \( \beta_i \) can now be occurrences of attributes associated to generic symbols. Thus, they are used and defined in the attribute equations as normal attribute occurrences. In this equation \( f \) is either a semantic function, which is defined in the attribute grammar, or a generic semantic function \( f \in F \). A generic semantic function \( f \) is a function that is not defined as part of \( GAG \). The type of \( f \) is induced by type inferencing the attribute equations of \( GAG \) where \( f \) occurs. In other words, \( GAG \) is parameterized with \( F \).

The value of every local and output attribute occurrence of \( p \) must be defined by exactly one single semantic equation. We say that an attribute grammar \( AG = \langle G, A, E \rangle \) is a complete generic attribute grammar if and only if the next three conditions hold:

- \( G \) is complete
- \( \forall p \in P \ O_{\text{def}}(p) = O_{\text{loc}}(p) \cup O_{\text{out}}(p) \)
- \( \forall \{((\ldots, \alpha_i, \ldots, \alpha_j, \ldots) = f \ldots), ((\ldots, \alpha_k, \ldots) = g \ldots) \} \subseteq E_p \alpha_i \neq \alpha_j \neq \alpha_k \)

Note that semantic equations induce dependencies among attribute occurrences, exactly as they do within first and higher-order attribute grammars. Thus, \( DP = \bigcup_{p \in P} DP(p) \) is the relation of direct dependencies among attribute occurrences associated to productions in \( P \). Finally, \( P = G \cup F \) is the set of static parameters of the generic attribute grammar.

Completeness alone does not guarantee that all attributes of a generic abstract syntax tree are effectively computable: circular dependencies may occur. Circularities in generic attribute grammars will be discussed in Section 5.2.2. However, if circularities do not occur for any derivable tree, the generic attribute grammar is called well-defined. That is, for each generic abstract syntax tree of a GENAG, all the attribute instances are effectively computable.
Definition 5.5 (Well-defined Generic Attribute Grammar) A generic attribute grammar \( GAG = \langle G, A, D \rangle \) is well-defined, if for each generic abstract syntax tree generated by \( G \), all attribute instances are effectively computable.

The traditional definition of well-defined attribute grammars assumes that the synthesized attributes of terminal symbols are defined by an external module: the lexical analyser [Paa95]. In our definition of well-defined generic attribute grammars, we make a similar assumption, that is, the synthesized attributes of the generic symbols are defined by an external module: an external generic attribute grammar. Thus, the attribute equations defining the synthesized attributes of a generic symbol are not included in the \( \text{GENAG} \), which uses such generic symbol.

5.2.1 Generic Attribute Grammar Specifications

This section describes our notation for generic attribute grammars. We extend the AG notation with two new constructs: Symbols and Functions. The former defines the set of generic symbols occurring in the \( \text{GENAG} \). The latter defines the set of generic semantic functions used by the \( \text{GENAG} \). For each generic symbol, we also define its set of inherited and synthesized attributes.

To show this notation, we begin by presenting a very simple example. Consider a generic attribute grammar \( GAG_1 \) rooted \( R \). Suppose that this grammar has a generic symbol \( X \in G \), with \( A_{\text{gen}}(X) = \{ \text{inh1, syn1, syn2} \} \) that is partitioned into the set of inherited attributes \( A_{\text{inhgen}}(X) = \{ \text{inh1} \} \) and the set of synthesized attributes \( A_{\text{syngen}}(X) = \{ \text{syn1, syn2} \} \). We use the following notation to denote this set:

\[
\text{Symbols} = \{ X < \downarrow \text{inh1}, \uparrow \text{syn1}, \uparrow \text{syn2} > \}
\]

The type of the attributes of a generic symbol is not specified in our notation. We consider that the name of the attribute denotes its (probably) polymorphic type, and we assume that a type inference system induces the types of such attributes, as soon as the grammar is parameterized with the actual value of a generic symbol.

Let us also consider that \( GAG_1 \) has a generic function \( f \in F \). This function has to be defined in the Functions part of the grammar. The set of generic semantic functions is denoted as follows:

\[
\text{Functions} = \{ f \}
\]

Next, we present the attribution rules of this generic attribute grammar. The generic symbol \( X \) occurs in the productions and in the semantic equations of the \( GAG_1 \), as a "normal" grammar symbol. The same holds for the generic function \( f \) which is used as a "normal" semantic function, too. The (probably) polymorphic type of \( f \) is determined by type inferencing.
5.2. Generic Attribute Grammars

We have now a notation to define generic attribute grammars. We shall proceed to present a more realistic and interesting example. We introduce a simple desk calculator language DESK. DESK was presented in a recent survey on attribute grammars [Paa95] where it is analysed in great detail. A program in DESK has the following form:

\[
\text{PRINT } <\text{Expression}> \text{ WHERE } <\text{Definitions}>
\]

where \(<\text{Expression}>\) is an arithmetic expression over numbers and defined constants, and \(<\text{Definitions}>\) is a sequence of constant definitions of the form:

\[
<\text{Constant Name}> = <\text{Number}> : <\text{Type}>
\]

Each named constant used in \(<\text{Expression}>\) must be defined in \(<\text{Definitions}>\), and \(<\text{Definitions}>\) may not give multiple values for a constant. We extend the original language with types, that is, a constant has a type \text{int} or \text{real}. The dynamic meaning of a DESK program is defined implicitly as a mapping into a lower-level code. A concrete sentence in DESK and the respective generated code look as follows:

\[
\begin{align*}
\text{PRINT } & 1 + x - y & \text{LOADi } 1 \\
\text{WHERE } & x = 2 : \text{int}, & \text{ADDi } 2 \ (x) \\
& y = 3 : \text{real} & \text{SUBr } 3 \ (y) \\
& & \text{PRINT } 0 \\
& & \text{HALT } 0
\end{align*}
\]

This language introduces several typical tasks common to real programming languages, like name analysis, type checking and code generation.

The DESK compiler will be defined as a set of GENAG components that we call aspects. We start by defining the component that performs the static semantics of DESK. The code generation component will be defined in Section 5.4.2.

Let us assume that the \(<\text{Expression}>\) part of DESK inherits one attribute, the environment (\text{env}), and synthesizes three attributes: the list of identifiers used and not contained in the inherited environment (attribute \text{errs}), a typed tree for the expression (\text{tt}) and the inferred type (\text{type}). Suppose that in a library of GENAG components, we have a generic component describing the decoration of such expression trees. Therefore, we refer to the \(<\text{Expression}>\) part of DESK as a generic symbol in the following way:

\[
\text{Symbols} = \{ \text{Exp} \ <\text{env}, \uparrow \text{errs}, \uparrow \text{type}, \uparrow \text{tt} > \}
\]

and we concentrate on defining the rest of the GENAG. This grammar is presented next. The start symbol is \text{Desk}. The concrete and the abstract syntax grammar of DESK differ...
slightly. Consequently, we could use a higher-order attribute to define the mapping between both syntax trees. However, to simplify our presentation of generic attribute grammars, and to follow closely the approach taken in [Paa95], we define the attribute equations directly over the concrete grammar of desk. The literal symbols are left out from the grammar. The symbols Name, Num, Type are pseudo-terminal symbols and, once again, are assumed to be provided by an external component (i.e., the scanner).

```
Desk   <- err>
Desk = RootProd Exp D

Exp.env = D.env
Desk.errs = concat Exp.errs D.errs

D -> OneDef D
D.env = [Name], err>
D.errs = D.errs
D.env = D.env
| NoD
D.errs = nil
D.env = []

CD <- name : Name, entry : (Num, Type)
CD = Assign Name Num Type
CD.name = Name
CD.entry = (Num, Type)

Aspect 1: The specification of the name analysis task of desk.
```

The generic symbol and its attribute occurrences are used in the semantic equations of the GENAG as normal symbols and attributes. Three functions operate on the environment: the (built-in) HASKELL list constructor, the empty list constructor and the function isin which is the list membership predicate. We have omitted them since they are not relevant for the discussion, but it should be noticed that they are part of this GENAG component.

The meaning computed by the GENAG component is the attribute errs, i.e., the only synthesized attribute of the root symbol. This attribute represents the list of errors that occur in a desk program. This attribute can be a list data structure that contains the errors (and that might be used by another GENAG component for further processing) or, it can be a string representing a pretty printed list of errors that is shown to the user. In order to make it possible to (re)use this GENAG component in all these cases, we define the semantic functions used to compute attribute errs as generic functions. This generic functions are declared as follows:

```
Functions = { concat, cons, nil }
```

As we will see later these generic semantic functions can be instantiated with the (built-in) list constructor functions and the meaning of this component is a list of errors. On the
other hand, those functions can also be instantiated with pretty printing functions and the meaning of the GENAG is a string. The list constructors and the pretty printing functions must follow the polymorphic type of the generic semantic functions.

### 5.2.2 Circularities

Generic attribute grammars are executable, that is, *efficient and generic implementations* can be automatically derived from generic attribute grammars. In order to derive correct implementations, however, we have to guarantee that the generic attribute grammar is well-defined, *i.e.*, no circular dependencies are induced by the GENAG. In other words, an *order* to compute a meaning for all the abstract syntax trees assigned by the GENAG under consideration must exist. As we have explained in Section 3.4, our goal is to derive functional programs from (generic) attribute grammars that are correct and always terminate. Thus, once again, we aim at using standard algorithms to detect circularities for generic attribute grammars, thus rejecting the ones that statically induce cycles. Although losing some generality (because under lazy evaluation some static cycles may not induce dynamic cycles and non-termination), we guarantee that their implementations are correct and always terminate when executed with a sentence of the language.

Most of the algorithms that analyse attribute grammars for attribute dependencies can be used to handle generic attribute grammars as well. The key idea is to provide those algorithms with the dependencies between the inherited and the synthesized attributes of the generic symbols. Such dependencies can only be partially inferred from the GENAG since the attribute equations defining the synthesized attributes of the generic symbols are not included in the GENAG.

A straightforward strategy to approximate the dependencies of the generic symbols is to look at the evaluators of these symbols as functions that map inherited to synthesized attributes. That is, every synthesized attribute of a generic symbol depends on all the inherited attributes of that same symbol, irrespective of the context where the generic symbol occurs. Under the class of L-ordered attribute grammars we would say that the interface of every generic symbol \( G \in \mathcal{G} \) is a singleton list defined as follows:

\[
\text{Interface}(G) = [(A_{\text{inh}}(G), A_{\text{syn}}(G))]
\]

Such an approach, however, may easily introduce fake cyclic dependencies. Suppose that in a production of a GENAG, where a generic symbol \( G \) occurs, one inherited attribute \( a \) of \( G \) (transitively) depends on one synthesized attribute \( b \) of \( G \). This type of dependencies induces a cyclic dependency since attribute \( b \) directly depends on attribute \( a \) (recall that in a generic symbol, every synthesized attribute depends on all the inherited ones). In other words, multiple traversal generic symbols are not guaranteed to pass this simplistic approximation. Consider the GENAG \( AG_1 \) defined in Section 5.2.1. Figure 5.1 shows the dependency graph \( DP(\text{ProdR}) \) pasted with the functional dependencies induced by the generic symbol \( X \) and the non-terminal symbol \( S \).
A better strategy to provide the circularity test algorithm with the dependencies between inherited and synthesized attributes of the generic symbols is to explicitly specify their dependency patterns in the GENAG. That is to say that, for every generic symbol $G$ we define all the possible ways the information may flow from the inherited to the synthesized attributes of $G$. Note that the circularity test presented in Section 3.4.1 synthesizes this information in the sets $IS\_SET(X)$: a set of dependency patterns reflecting how the information flows from the inherited to the synthesized attributes for all contexts where $X$ may occur. In a classical attribute grammar, these sets are determined according to the dependencies of the attributes of $X$ induced by the attribute equations. Under the generic attribute grammar formalism no equations are defined for the attributes of generic symbols. Consequently, we simply provide the circularity test with the $IS\_SET(G)$, for every $G \in G$.

Having defined the set $IS\_SET$, for all generic symbols of the GENAG, we can use the standard circularity test to determine whether the GENAG is circular or not. Moreover, if no static circular dependencies occur, then, a well-formed lazy evaluator can straightforwardly be derived: the mapping presented in Sections 3.5 and 4.2 can be easily adapted to generate circular attribute evaluators for generic attribute grammars. These evaluators have to provide a mechanism to handle the (not-yet-defined) visit-functions of the generic symbols, i.e., their evaluators. The obvious solution is to parameterize the standard circular attribute evaluator with the evaluators that decorate the generic symbols. We will present this approach in Section 5.3 in the context of strict attribute evaluation of generic attribute grammars.

Although, defining the sets $IS\_SET$ for generic symbols explicitly makes it possible to derive correct implementations for GAGs, it restricts such implementations to the context of lazy attribute evaluation, since the dependencies induced by such a GENAG may not be partitionable. Furthermore, for large GENAG components, where generic symbols may occur in many different contexts, it can be complex to define all the possible computation patterns for each of such contexts. Another drawback of this approach is that the resulting GENAG may be difficult to read and to understand. To overcome these drawbacks we propose an approach that mimics the one taken by the class of the partitionable or L-ordered attribute grammars, i.e., we define the interface for the generic symbols. Thus,
we introduce flow types that define and fix the computational pattern for generic symbols, for all the contexts where they may occur. As a result of introducing flow types, the AG scheduling algorithms and the functional, strict attribute evaluators can be used to analyse and implement GENAG, respectively.

Next, we show in Figure 5.2 the relation between generic attribute grammars and purely functional evaluators. After that, we define flow types.

![Figure 5.2: Generic Attribute Grammars and Purely Functional Attribute Evaluators.](image)

**Flow Types**

A flow type of a generic symbol \( G \) is a finite list of pairs, which together define a computation pattern for \( G \). Each pair consists of a finite list of inherited attributes of \( G \) (the first element of the pair) and a non-empty finite list of synthesized attributes of \( G \) (the second element of the pair). Thus, a pair defines a function from the inherited to the synthesized attributes of a generic symbol.

Consider the generic symbol \( G \in G \). A flow type for \( G \) is defined as follows:

\[
[ (arg_1, res_1), \ldots, (arg_n, res_n) ]
\]

, with \( arg_1 \cup \cdots \cup arg_n = A_{inh}(G) \) and \( res_1 \cup \cdots \cup res_n = A_{syn}(G) \) and \( res_1 \cap \cdots \cap res_n = \{\} \). That is, inherited attributes may be used in different computation patterns, but a synthesized attribute is defined in exactly one. We annotate the generic symbols with the flow types as follows:

\[
G ::= (arg_1) \to (res_1) \\
\ldots \\
(\arg_n) \to (\res_n)
\]

where \( \arg_j = inh_1, \ldots, inh_k \in A_{inh}(G) \), with \( 1 \leq j \leq n \).

Let us return to the generic attribute grammar \( GAG_1 \). Observe that according to the attribution rules of production \( PROD_R \), the generic symbol \( X \) is a two traversal symbol:
the two traversals are due to the dependency from a synthesized \((\text{syn}_2)\) to an inherited attribute \((\text{inh}_1)\) of \(X\). Accordingly, we need a flow type which specifies a two traversal computation pattern. The first traversal synthesizes attribute \(\text{syn}_2\), and the second one uses attribute \(\text{inh}_1\). A possible flow type looks as follows:

\[
\text{Symbols} = \{ \ X \ :: \ (\rightarrow \text{syn}_2, \text{inh}_1 \rightarrow \text{syn}_1) \ \}
\]

This flow type corresponds to the following interface of \(X\):

\[
\text{Interface}(X) = [\{\}, \{\text{syn}_1\}], ([\text{inh}_1], \{\text{syn}_2\})]
\]

Figure 5.3 shows the dependency graph for production \(\text{ProdR}\) induced by both the attribute equations associated with this production and by the above flow type.

![Figure 5.3: The dependency graph \(DP(\text{ProdR})\) induced by the flow type of \(X\).]

Let us return to the desk calculator example. The flow type we define for the generic symbol \(\text{Exp}\) looks as follows:

\[
\text{Symbols} = \{ \ \text{Exp} \ :: \ (\text{env} \rightarrow (\text{errs}, \text{type}, \text{tt})) \ \}
\]

Observe that this flow type does not introduce any circularity in the \text{GENAG} component \text{Aspect}1.

### 5.3 Generic Attribute Evaluators

This section presents \textit{Generic Attribute Evaluators (GAE)}, which are an efficient implementation for \textit{generic attribute grammars}. The generic attribute evaluators are based on strict and deforested functional attribute evaluators presented in Chapter 4.

This mapping handles the generic symbols and semantic functions in the “standard” way: they are extra parameters of the \(\lambda\)-attribute evaluators. That is, for every generic symbol, the attribute evaluator receives an extra argument corresponding to the evaluator of that symbol. The generic semantic functions are also handled as extra arguments to the generic evaluator. In other words, the \(\lambda\)-attribute evaluator derived for a generic attribute grammar \(GAG\) is parameterized with the \textit{static parameters} \(P\) of \(GAG\).
This technique is orthogonal to the strict and lazy implementations of generic attribute grammars. The examples of generic attribute evaluators presented in this section are based on the strict implementation only.

**Generic Symbols**

The generic symbols are handled as normal non-terminal symbols: the visit-functions derived for the productions where a generic symbol occurs receive as an extra argument the generic evaluator which decorates the generic symbol. This is just as normal grammar symbols are handled. The generic attribute evaluators, however, refer to the visit-functions which decorate the generic symbols as parameters of the GAE.

In order to derive strict λ-attribute evaluators for generic attribute grammars the attribute evaluation order must first be computed statically. The flow types are used to provide the scheduling algorithm with the evaluation order of the generic symbols.

Consider the generic attribute grammar $GAG_1$ and the flow type of $X$ as defined in Section 5.2.2. According to the partitions induced by the flow type and the dependencies defined in both productions of $GAG_1$ (see Figure 5.3) the visit-sequences produced by the the scheduling algorithm are:

```
plan ProdR
begin 1  inh()
  visit (X, 1)
  eval (S.inh1)
  visit (S, 1)
  eval (X.inh1)
  visit (X, 2)
  eval (R.syn1)
end 1  syn(R.syn1)
```

```
plan ProdS
begin 1  inh(S.inh1)
  eval (S.syn1)
end 1  syn(S.syn1)
```

The visit-sequences can be directly implemented as λ-attribute evaluators using the techniques described in Chapter 4. For the sake of completeness, the resulting deforested visit-functions are shown next. The generic semantic function $f$ is considered in this example as a normal semantic function, i.e., it is part of the evaluator. Generic functions are discussed in the next section.

\[
\lambda_{ProdR} = \begin{cases} 
\lambda_X(X, 1) & \lambda_S = syn1 \\
\lambda_X(X, syn2) & \lambda_S = syn1 \\
\lambda_X(X, syn2) & \lambda_S = syn1 \\
\end{cases}
\]

\[
\lambda_{ProdS} = \begin{cases} 
tN.inh1 = syn1 \\
syn1 = f(tN.inh1) \\
\end{cases}
\]

As expected, the evaluator of the generic symbol $X$ is one argument of the visit-function $\lambda_{ProdR}$. Observe that this evaluator represents the first visit to the generic symbol only. When such a function is applied to the inherited attributes of the first visit (an empty set, in this case), it returns the visit-function to the next visit: exactly as “normal” deforested
visit-functions of $\lambda$-attribute evaluators.

Consider the productions $\text{RootProd}$ and $\text{Assign}$ presented in Aspect 1 of desk. To simplify the discussion, let us assume that function $\text{concat}$ is part of this GENAG component. According to the attribute equations associated to these productions and the flow type defined for symbol $\text{Exp}$, the following two deforested visit-functions are generated:

\[
\begin{align*}
\lambda_{\text{RootProd}} & : \quad \lambda_{\text{Exp}} \lambda_{\text{Def}} = (\text{errs}) \\
\text{where} & \quad (\text{env}_3, \text{errs}_3) = \lambda_{\text{Def}} \\
& \quad (\text{errs}_2, \text{type}_2, \text{tt}_2) = \lambda_{\text{Exp}} \text{env}_3 \\
& \quad (\text{errs}) = \text{concat} \quad \text{errs}_2 \quad \text{errs}_3 \\
\lambda_{\text{Assign}} & : \quad t\text{Name} \ t\text{Num} \ t\text{Type} = (\text{name}, \text{entry}) \\
\text{where} & \quad \text{name} = t\text{Name} \\
& \quad \text{entry} = (t\text{Num}, t\text{Type})
\end{align*}
\]

where $\lambda_{\text{Exp}}$ is the reference to the partial parameterized visit-function that is obtained when parsing the expression part of a DESK program. In the body of the visit-function $\lambda_{\text{RootProd}}$, we can see that the scheduling algorithm inferred the right order of evaluation, i.e., the right-to-left pass of the attribute evaluator: first, the function that represents the evaluation of the $<\text{Definitions}>$ part of DESK is called (and it returns the environment). After that, the function which represents the evaluation of the $<\text{Expression}>$ part is computed (and it uses the computed environment). Note also that the deforested visit-function $\lambda_{\text{Def}}$ has all the parameters it needs at parse-time (no parameters in this case). Thus, when the evaluator is executed, the calls to that function are totally parameterized and evaluated at parse-time.

The deforested visit-functions have the following types:

\[
\begin{align*}
type \text{Exp} & = [\text{Name}] \to (\text{errs}, \text{type}, \text{tt}) \\
type \text{Defs} & = ([\text{Name}], \text{errs}) \\
\lambda_{\text{RootProd}} & : \quad \text{Exp} \to \text{Defs} \to \text{errs} \\
\lambda_{\text{Assign}} & : \quad \text{Name} \to \text{Num} \to \text{Type} \to (\text{Name}, (\text{Num}, \text{Type}))
\end{align*}
\]

The type variables $\text{errs}$, $\text{type}$ and $\text{tt}$ denote the polymorphic types of the attributes. Note that, the previous visit-functions do not rely on any particular type definition. As a result, they can be re-used in different contexts, provided that the flow of data is the one specified by such functions. For example, both in this evaluator and in the GENAG, the types $\text{Name}$, $\text{Num}$ and $\text{Type}$ are provided by an external lexical analyser. The identifiers of DESK language could equally well be a single character, a string, or even a numeral. The GENAG and the evaluator can be reused in all those cases.

**Generic Semantic Functions**

The generic semantic functions are handled as extra arguments of the evaluators. We consider two possibilities: the generic semantic functions are normal inherited attributes, or the generic functions are syntactic referenced symbols. In the next two sections we describe both approaches.
Generic Semantic Functions as Inherited Attributes

Generic semantic symbols are handled as normal inherited attributes of the root symbols. They are passed down in the tree, as normal attributes, to the nodes where they are used. The standard scheduling algorithms can be used to schedule their propagation in the tree, since those functions have become normal attributes. In this section, we show how to transform a GENAG which contains generic functions into an equivalent one without generic functions. As a result of this transformation, we can use the techniques described thus far to derive generic attribute evaluators.

A generic attribute grammar with generic functions can be transformed into an equivalent grammar with no generic functions as follows: let $GAG = \langle G, A, D \rangle$ be a generic attribute grammar and $R$ be the axiom of the generic context-free grammar $GAG$.

1. For every generic semantic function $f \in F$ we add an inherited attribute $f$ to $A_{inh}(R)$.

2. Every occurrence of $f$ in $E$ is replaced by $\uparrow R.f$.

Let us consider the generic attribute grammar $GAG_1$. Using the above transformation, we obtain the following equivalent attribute grammar:

$$
\begin{align*}
R & \quad \langle \downarrow f \rangle \uparrow syn1 > \\
R & \rightarrow PRODR (X S) \\
X_{inh1} & = S.syn1 \\
S_{inh1} & = X.syn2 \\
R.syn1 & = X.syn1
\end{align*}
$$

and the derived $\lambda$-attribute evaluator looks as follows:

$$
\begin{align*}
\lambda_{ProdR} & \quad \lambda X, _1 f = syn1 \\
\lambda_{ProdS} & \quad \lambda tN inh1 f = syn1 \\
\lambda_{Syn} & \quad \begin{cases}
\lambda_{X1, syn2} = \lambda X2, syn1 \\
\lambda_{X2, syn1} = \lambda X1, syn2
\end{cases}
\end{align*}
$$

Under this straightforward approach, however, a problem arises when productions applied to the same non-terminal symbol are distributed into different grammar components. In this case, two productions that use different semantic functions and that were analysed separately may induce different inherited attributes independently of each other. Consequently, the two deforested visit-functions (independently) derived for such productions will get different arguments and, obviously, will have different types, thus causing a type error when the deforested evaluator is constructed.

Generic Semantic Functions as Syntactic Symbols

A better strategy to handle generic functions is to consider the generic function as the first (static) argument of the visit-functions where such function is used. In other
words, generic symbols are handled as *syntactic symbols*. In this case, each visit-function defining the first visit to a particular constructor receives as first argument the set of generic functions used in the respective production. The generic semantic functions are passed to the visits where they are applied, as results/arguments of the visit-functions which perform the different visits to that particular constructor. This is exactly as *syntactic references* are handled by $\lambda$-attribute evaluators. In this section we present the attribute evaluators obtained with this technique.

Consider the GENAG $GAG_1$ again. The generic attribute evaluator obtained with this technique is:

$$
\lambda_{ProdR} \frac{}{X, S} = \text{syn}_1
$$

where

$$
(\lambda X_2, \text{syn}_2) = \frac{X_1}{X_1} = \frac{}{S, \text{syn}_1}
$$

and

$$
\text{syn}_1 = \frac{tN}{\text{inh}_1} \text{syn}_1
$$

The generic functions are now the first arguments of the visit-functions. This approach has an important advantage when compared to the previous one: the visit-functions can be partially evaluated [JGS93] with respect to generic semantic functions. Let us recall that generic functions are the static parameters of the GENAG. That is, when we parameterize the generic function of the GENAG with a particular function (that is a static parameter of the GENAG) we also partially evaluate the respective evaluator, as far as this function is concerned. As a result of the partial evaluation, the evaluator becomes a “standard” $\lambda$-attribute evaluator: all the semantic functions are part of the evaluator and they are not arguments of the visit-functions anymore.

Let us consider the DESK calculator again and present a GENAG component which defines the processing of expression trees. The root symbol is $Exp$.

<table>
<thead>
<tr>
<th>Functions</th>
<th>=</th>
<th>{ add, fac, concat, cons, nil }</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Exp$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$Exp, env$</td>
<td>$\rightarrow$</td>
<td>$Exp$</td>
</tr>
<tr>
<td>$Fac, env$</td>
<td>$\rightarrow$</td>
<td>$Fac$</td>
</tr>
<tr>
<td>$Exp, errs$</td>
<td>$\rightarrow$</td>
<td>$Conc$</td>
</tr>
<tr>
<td>$Exp, type$</td>
<td>$\rightarrow$</td>
<td>$Inftype$</td>
</tr>
<tr>
<td>$Exp, tt$</td>
<td>$\rightarrow$</td>
<td>$Add$</td>
</tr>
<tr>
<td>$Fac, val$</td>
<td>$\rightarrow$</td>
<td>$Name$</td>
</tr>
<tr>
<td>$Exp, env$</td>
<td>=</td>
<td>$Exp, env$</td>
</tr>
<tr>
<td>$Fac, env$</td>
<td>=</td>
<td>$Exp, env$</td>
</tr>
<tr>
<td>$Exp, errs$</td>
<td>=</td>
<td>$Conc$</td>
</tr>
<tr>
<td>$Exp, type$</td>
<td>=</td>
<td>$Inftype$</td>
</tr>
<tr>
<td>$Exp, tt$</td>
<td>=</td>
<td>$Add$</td>
</tr>
<tr>
<td>$Fac, val$</td>
<td>=</td>
<td>$Getval$</td>
</tr>
<tr>
<td>$Fac, type$</td>
<td>=</td>
<td>$Gettype$</td>
</tr>
<tr>
<td>$Exp, errs$</td>
<td>=</td>
<td>$Nil$</td>
</tr>
<tr>
<td>$Fac, errs$</td>
<td>=</td>
<td>$Foreach$</td>
</tr>
</tbody>
</table>

| $Exp$ | $\rightarrow$ | $\rightarrow$ |
| $Exp, env$ | $\rightarrow$ | $Exp, env$ |
| $Fac, env$ | $\rightarrow$ | $Fac, env$ |
| $Exp, errs$ | $\rightarrow$ | $Fac$ |
| $Exp, type$ | $\rightarrow$ | $Fac, type$ |
| $Exp, tt$ | $\rightarrow$ | $Fac, val$ |

Aspect 2: The type checking for expressions.

Both the semantic functions used to compute the values of the synthesized attributes $errs$ and the typed tree $tt$ are defined as generic functions. The semantic functions $Getval$, $Gettype$, $Forall$, and $Foreach$ are used to compute the values of the synthesized attributes $errs$ and the typed tree $tt$. The functions $Getval$, $Gettype$, $Forall$, and $Foreach$ are used to compute the values of the synthesized attributes $errs$ and the typed tree $tt$.
gettype, inftype and isin are normal (static) semantic functions that are part of this GENAG. To simplify the discussion we have omitted their definition from this component. The single deforested visit-function derived for the production ADD is presented next:

\[
\lambda_{Add} \left[ \begin{array}{c}
\text{concat} \\
\text{add}
\end{array} \right] \lambda_{Exp} \lambda_{Fac} \quad \text{env} = (errs, type, tt)
\]

where

\[
(\text{errs}_3, \text{type}_3, \text{val}_3) = \lambda_{Fac} \quad \text{env}
\]
\[
(\text{errs}_2, \text{type}_2, \text{tt}_2) = \lambda_{Exp} \quad \text{env}
\]
\[
\text{errs} = \text{concat} \quad \text{errs}_2 \quad \text{errs}_3
\]
\[
\text{type} = \text{inftype} \quad \text{type}_2 \quad \text{type}_3
\]
\[
\text{tt} = \text{add} \quad \text{val}_3 \quad \text{type} \quad \text{tt}_2
\]

The first arguments of the visit-function are now the references to the generic functions applied on the production ADD. Next, we show the type definition for this deforested visit-function. This function and its type will be used in the next section to explain the composition of GENAG components.

\[
type \quad Exp = env \to (errs, type, tt)
\]
\[
type \quad Fac = env \to (errs, type, tt)
\]
\[
\lambda_{Add} :: (errs \to errs \to errs) \to (val \to type \to tt \to tt) \to Exp \to Fac \to (errs, type, tt)
\]

As we can see in this type definition, the first four type sub-expressions represent the polymorphic types of four functions: the two generic functions and the two initial visit-functions derived for the right-hand side non-terminal symbols of ADD.

### 5.4 Semantic Compositionality

This section discusses the semantic compositionality of generic attribute grammars.

#### 5.4.1 Extending the Productions of a Non-terminal

The generic attribute evaluators are independent of the abstract tree data type. This data type freeness allows the GENAG and the GAE to be easily extended with new productions. Suppose we want to extend a generic attribute grammar with new productions. In a traditional AG implementation, the existing attribute evaluator would have to be modified, since the type of the abstract syntax tree changes. As a result, the attribute evaluator for the complete AG would have to be produced again.

In our implementation, however, we can define a new GENAG component where the new production and its semantic equations are specified. This component can be analysed separately and the deforested visit-function(s) for the production can be produced independently. No "global" tree data type has to be modified. To compute the order of the attribute evaluation we simply have to provide the scheduling algorithm with the flow type of the non-terminal symbol on the left-hand side of the production.

Let us return to the DESK example. Suppose that we want to extend the productions of the expression GENAG in order to allow the operation ◇, which is not yet supported
by the reused component. Using generic attribute grammars we define this extension in a
new component. The non-terminal symbol in the left-hand side of the grammar is defined
as a generic symbol and its flow type is specified. The new production and its attribute
equations are defined as follows:

Symbols = \{ \text{Exp} :: (\text{env} \to (\text{errs, type, tt})) \}
Functions = \{ \text{dia, concat} \}
\text{Exp} \quad \langle \downarrow \text{env}, \uparrow \text{errs}, \uparrow \text{type}, \uparrow \text{tt} \rangle
\text{Exp} \to \text{DIA} \ (\text{Exp} \diamond \text{Fac})
\text{Exp}_{\text{env}} = \text{Exp.env}
\text{Fac}_{\text{env}} = \text{Exp.env}
\text{Exp}_{\text{errs}} = \text{concat Exp}_{\text{errs}} \text{Fac}_{\text{errs}}
\text{Exp}_{\text{type}} = \text{inftype Exp}_{\text{type}} \text{Fac}_{\text{type}}
\text{Exp}_{\text{tt}} = \text{dia Fac.val Exp.type Exp}_{\text{tt}}

Aspect 3: Extending the productions of the language.

Observe that the start symbol of this GENAG is the generic symbol \text{Exp}. In this exam-
pole, we use this grammar component with the single purpose of extending its productions.
From this GENAG component, a deforested visit-function is generated. The flow type of
\text{Exp} is the interface of non-terminal \text{X} provided to the scheduling algorithm by this com-
ponent. Thus, the header of the function derived for this component is:

$$\lambda_{\text{DIA}} \quad \text{concat} \quad \text{dia} \quad \lambda_{\text{Exp}} \quad \lambda_{\text{Fac}} \quad \text{env} = (\text{errs, type, tt})$$

Now suppose that the generic function \text{dia} is handled as an inherited attribute (Sec-
tion 5.3) as we suggested in the first technique to handle generic semantic functions we
have presented. In this case, a new inherited attribute \text{dia} is added to the non-terminal
symbol \text{Exp}. As a result, all the visit-functions derived for the productions applied on \text{Exp}
have to be extended with an extra argument, too. If we consider the separate compilation
of the modules, however, we get a type error, since the visit-functions derived for those
productions have different types!

5.4.2 Composing GENAG Components

The generic attribute grammars are efficiently and easily composed. The generic func-
tions of one GENAG can be instantiated with the visit-functions of the evaluator derived
from another GENAG. Because this evaluator is totally deforested, no intermediate data
structure is explicitly constructed nor traversed. Furthermore, we can give different se-
manitics to a GENAG by instantiating its static parameters (i.e., the generic functions and
the generic symbols) with different arguments.

Consider the code generation task of the desk calculator. The code generation uses
type information computed by the type checker. Type checking is defined in the generic
attribute grammar defined in *Aspect* [2]. Thus, we wish to define a new aspect of our *Desk* language, *i.e.*, the code generation, that can be analysed and compiled independently from the type check aspect. Moreover, we wish to combine both aspects easily and efficiently.

Before we define the code generation aspect, let us recall that the GENAG component for expressions was designed to be easily reusabled. Thus, one of the attributes synthesized by such a generic component is the type information, *i.e.*, attribute $tt$, that is collected during the decoration of expression trees. Furthermore, to make it possible to parameterize the component with the functions that handle the type information, the functions that construct values for attribute $tt$ are generic semantic functions. As a result we can, for example, parameterize the expression GENAG with a tree algebra and explicitly construct a typed tree. This tree can be defined by the following algebraic data type:

\[
\text{data } ETree = \text{ConsTree } \text{Num } \text{Type } ETree \\
\quad \quad \text{FacTree } \text{Num } \text{Type}
\]

where $\text{Num}$ and $\text{Type}$ are the types of the correspondent pseudo-terminal symbols of the GENAG. The constructors of the tree data type must follow the polymorphic types inferred from the attribute equations of the generic grammar. Thus, we can instantiate the generic functions with these constructor functions as follows:

\[
\begin{align*}
\text{add} &= \text{ConsTree} \\
\text{fac} &= \text{FacTree}
\end{align*}
\]

In this case, the evaluation of attribute $tt$ actually constructs a typed tree. In a traditional AG implementation this tree would be traversed later on, in order to generate the assembly code. Thus, this tree would be the intermediate structure which glues the GENAG components.

Let us now present one GENAG component that describes the code generation task for the *Desk* language.

```
Exp <↑ code : [String] >
Exp → \text{CODEADD} (\text{Num } \text{Type } \text{Exp})
\quad \text{Exp.code} = \begin{cases} 
\text{if isinttype type then } \text{Exp2.code} ++ [\text{\'ADDi\'} ++ \text{num}] \\
\text{else } \text{Exp2.code} ++ [\text{\'ADDr\'} ++ \text{num}]
\end{cases}
\mid \text{CODEFACT} (\text{Num } \text{Type})
\quad \text{Exp.code} = \begin{cases} 
\text{if isinttype Type then } [\text{\'LOADi\'} ++ \text{Num}] \\
\text{else } [\text{\'LOADr\'} ++ \text{Num}]
\end{cases}
```

*Aspect 4: The code generation.*

and derive the deforested generic attribute evaluator. Next, we show the deforested visit-function derived for the **CODEADD** production:

\[
\lambda_{\text{Code.Add}}: \quad \text{tNum } \text{tType } \lambda_{\text{Exp2}} = \text{code}
\]

where

\[
\begin{align*}
\text{code} &= \text{if isinttype } \text{tType1} \text{then } \text{code2} ++ [\text{\'ADDi\'} ++ \text{tNum}] \\
&\quad \text{else } \text{code2} ++ [\text{\'ADDr\'} ++ \text{tNum}]
\end{align*}
\]
Using the deforested generic attribute evaluators, we can instantiate the generic functions with the deforested visit-functions derived for the code generation GENAG. We compose the GENAG components as follows:

\[
\begin{align*}
add &= \lambda_{\text{CodeAdd}}^4 \\
fac &= \lambda_{\text{CodeFac}}^3
\end{align*}
\]

Observe that these visit-functions get all the arguments they need during the decoration of the tree that defines the DESK expressions. As a result, the code generation is actually computed during the evaluation of these trees. Consequently, the visit-function that decorates the expression part of the DESK language returns, as one of its results, a list with the assembly code. The intermediate typed tree is not constructed nor traversed. Observe also that the list (with the generated code) can also be eliminated if we define the pretty printing of the assembly code in an attribute grammar setting and derive the corresponding deforested visit-functions. In this case, the list is eliminated and the previous function returns a string: the assembly code.

### 5.4.3 Inter-Module Attribute Dependencies

The main problem in a purely functional implementation of attribute grammars is to handle attribute instances that are computed during one traversal of the evaluator, and are used in a future traversal. A similar problem occurs with the separate compilation of GENAG components: how do we pass attribute values that are computed in one GENAG component and that will be used in a different component? We use the approach taken for the deforested evaluators: those values are passed between components as results/arguments of the visit-functions derived for the GENAG components. No additional data structures are required.

As the reader may have noticed, we have used this approach to pass the types inferred in the type checking aspect of DESK into the code generation component. Let us be more precise: the attribute occurrence \( \text{Exp}.\text{type} \) is a “hidden” result of the type checking GENAG component. It is an argument of the generic semantic function \( add \) of Aspect \( 2 \) which defines the synthesized attribute \( tt \). The type of the expressions is an argument of the code generation aspect, since it is an argument of the constructor \( \text{CodeAdd} \). Consequently, when we compose both aspects, we are, in fact, passing the attribute \( type \) as a hidden result/argument of the evaluators derived from such components. This is exactly as the attribute values induced by inter-traversal attribute dependencies are handled by deforested evaluators.

In this chapter, we have discussed the strict, functional implementation of generic attribute grammars. Such an implementation of attribute grammars is attractive for two main reasons: firstly, efficient and correct implementations can be derived using well-known attribute grammar techniques. Secondly, an efficient and elegant incremental attribute evaluator can be obtained through standard function memoization techniques. This is the subject of the next chapter.
Chapter 6

Functional Incremental Attribute Evaluation

Summary

This chapter presents functional and incremental attribute evaluation. The change propagation and the visit-function memoization approach are described. The incremental behaviour of the binding-tree and visit-tree based attribute evaluators is analysed and compared. Furthermore, a new technique for the incremental evaluation of deforested attribute evaluators is presented. A criterion to define the set of memoized visit-functions is presented and techniques to manage the function cache are described.

Attribute Grammars are the underlying formal basis of a number of language-based environments and environment generators [RT89, TC90, KS98]. Such environments repeatedly apply a software tool to a sequence of similar inputs. Examples include compilers, interpreters, type inference engines, theorem provers and text processors whose series of inputs usually are incrementally modified text files. In such environments, an abstract syntax tree is used to represent the object being edited. The attribute instances are kept up to date, as the underlying tree is modified either by direct manipulation or by indirect transformations. The attributes can provide immediate feedback to guide the user through the editing process, and they can also provide immediate incremental translation.

One of the key features to handle interactive environments is the ability to perform efficient re-computations after each interaction with the user. Typically, each user interaction slightly modifies the input, and, as a result, most of the computations induced by this interaction are actually repeating computations performed previously. Consequently, an efficient algorithm for such interactive environment aims at reusing the unaffected computations and at performing only those computations which are really affected by the interaction. That is to say that, interactive environments rely heavily on efficient incremental algorithms.
In this chapter we discuss efficient techniques for the incremental evaluation of higher-order attribute grammars based on our purely functional attribute evaluators. Under a purely and strict functional setting, efficient incremental attribute evaluation can be easily achieved by memoizing calls to the functions of the evaluator, the so-called memoized functions. A memoized function remembers the arguments to which it has been applied, together with its results. When a memoized function is again applied to the same arguments, incremental evaluation is achieved by returning the memoized result, rather than re-evaluating it from scratch. In this chapter we show how efficient incremental evaluation for the binding-tree and visit-tree approach can be obtained. A new efficient approach for incremental evaluation based on the memoization of deforested attribute evaluators is equally presented.

6.1 Incremental Attribute Evaluation

Attribute grammars have been used with great success in the development of language-based tools since Thomas Reps first used attribute grammars to model syntax-oriented editors [Rep82]. In such an interactive environment, a user slightly modifies a decorated tree $T$ into $T'$. After that, an incremental attribute evaluator uses $T$ and its attributes instances to compute the attributes instances of $T'$, instead of decorating $T'$ from scratch. The underlying assumption is that the decoration of $T'$ from scratch is more expensive (i.e., more time consuming) than an incremental update of $T$.

Once the tree $T'$ has been created, an incremental evaluator is applied to compute correct values for all attribute instances in $T'$, which are new or have a changed value. Generally, the attribute instances in $T'$ are classified as retained and newborn. The former contains the attribute instances that are in $T$, and the latter contains all the other instances, i.e., instances that never existed previously. The attribute instances in retained are further classified as equal, meaning instances whose values do not change due to the tree modification, or as changed, in case the values change due to the modification. The set of attribute instances that require a recomputation is referred to as $\Delta$, and it is defined as $\Delta = \text{newborn} \cup \text{changed}$. It is commonly known as affected.

Although any non-incremental attribute evaluator (thereafter also referred to as exhaustive attribute evaluator) can be applied to completely re-decorate tree $T'$, the goal of an optimal incremental attribute evaluator is to limit the amount of work to $O(|\Delta|)$. The assumption is that all semantic functions have the same unit cost. In reality, such functions can be more expensive as it is illustrated by the lookup semantic functions \(mBin\) and \(mNBin\) of the block AG.

The traditional approach to achieve incremental evaluation involves propagating changes of attribute values through the attributed tree [RTD83, YK88, RT89]. The basic idea is the following: attribute instances in newborn are marked inconsistent. Next, a change propagator algorithm selects a marked attribute instance that has no marked predecessors and computes its value. If the new value differs from the old one, all the successors of
that attribute instance are marked. When propagation has died out, all instances will be consistent again, i.e., they have correct values.

To obtain optimal incrementality, the change propagator algorithm has to guarantee that every affected instance is re-evaluated after all the arguments of its defining equation have gotten their correct final value. So, the dependencies between attributes must be known. Reps presented an algorithm that dynamically constructs a dependency graph that is topologically sorted on the fly \cite{Rep82, RTD83}. It is optimal, in the sense that only the affected instances are re-evaluated, but its space consumption is considerable \cite{RT89}. An evaluator based on the visit-sequence paradigm implicitly establishes an order on the attributes instances of the syntax tree. An incremental evaluator can naturally make use of this order \cite{Yeh83, YK88, RT89}. Both algorithms need to locate the starting affected attribute instance and to administrate which attribute instances are affected by a tree transformation. A different approach is proposed in the OPTRAN system \cite{LMOW88}. It combines data driven and demand driven algorithms for the reevaluation of subsets of attributes. The data driven is a change propagator algorithm. The demand driven algorithm is able to delay reevaluation of attributes, until their values are needed. In other words, it is a lazy attribute evaluator.

Change propagating algorithms were developed to efficiently process first-order attribute grammars. When considering the incremental evaluation of higher-order attribute grammars different questions arise. For example, how can we determine which parts of the modified tree reappear in the new one? In the first-order approach it is clear what parts of the tree remain in the new one, since the tree is fixed during decoration. That is not so clear in the higher-order case, since the (higher-order) trees are constructed dynamically. Another question to be answered is, how can we efficiently compare attribute instances that are now higher-order trees? Obviously, such trees can be deeply structured values, which are expensive to compare by standard algorithms. Moreover, different instances of higher-order attributes may be decorated (completely or partially) with the same attribute values. Thus, how can we avoid the repeated decoration of similar attributed trees that inherit the same attribute values? The answer to all these questions is to use purely functional attribute evaluators and to use function memoization. This approach is described in detail in the next section. After that, the incremental evaluation of HAG is discussed.

#### 6.2 Visi-t-Function Memoization

The visit-function memoization proposed by Pennings \cite{Pen94} is based on the following combination of ideas:

**Purely functional attribute evaluators:** Syntax trees are decorated by binding-tree based attribute evaluators. Thus, attribute instances are not stored in the tree nodes, but, instead, they are the arguments and the results of pure (side-effect free) functions: the visit-functions.
**Data constructor memoization:** Since attribute instances are not stored in the syntax tree, multiple instances of the syntax tree can be shared. That is, trees are collapsed into minimal *direct acyclic graphs (DAG)*. DAGs are obtained by constructing trees bottom-up and by using constructor memoization to eliminate replication of common sub-expressions. This technique, also called *hash-consing* [Hug85], guarantees that two identical objects share the same records on the heap, and thus are represented by the same pointer.

The basic idea of hash-consing is very simple: whenever a new node is allocated (or "consed") we check whether there exists already an identical record in the heap. If so, we avoid the allocation and simply use the existing one. Otherwise, we perform the allocation as usual. Generally, a *hash table* is used to search the heap for a duplicated record. Hash-consing can be applied to pure values only, *i.e.*, values that never change during the execution of a program: for if two updatable records have identical values now, they might not be identical later, and so merging them could lead to incorrect values. Observe that this is the case if attributes are stored in the tree nodes, because shared nodes may have to store different attribute values induced by different, but shared, attribute instances (and most probably will).

Let us return to the block AG. Figure [6.1](#figure6.1) shows the collapsed abstract syntax tree obtained when decorating the concrete sentence: \([\text{decl } x ; \text{use } y ; \text{decl } y ; \text{use } y]\). The nodes of the DAG are labelled with numbers for future references.

![Figure 6.1: The DAG representation of the abstract syntax tree assigned to the sentence [\text{decl } x ; \text{use } y ; \text{decl } y ; \text{use } y]. This figure is a graphical representation of the term cache.](#figure6.1)

Attribute values may be large structures (*e.g.*, symbol tables, higher-order attributes, etc). Therefore, the constructors for user defined types are also shared. This technique solves the problem of expensive attribute equality test during evaluation and it also settles the problem of huge memory consumption due to multiple instances of the same attribute value in a syntax tree.

This technique has three main advantages: firstly, it considerably reduces the memory usage; secondly, it allows for efficient equality tests between all terms because a pointer
comparison suffices; thirdly, as we will explain next, it makes efficient visit-function memoization possible.

**Visit-function memoization:** Due to the pure nature of the visit-functions, incremental evaluation can now be obtained by memoizing calls to visit-functions. The binding-tree based attribute evaluators are constructed as a set of strict functions. Thus, standard function memoization techniques can be used to memoize their calls [PSV92]. Memoization is obtained by storing in a memo table calls to visit-functions. Every call corresponds to a memo entry in the memo table, that records both the arguments and the results of one call to a visit-function.

The essence of the visit-function memoization is as follows: each time a memoized visit-function, say \( \text{visit}_1X \), is applied to a subtree and to a set of remaining arguments (i.e., values of attribute instances), we search a memo table to check whether that function was previously applied to those arguments, or not. If the memo table contains an entry corresponding to the call, the result in that entry is returned. If no such entry exists, the visit-function is applied to the arguments and the call is memoized.

A memoized function is defined by annotating its application with the keyword `memo`. For example, the memoized version of visit-function \( \text{visit}_1P \) of the binding-tree based evaluator \( \text{Eval}^2 \) looks as follows:\[
\text{visit}_1P \ (R \ tIts) = \text{errs}_1 \\
\text{where} \ \
lev_1 = 0 \\
dcli_1 = \left[ \right] \\
(tIts_1^{1-2}, dclo_1) = \text{memo} \ \text{visit}_1\text{tIts} \ tIts \ dcli_1 \ lev_1 \\
errs_1 = \text{memo} \ \text{visit}_2\text{tIts} \ tIts \ dclo_1 \ tIts_2^{1-2}
\]

Efficient incremental evaluation relies heavily on efficient equality test of terms. Observe that the syntax tree itself is one of the parameters for the visit-functions, and, consequently, must be checked for equality, too. The same holds for the remaining arguments that are tree structures, e.g., instances of higher-order attributes (i.e., higher-order trees) and binding-trees. These arguments must also be checked for equality. Thus, an efficient equality test between terms is essential for an efficient visit-function memoization scheme. This efficiency is provided by the hash-consing technique: the syntax trees, the binding-trees and the higher-order trees are “hash-consed” in the term cache. They are stored as DAGs and thus they are efficiently tested for equality.

For example, when memoizing the function application \( \text{visit}_1X \ T \ args \) under a hash-consing scheme, the values of \( T \) and the remaining arguments \( args \) have already been hash consed to unique addresses. So, it is possible to perform function memoization with very little overhead: an efficient search operation in the hash table can be performed to look up

---

1 Actually, the memo annotation that was used corresponds to a primitive function of the gofer system [MP91] that was extended with a memoization mechanism [vD92]. We have used this system to prototype the incremental evaluation of our purely functional evaluators [SSKP96]. Memoized “gofer” visit-functions are automatically derived from HAGs.
for the arguments. Note that without the constructor memoization scheme, searching for
the call \texttt{visit\_X T args} in the memo table is nontrivial, specially because the syntax tree \texttt{T}
(and possibly other arguments) is a deeply structured value. The use of a standard equality
test between terms that traverse the trees will drastically decrease the performance of the
memoization scheme.

Although a distinction is generally made between constructor and visit-function (\textit{i.e.},
non-constructor) memoization, they are much alike: both techniques memoize function
calls. The former, at the data constructor domain, and the latter, at the function domain. It
should be noted, however, that there are also some differences which may lead to dif-
ferent implementations of the two. Non-constructor functions are memoized in order to
increase the performance of the program, taking advantage of reusing memoized results.
On the contrary, constructor functions are memoized with the aim of decreasing memory
consumption and, specially, increasing the performance of the equality test between terms.
The time to evaluate a constructor function is expected to be smaller than the overhead
due to the memoized evaluation. A more important difference is that the techniques used
to purge (\textit{i.e.}, to discard) constructor and non-constructor function calls from the cache
are also different. For example, one can always remove an arbitrary memoized function
call without influencing the correctness of the evaluator. This is not the case for memoized
constructor functions. For this reason Pennings proposes the use of two separate caches:
a \textit{term cache} and a \textit{function cache}. We will discuss the management of these caches in
Section 6.2.2.

6.2.1 The Visit-Function Memoization at Work

To explain in more detail how the visit-function memoization works, we shall analyse
the incremental evaluation of the \textit{BLOCK} AG. In order to simplify the discussion we consider
the memoized version of the binding-tree based evaluator \texttt{Eval[2][7]} and we focus on the ab-
stract syntax of the language. We use the simple sentence \texttt{[decl x ; use y ; decl y ;
use y]} as the input \textit{BLOCK} program and we assume that the term cache contains the DAG
presented in Figure 6.1.

We proceed now to decorate the abstract syntax tree from scratch, \textit{i.e.}, we assume that
the visit-function memo table is empty. As usual in standard evaluators, the binding-tree
evaluator traverses the tree and applies the memoized visit-function at each node. Thus,
we start with the following memoized call:

\begin{verbatim}
  memo visit_Its (1) [] 0
\end{verbatim}

At this moment the primitive function \texttt{memo} searches the memo table to check whether
this function has already been applied to these arguments or not. Since the memo table is
empty, the answer is no, and the function is really computed and memoized. In this case,
we say a \textit{cache miss} occurred.

\begin{verbatim}
  visit_Its (1) [] 0 \rightarrow ((bt1), [x, y])
\end{verbatim}
As a result of evaluating this function, the subtrees of (1) are recursively decorated. The function returns the list of declared identifiers, that we simply represent as the list of identifiers \([x, y]\), and it also returns the binding-tree, referenced as (bt1). The complete binding-tree annotated with references like (bt1) is presented in Figure 6.3. Note that these two attribute values are memoized in the term cache during the evaluation of visit1Its.

Now we are ready to perform the second traversal of the evaluator. Thus, the visit-function visit2Its is applied to the tree’s root (1), to the computed list of identifiers \([x, y]\), and to the computed binding-tree (bt1). Since this is the first call to visit2Its, a cache miss occurs. The result of evaluating this function is an empty list of errors:

\[
\text{visit2Its } (1) \ [x, y] \ (bt1) \mapsto \emptyset
\]

Figure 6.2 shows the complete function cache obtained as a result of this decoration.

| visit1Its (1) | \([\ ]\) 0 \mapsto ((bt1), \[x, y]\) | visit2Its (1) | \([x, y]\) (bt1) \mapsto \emptyset |
| visit1It (2) | \([\ ]\) 0 \mapsto ((bt2), \[x]\) | visit2It (2) | \([x, y]\) (bt2) \mapsto \emptyset |
| visit1Its (3) | \([x]\) 0 \mapsto ((bt3), \[x, y]\) | visit2Its (3) | \([x, y]\) (bt3) \mapsto \emptyset |
| visit1It (4) | \([x]\) 0 \mapsto ((bt4), \[x]\) | visit2It (4) | \([x, y]\) (bt4) \mapsto \emptyset |
| visit1Its (5) | \([x]\) 0 \mapsto ((bt5), \[x, y]\) | visit2Its (5) | \([x, y]\) (bt5) \mapsto \emptyset |
| visit1It (6) | \([x]\) 0 \mapsto ((bt6), \[x, y]\) | visit2It (6) | \([x, y]\) (bt2) \mapsto \emptyset |
| visit1Its (7) | \([x, y]\) 0 \mapsto ((bt6), \[x, y]\) | visit2Its (7) | \([x, y]\) (bt6) \mapsto \emptyset |
| visit1It (4) | \([x, y]\) 0 \mapsto ((bt4), \[x, y]\) | visit2It (4) | \([x, y]\) (bt4) \mapsto \emptyset |
| visit1Its (8) | \([x, y]\) 0 \mapsto ((bt7), \[x, y]\) | visit2Its (8) | \([x, y]\) (bt7) \mapsto \emptyset |

Figure 6.2: The function cache after decorating from scratch the abstract syntax tree of Figure 6.1 using the binding-tree evaluator Eval^2_{[71].

It is worthwhile to discuss in some detail the boxed memo entry in the function cache of Figure 6.2. It corresponds to the reuse of a previously evaluated visit-function call. Note that during the second traversal of the evaluator, the decoration of the two subtrees which correspond to the two occurrences of statement “use y” yields the same result: in this case, an empty list of errors. Furthermore, the arguments of both visits are equal: due to sharing, the two subtrees are collapsed into a single one (reference (4)), and the remaining argument is an instance of the total environment (computed in the first traversal). Thus, the visit-function that decorates the shared subtrees gets the same arguments. Consequently, the second call finds its result in the function cache and it is not evaluated. In this case, we say that a cache hit occurred. Thus, under the visit-function memoization, incrementality is also achieved when decorating a syntax tree from scratch. This reuse is only possible due to the hash-consing scheme.

Observe also that during the decoration of the second occurrence of “use y” the evaluator is reusing the value of an instance of attribute errs. That attribute instance is normally assigned to the node corresponding to the first occurrence of “use y”. Thus, under the visit-function memoization approach the values of remote attribute instances
can be reused. This is particularly important when we consider instances of higher-order attributes. Such instances may define complex computations and they should be reused to obtain efficient incremental evaluation. We will return to the incremental evaluation of HAGs in Section 6.3.

Once we have decorated the sentence from scratch, we can transform now the original sentence and, after that, perform the incremental decoration of the transformed sentence. To simplify the discussion, we consider a very simple tree transformation: the first occurrence of “use y” is transformed into “use x”. So, the original sentence, thereafter referenced as $s_1$:

$$s_1 = \texttt{[decl x ; use y ; decl y ; use y]}$$

is transformed into sentence $s_2$:

$$s_2 = \texttt{[decl x ; use x ; decl y ; use y]}$$

Figure 6.3 presents a graphical representation of the term cache after decoration from scratch. This figure also reflects the tree transformation described above. Three new nodes have been created as result of constructing the modified syntax tree with the hash-consing scheme: One node was induced by the transformation itself (reference (11)) and the other two (references (9) and (10)) are the newly created nodes in the path from the root to the modified subtree.

Figure 6.3: The term cache after decorating the sentence $s_1$ from scratch. The shared tree in the left is the binding-tree constructed during the decoration of the example sentence. The tree transformation is reflected in the “new” syntax tree shown on the right. The entries with references (1) and (3) have become garbage, ready to be purged from the cache.

We are now in a position to call the memoized attribute evaluator. Once again we start with the following call:
6.2. Visit-Function Memoization

and in return we get again a cache miss. Although the visit\textit{1}Its has been applied to the arguments [] and 0, it has not been applied to its current first argument: the newly created node (9). The function cache is now extended with the new entries presented in Figure 6.4. The boxed entries correspond to cache hits and actually they do not induce any new cache entry. The remaining calls are cache misses. So, the functions are computed and the calls memoized.

Figure 6.4: The calls to the visit-function that are memoized (the unboxed calls) and reused (the boxed entries) during the incremental evaluation of the modified abstract syntax tree of Figure 6.1.

Although the value of the arguments that correspond to the inherited attributes of the second traversal of the evaluator are the same, part of the second traversal has to be performed: the tree, \textit{i.e.}, the first argument of visit-function visit\textit{1}Its, has changed ((1) \mapsto (9)). On the contrary, the binding-tree did not change: the tree transformation did not affect the attribute instances that have to be bound from the first to the second traversal.

6.2.2 Managing the Memo Tables

During an edit session, the abstract syntax tree changes constantly, causing the term cache to grow. Visit-functions applied to a changed tree induce new entries in the visit-function cache, as well. That is to say that, both caches keep growing and have to be purged frequently.

An entry from the function cache may be removed at any time. The efficiency of the incremental evaluator may be affected, but not its correctness: if a (discarded) visit-function call is not found in the cache, the incremental engine simply (re-)computes it. To achieve efficiency, however, entries from the function cache cannot be arbitrarily discarded. So, a technique to manage the function cache has to be defined. Furthermore, the overhead introduced due to the fast growth of the function cache may decrease the performance of the incremental evaluator. Visit-functions that do little work, like the function visit\textit{1}Its when applied to Nil\textit{Its}, do not necessarily have to be memoized. That is to say, a criterion to define the set of memoized visit-functions induced by an AG has to be defined. In Section 6.8 we define such a criterion.
Entries from the term cache cannot be removed arbitrarily, because, obviously, entries corresponding to the syntax tree being edited cannot be deleted. During evaluation, the evaluator creates an internal stack which contains references to entries in the term cache. So, when the term cache is purged during evaluation, such references should also be taken into account. But, entries in the function cache also refer to entries in the term cache. Thus, terms reachable from the function cache cannot be deleted either.

There are several algorithms to perform garbage collection under a hash-consing scheme \[vD92, AG93, Pen94, vdBKO99\]. These algorithms can be used to manage the term cache, provided that the references from the function cache are taken into account as well.

Consider, for example, the nodes (1) and (3) in the term cache presented in Figure 6.3. Such entries correspond to discarded parts of the tree (due to the edit action). So, they are the garbage amenable for purging from the term cache. Note, however, that those nodes are still referenced from outside the term cache: the memoized visit-function calls applied to them are still cached in the function cache. Then, both entries have to be considered by the garbage collector. As a result, the techniques used to manage the function cache, have a direct influence on the term cache garbage collection.

### 6.3 Incremental Evaluation of HAGs

It is known that the incremental attribute evaluator for ordered attribute grammars \[Kas80, Yeh83, RT89\] can be trivially adapted for the incremental evaluation of higher-order attribute grammars. The adapted evaluator, however, decorates every instance of a higher-order attribute separately \[TC90\]. Note that in such traditional evaluators the usual representation for an attributed tree is an unshared tree, \(i.e\.), a tree represented without the hash-consing scheme. Higher-order attribute grammars define higher-order attributes, which instances are higher-order trees. Higher-order trees are constructed and destructed during evaluation, like the values of first-order attributes. The most efficient representation for such trees is a DAG. Consequently, there is a clash between the two representations, namely, tree and DAG. There are two ways to solve this tension: either we use specific techniques to decorate DAGs, or, we guarantee that the terms are, in fact, trees that are decorated by an adapted evaluator.

Let us discuss first the use of adapted evaluators. Teitelbaum \[TC90\] proposes a simple approach to handle DAGs: whenever a higher-order attribute has to be instantiated with a higher-order tree \(i.e\., with a DAG\), the tree sharing is broken and the attribute is actually instantiated with a tree. This tree is a "tree-copy" of the DAG. After that, the higher-order tree can be easily decorated by an adapted change propagator since attribute values can be associated with the tree nodes in the standard way.

This approach, however, leads to a non-optimal incremental behaviour when higher-order attributes are affected by edit actions, as shown in \[CP96\]. Note that as a result of breaking the sharing, different instantiations of higher-order attributes are, indeed, different trees. Such trees have to be decorated separately, without the possibility of reusing at-
tribute values across the different decorations of those attributed trees. Note that instances of the same higher-order attribute are likely to be (completely or partially) decorated with the same attribute values. In order to efficiently (incrementally) evaluate such instances, the reuse of those values should be achieved.

Consider, for example, the definition of the lookup operations as attributable attributes in the BLOCK HAG AG\textsubscript{3} (see Fragment 12). The production Use has an occurrence of a higher-order attribute, \textit{i.e.}, the ata table. During decoration, and for every visit to nodes that are instances of Use, the ata table is instantiated with the tree that represents the (inherited) environment. After that, the higher-order tree is decorated, having the used identifier as inherited attribute. The result of such a decoration is the synthesized attribute mBln which signals the occurrence of an error. Most likely there will be several occurrences of the same identifier in the same environment (in the sentence shown in Figure 6.1, for example, the two occurrences of “use y” inherit the same environment). In such nodes, the instantiation and decoration of the higher-order tree is exactly the same. The adapted evaluators, however, decorate those trees separately, without reusing previous evaluated attribute values.

To solve the inefficiency of the adapted change propagator algorithm, Pennings \cite{Pen94} developed a specific technique to decorate DAGs: the binding-tree based attribute evaluators. A key feature of this technique is the fact that attributes are no longer stored in the nodes of the original trees. Every tree (syntax or higher-order) is represented as a DAG that has the single purpose of guiding the attribute evaluator. It is this feature that makes the efficient incremental evaluation of HAGs possible: multiple instances of higher-order attributes do share the same DAG. No complex techniques to break such efficient sharing have to be used in order to store attribute values in the tree nodes. By using purely functional evaluators, whenever a higher-order attribute has to be instantiated with a higher-order tree, the attribute is actually instantiated with a DAG. Such a higher-order attribute is decorated by the evaluator’s visit-functions as usual.

Higher-order attributes have to be instantiated and decorated. Let us analyse first the instantiation process. Without loss of generality, consider that two instances of a higher-order attribute, say \(\alpha_1\) and \(\alpha_2\), have to be instantiated with the same higher-order tree \(T\). Using Pennings techniques, \(\alpha_1\) and \(\alpha_2\) are instantiated with references to (the DAG) \(T\). The two instances share the same data structure, saving memory consumption and copy operations. Let us consider the decoration of the higher-order attributes. Consider now that \(\alpha_1\) and \(\alpha_2\) inherit the same attribute values. This means that, both attributed trees (or DAGs) have to be decorated with the same attribute values. Using the visit-function memoization, only the decoration of one higher-order attribute is performed. The decoration of the second instance results in a cache hit: a visit-function call applied to the same arguments (\textit{i.e.,} DAG and inherited attribute values) which is found in the (global) function cache. So, its result is reused.

The binding-tree based function memoization has the following properties:

- It is an efficient and elegant way to achieve incremental attribute evaluation.
• It efficiently handles the incremental evaluation of higher-order attribute grammars [Pen94] and modular attribute grammars [CP96].

• Syntax trees are represented as DAGs, using the hash-consing scheme, which provides both fast equality comparison between terms and efficient memory usage.

• The original exhaustive attribute evaluator needs virtually no change: the underlying memoization mechanism takes care of their incremental evaluation.

• It does not need any specific purpose algorithm to keep track of which attribute values have been affected by a change in the tree. The management of the function cache is the only concern.

The visit-function memoization has three drawbacks:

• First, the memoization mechanism has an interpretative overhead: aiming at improving the potential incremental behaviour of the evaluators, Pennings proposes the use of binding-trees that are quadratic in the number of traversals. Although this approach yields binding-tree evaluators which maximize the number of cache hits, such behaviour comes at a price: first, many binding-tree constructor functions may have to be memoized, which may fill the term cache, and, consequently, increase the overhead of its search operation [AG93]. Second, the binding-trees induce additional arguments (and results) to the visit-functions. Thus, the visit-function may be extended with a large number of arguments. That is to say, having more binding-trees means having more (argument) values to test for equality, when looking for a memoized call in the function cache. Although the equality test among “hash-consed” binging-trees per se is cheap, a large number of tests may lead to a not negligible impact in searching the function cache.

• Second, no total traversal of a binding-tree evaluator can be avoided after an edit action: the underlying syntax tree changes and, consequently, all the visit-functions applied to the newly created nodes (i.e., the nodes in the path from the root to the modified subtrees) have to be really computed. The first argument of all the visit-functions that perform the different traversals has changed and, so, no previous calls can be found in the function cache. Obviously, some of the traversals may be not affected by edit actions and the visit-functions that perform such traversals should reuse their computations.

• Third, there is no obvious technique to manage the function cache which is efficient for all problems. In other words, how to define a function cache management technique which efficiently discards entries from the function cache? Note that the function cache keeps growing, and has to be purged frequently. In Section 6.8 we will present techniques to manage the function cache that are efficient in practical applications.
Pugh [PT89] proposes the memoization of the semantic function calls only. In a traditional non-functional setting Pugh’s semantic function memoization has the problem of storing attributes in the tree nodes and, consequently, there is no possibility of having tree sharing. Consequently, under a traditional implementation of AGs, Pugh’s approach does not handle HAGs efficiently. Besides, in a functional setting, the visit-function memoization is more efficient since the reuse of a visit-function call means that an entire visit to an arbitrarily large tree can be skipped. As we will explain in next section, another advantage of our new technique for the visit-function memoization is that it allows an entire visit of an evaluator to be skipped. Such incremental behaviour is not possible under Pugh’s approach: all the visits of the evaluator have to be always performed.

Nevertheless, Pugh’s approach can be easily implemented within the visit-function memoization approach if the calls to semantic functions are memoized exactly in the same way the calls to visit-functions are. Pugh’s approach can also be easily modelled within the higher-order AG formalism: the semantic functions are modelled as higher-order attributes (i.e., the role for which they were introduced) and only the calls to visit-functions that decorate such attributes are memoized. In other words, only the computations of higher-order attributes are memoized.

6.4 Memoization of Visit-Tree based Attribute Evaluator

Visit-tree based attribute evaluators are purely functional evaluators. Thus, visit-function memoization can be used to achieve incremental attribute evaluation: the visit-trees are also represented as DAGs by “hash-consing” their constructor functions, and the calls to the visit-functions (based on visit-trees) are memoized, exactly as the visit-functions based on binding-trees. The incremental evaluation of the binding-tree and the visit-tree evaluators are very similar, indeed, but there are some subtle differences.

In order to show such differences, let us consider sentence $s1$ (represented as a DAG in Figure 6.1) and the visit-tree evaluator $Eval \{3, [S2] \}$. The DAG is also the condensed representation of the visit-tree for the first traversal of this evaluator. Note that when we defined the visit-tree data type (see page 81) we omitted the traversal number for the visit-tree types of the first traversal. Like in Section 6.2.1 we begin the decoration of the abstract syntax tree with an empty function cache. The visit-tree based evaluator begins the decoration by calling the following memoized call:

\[
\text{memo} \quad \text{visit}_1\text{Its} \quad (1) \quad [] \quad 0
\]

Since the cache is empty, this function is really computed and the call memoized, exactly as in the memoization using binding-trees. This function returns the visit-tree for the next visit (referenced as (vt1)), instead of returning a binding-tree.

\[
\text{visit}_1\text{Its} \quad (1) \quad [] \quad 0 \quad \rightarrow \quad ((vt1), [x, y])
\]
After having completed the first traversal, the evaluator is now ready to perform its second one. The visit-tree based visit-function $\text{visit}_2\text{Its}$ is now applied to the visit-tree (vt1) computed previously and to the environment $[x, y]$. As expected, the result is an empty list of errors:

$$\text{visit}_2\text{Its} \ (vt1) \ [x, y] \mapsto []$$

As a result of decorating the sentence from scratch we obtain the following entries in the function cache:

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Arguments</th>
<th>Memoized Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{visit}_1\text{Its}$ (1)</td>
<td>[]</td>
<td>((vt1), [x, y])</td>
</tr>
<tr>
<td>$\text{visit}_1\text{It}$ (2)</td>
<td>[]</td>
<td>((vt2), [x])</td>
</tr>
<tr>
<td>$\text{visit}_1\text{Its}$ (3)</td>
<td>[x]</td>
<td>((vt3), [x, y])</td>
</tr>
<tr>
<td>$\text{visit}_1\text{It}$ (4)</td>
<td>[x]</td>
<td>((vt4), [x])</td>
</tr>
<tr>
<td>$\text{visit}_1\text{Its}$ (5)</td>
<td>[x]</td>
<td>((vt5), [x, y])</td>
</tr>
<tr>
<td>$\text{visit}_1\text{It}$ (6)</td>
<td>[x]</td>
<td>((vt2), [x, y])</td>
</tr>
<tr>
<td>$\text{visit}_1\text{Its}$ (7)</td>
<td>[x, y]</td>
<td>((vt6), [x, y])</td>
</tr>
<tr>
<td>$\text{visit}_1\text{It}$ (8)</td>
<td>[x, y]</td>
<td>((vt7), [x, y])</td>
</tr>
</tbody>
</table>

Figure 6.5: The function cache obtained after decorating from scratch the abstract syntax tree of Figure 6.1 using the BLOCK visit-tree evaluator Eval $\text{Eval} [\text{Eval}]$.

The boxed memo entries in Figure 6.5 correspond to cache hits and not to new cache entries. Using the visit-tree evaluator we reuse the result of two memoized calls to visit-function $\text{visit}_2\text{It}$. One of the cache hits occurred using the binding-tree memoization: the second decoration of the shared subtree (that represents the two statements “use y”) results in a cache hit. Under the visit-tree memoization a second hit occurs: it corresponds to the equal decoration of Decl nodes! Note that the two statements of the example sentence “decl x” and “decl y” induce a shared subtree for the second traversal: the two declarations do not cause any semantic error, so the respective subtrees are represented by a common subtree (referenced as (vt2)). Since the shared subtree is decorated again with the same total environment, a cache hit is obtained. This is a typical example where the visit-tree evaluator induces better memoization than the binding-tree evaluator.

Let us be more precise about the previous cache hit, i.e., the one occurring in Decl nodes. The arguments of a function play a key role during function memoization. They determine whether a function call hits or misses. Non-injective functions allow for memoization optimizations. For example, there is a possibility to increase the chance of reusing a function call by mapping different arguments yielding the same results to equal arguments. Next, we explain this technique.

Consider, for example, the equation of the visit-tree evaluator that determines the existence, or not, of a duplicate declaration.

$$\text{errs} = (\text{Pair tName lev} \ 'mNBI\text{ns'} \ dcli$$
When decorating the example sentence, this equation is calculated twice, in order to check whether identifiers “x” and “y” have been declared already. Both calculations yield the same result: an empty list of errors, although the arguments of the semantic function $mNBIn$ differ. In other words, function $mNBIn$ is not injective. Let us focus now on the visit-functions of our evaluator. This equation is scheduled for the first traversal. Consequently, the two visits to $DECL$ nodes result in the same visit-tree for the second traversal, i.e., the shared subtree ($DECL^z$ $NILERROR$). These two visit-trees are the (equal) first argument of the visit-functions that perform the second visit to declaration nodes. That is to say that, by scheduling that equation to the first traversal of the evaluator we have mapped different arguments yielding the same results to equal arguments. Consider now the evaluator $Eval$ $\delta^{[4]}48$, where the previous equation is schedule to its second traversal. In this case, the arguments of the above equation are part of the resulting visit-tree for the second traversal. Since the values of those arguments differ, they induce two unshared visit-trees. Thus, no cache hit is obtained.

We perform now the same transformation in sentence $s_1$, i.e., sentence $s_1$ is modified into $s_2$. Figure 6.6 presents the graphical representation of the term cache after evaluating the initial sentence from scratch and after performing the transformation.

Figure 6.6: The term cache after decorating the sentence $s_1$ from scratch. The shared tree in the left is the constructed visit-tree for the second traversal of the evaluator. The transformation is reflected in the new abstract syntax tree shown on the right.

Note that the subtree (vt2) represents the shared visit-tree for the second visit to declaration nodes. As we have just explained, the visit-tree approach and the hash-consing scheme allowed an extra cache hit. We proceed by incrementally decorating the newly created shared tree.

In this case, the number of cache hits using the visit-tree evaluator does not differ from the binding-tree one. Note that the tree modification was performed in one subtree ($USE$ node) that is really used in the second traversal of the evaluator. So, the visit-tree for the
second traversal is affected by the change, and consequently, no visit of the evaluator could be skipped.

Now, let us consider the change to a declaration. Recall that declarations are collected in the first traversal and their subtrees are not needed in the second one. So, we shall consider now the following transformation to the sentence shown in Figure 6.6: the first declaration "decl x" is transformed into "decl w". As a result of this transformation a new (shared) visit-tree for the first traversal has to be created. During the decoration of this tree, the visit-tree for the second traversal is being created. This visit-tree, however, corresponds exactly to the same shared tree that the evaluator computed when decorating the original sentence. The declaration nodes are shared since no error is induced by both declarations. So, the call to the visit-function that performs the second traversal of the evaluator gets the same term value as first argument (reference (vt1)). Although the underlying visit-tree does not change, in this particular case, the second visit of the evaluator could not be skipped: the change to a declaration induces a different environment, which is an argument of the visit-functions for the second traversal. So, all of those visit-functions would have to be computed and memoized.

The memoization of the calls to the functions of the visit-tree based attribute evaluator has not only improved the potential incremental behaviour of the evaluator, but it also decreased the number of equality tests required when looking for a call in the function cache. The visit-functions based on visit-trees have fewer arguments (and results) than the binding-tree ones. As a consequence, when looking for a visit-function call in the function cache, fewer equality tests have to be performed. Consider, for example, the cache hit of function \texttt{visit2It} obtained with both evaluators when decorating the shared tree corresponding to the second occurrence of "use y". Using the binding-tree evaluator the incremental engine has to perform three equality tests to detect that the function was previously applied to those arguments. The arguments were the tree (4), the environment \([x, y]\) and the binding-tree (b4) (see Figure 6.2). Using the visit-tree evaluator, only two equality tests have to be performed. They correspond to the test of the visit-tree (vt4) and the environment \([x, y]\) for equality (see Figure 6.3).
6.4. Memoization of Visit-Tree based Attribute Evaluators

6.4.1 Visit-Tree versus Binding-Tree Approach

We have developed the visit-tree based attribute evaluators not only to provide efficient and clear purely functional implementations for attribute grammars, but also to allow efficient incremental evaluation of AGs. The visit-tree based evaluators were designed in order to overcome some of the drawbacks of the binding-tree based evaluators. Next, we describe how such drawbacks can be avoided.

- The interpretative overhead due to function memoization is smaller in the visit-tree approach than in the binding-tree one. The visit-tree approach induces fewer arguments to the visit-functions and, as a result, fewer equality tests have to be performed when searching for an entry in the function cache. Note that under the visit-tree approach, every visit-function has the two following arguments: the visit-tree for a particular visit of the AE, and the inherited attributes, assigned by the scheduler to that particular visit. During incremental evaluation each argument is a term that needs to be tested for equality. Under the binding-tree approach, the visit-functions get additional arguments: they correspond to the binding-trees. Let us recall that the number of binding-trees is quadratic in the number of traversals. Consequently, a large number of terms may have to be tested for equality.

Let us consider, for example, the largest multiple traversal attribute evaluator we have constructed: the AE of the Lrc system. It performs eleven traversals over the tree. Under the binding-tree approach a single non-terminal (to be more precise, non-terminal \textit{declaration}) induces thirty four \((34!)\) binding-tree data types. Such binding-trees induce additional arguments to the visit-functions. For example, the visit-function that performs the last visit to \textit{declaration} gets nine \((9)\) additional arguments. Such arguments have to be tested for equality when incremental evaluation is considered. Under the visit-tree approach, although eleven visit-tree data types still have to be defined, the visit-functions do not get any additional arguments.

- One of the key features of the visit-tree approach is the fact that the syntax tree under decoration is dynamically specialized \textit{(i.e., during evaluation)} for each of the individual traversals. This feature not only allows for efficient memory usage (data not needed is no longer referenced), but also improves the behaviour of the incremental evaluators, because the evaluator can now skip entire traversals. To be more precise, it skips the traversals where visit-trees do not contain any reference to the changed parts of the underlying syntax trees. Under the binding-tree approach no visit can be skipped since the modified syntax tree is one argument for all visit-functions.

Although the visit-tree approach solves some of the problems of the binding-tree approach, in some situations, it can yield less efficient incremental evaluators than the binding-tree one. This situation occurs when values of attribute instances that have ITAD are affected by tree transformations. Recall that under the visit-tree approach these attribute values remain in the tree since they are created until their last use. Consequently,
unnecessary intermediate visits may have to be performed, because the visit-tree is used
to pass on such changed values.

To be more precise, let \( T' \) be an abstract syntax tree resulting from a tree replacement
at node \( N \) in tree \( T \). Without loss of generality, consider that a strict evaluator performs
three traversals to decorate \( T' \). Consider also that an attribute instance \( \alpha \) is affected by
the tree replacement and that \( \alpha \) is defined in the first traversal and used in the third one
only. Under the visit-tree approach no traversal of the evaluator can be skipped: all the
visit-trees of the evaluator are affected by the tree replacement, since all the trees store
the changed instance \( \alpha \). Although the second traversal may not be “directly” affected
by the change, the visit-functions applied to the nodes in the path from the root to the
node where \( \alpha \) is stored have to be recomputed because the visit-tree for this traversal has
changed. This is represented graphically in Figure 6.8.

Under the binding-tree approach, the binding-tree makes a “bridge” from the first
(origin) to the third (destination) traversal, in order to pass the value of attribute \( \alpha \). As
a result, the instance \( \alpha \) does not force the re-decoration of part of the second traversal of
the evaluator. Nevertheless, part of this traversal has to be decorated because the syntax
tree \( T \) has changed.

Pennings proposes a technique that completely avoids the unnecessary decoration of
non-affected traversals of the evaluator: the combination of binding-trees and split-trees
[PSV92]. A split-tree of a syntax tree \( T \) is a tuple of terms \( (T_1, \ldots, T_n) \), with \( n = v(T) \),
where each term \( T_v \) includes only the parts of \( T \) that are really inspected in visit \( v \). The
basic idea of this approach is that every function defining a visit of the evaluator gets as its
first argument the split-tree for that visit. Because the split-tree may not contain the parts
of the original syntax tree affected by an edit action, complete traversals of the evaluator
can be skipped. Values induced by ITADs are handled by binding-trees, exactly as in the
“standard” binding-tree approach.

Although split-trees and visit-trees are similar, they are slightly different: split-tree
\( T_v \) contains precisely the parts of the original tree inspected in visit \( v \), while visit-tree
contains the parts inspected in visit $v$ and following ones. Furthermore, a visit-tree contains also attribute values that have to be passed to following visits.

The use of split-trees potentially improves the incremental behaviour of the evaluator. This approach, for example, efficiently handles the situation shown in Figure 6.8. On the other hand, relying on additional arguments/results (i.e., the binding-trees) to pass attribute values between different traversals may result in a non negligible interpretative overhead, exactly as for “standard” binding-tree evaluators.

In a similar way, we can also improve the potential incremental behaviour of our evaluators by combining visit-trees with binding-trees. The basic idea is that some values induced by ITADs are passed to the following traversals in visit-trees, and others, in binding-trees. Let us be more precise: during visit $v$ of the evaluator, the attribute values that are alive in $v$ and are used in $v+1$ are passed in a visit-tree. Values alive in visit $v$ and which are used in visit $w$, with $w > v + 1$ are passed in a binding-tree. That is, the binding-trees make the bridge for values used in non-consecutive traversals. This approach will efficiently handle the situation presented in Figure 6.8.

### 6.5 Memoization of Deforested Attribute Evaluators

This section presents a new approach for the efficient incremental evaluation of higher-order attribute grammars: the memoization of deforested attribute evaluators. Deforested strict attribute evaluators consist of a set of pure visit-functions. These functions are strict in all their arguments. Consequently, standard function memoization can be applied to memoize the calls to these functions, exactly in the same way as in the memoization of the binding-tree or the visit-tree approach. To explain the memoization of $\lambda$-attribute evaluators, we make an analogy with the visit-tree approach.

- A visit-tree evaluator recursively calls its visit-functions in order to traverse and decorate a syntax tree, and, in this way, it assigns a meaning to the sentence under consideration. The deforested evaluator recursively calls its deforested visit-functions in order to assign a meaning to the sentence under consideration.

- In the visit-tree evaluator, every visit-function constructs a visit-tree for the next traversal of the evaluator. These intermediate trees are memoized. In the deforested counterpart, every visit-function returns a partially parameterized visit-function for the next traversal. These intermediate computations are memoized. That is, a tree, the *call-tree*, representing the structure of the calls to the visit-functions is memoized.

- In the visit-tree evaluator, the calls to visit-functions are memoized. In the deforested evaluator, the total parameterizations of the visit-functions are memoized. That is, a pair recording all the arguments and results of a call to a deforested visit-function is stored in the memo table. Incrementality is achieved by reusing such memoized calls.
The standard memoization of visit-tree and deforested evaluators is very similar. There are, however, some differences: for example, instead of memoizing data constructor functions, it memoizes the partially parameterization of functions. Nevertheless, the standard hash-consing scheme can be used to memoize partially parameterized functions, as well. In this case, a cache entry consists of both a reference to the visit-function and of a set of references to the values to which the visit-function has been applied. Such arguments are values induced by ITADs and other partially parameterized visit-functions. The partial parameterization of visit-functions defines the structure of the call-tree for the next traversal. Such tree is collapsed into a minimal DAG, exactly as “normal” trees are.

Moreover, we have to distinguish between the memoization of partial and total function applications. Thus, we introduce a second keyword \texttt{memo}\textit{p} to annotate the partial application of visit-functions. Note that it basically corresponds to the memoization of the term constructor function, which was omitted in the previous approach. In that case, we have assumed that constructor functions were always memoized. We introduce this new primitive to stress the memoization of partial application of functions, only. For example, the memoized version of visit-functions $\lambda_{R^1}$ and $\lambda_{ConsIts^1}$ of the $\lambda$-attribute evaluator $Eval_{[S^7]}$ looks as follows:

$$
\lambda_{R^1} \lambda_{Its^1} = errs \\
\text{where} \ \\
lev_1 = 0 \\
dcli_1 = [] \\
(\lambda_{Its^2}, dclo_1) = \text{memo} \lambda_{Its^1} dcli_1 lev_1 \\
errs = \text{memo} \lambda_{Its^1} dclo_1 \\
\lambda_{ConsIts^1} \lambda_{Its^1} dcli lev = \\
\text{where} \ (\lambda_{Its^2}, dclo_1) = \text{memo} \lambda_{Its^1} dcli lev \\
(\lambda_{Its^2}, dclo) = \text{memo} \lambda_{Its^2} dcli lev
$$

To describe this approach in more detail, let us consider again sentence $s1$ and discuss its evaluation under the memoization approach discussed thus far. To simplify the discussion, we focus again in the abstract syntax of BLOCK and we use Evaluator $Eval_{[S^7]}$. We assume that as a result of decorating the concrete tree (assigned for the above sentence), the following partially parameterized function calls have been memoized by the primitive \texttt{memo}\textit{p} (this is the case, for example, if we consider evaluator $Eval_{[G^9]}$). The cache entries are labelled for future references.

The structure of the DAG obtained by using the memoization of the partially parameterization of visit-functions is very similar to the shared syntax tree presented in
Figure 6.1. This is the expected result, since both tree structures guide the respective incremental evaluators on their first traversals. Visit-function calls that have the same partial parameterization are shared (e.g., the shadowed entry \( \lambda_{U_1} \)), in the same way as equal trees are shared under the hash-consing scheme.

Having constructed the structure of the call-tree for the first traversal of the evaluator, we are now ready to perform its first traversal, i.e., to totally parameterize the visit-functions defined in the memoized call-tree structure. Because every memoized entry contains a reference to the visit-function to be evaluated, thus, no pattern matching is really needed (this is one of the features of the deforested evaluators). Thus, evaluation starts with the following memoized call:

\[
\text{memo } \lambda_{\text{ConsIts}} (pp_2) (pp_3) [] 0
\]

As a result of performing such a call, the partial calls stored in the previous DAG are recursively evaluated, i.e., are totally parameterized. Next, we present the memoized deforested visit-function calls obtained during the first traversal of the evaluator. The first result of such functions refers to the call-tree memoized for the second traversal.

\[
\begin{align*}
\lambda_{\text{ConsIts}} (pp_2) (pp_3) [] 0 & \mapsto ((pp_2_1), [x, y]) \\
\lambda_{\text{Decl}} (x) [] 0 & \mapsto ((pp_2_2), [x]) \\
\lambda_{\text{ConsIts}} (pp_4) (pp_5) [x] 0 & \mapsto ((pp_2_3), [x, y]) \\
\lambda_{\text{Use}} (y) [x] 0 & \mapsto ((pp_2_4), [x]) \\
\lambda_{\text{ConsIts}} (pp_6) (pp_7) [x] 0 & \mapsto ((pp_2_5), [x, y]) \\
\lambda_{\text{Decl}} (y) [x] 0 & \mapsto ((pp_2_6), [x, y]) \\
\lambda_{\text{ConsIts}} (pp_4) (pp_8) [x, y] 0 & \mapsto ((pp_2_7), [x, y]) \\
\lambda_{\text{Use}} (y) [x, y] 0 & \mapsto ((pp_2_8), [x, y]) \\
\lambda_{\text{NilIts}} [x, y] 0 & \mapsto ((pp_2_9), [x, y])
\end{align*}
\]

The two calls to visit-function \( \lambda_{U_1} \) share the partial parameterization, but not its total parameterization: its first call gets [x] as the first “remaining” argument value, but the second call gets a different value, i.e., [x, y]. Thus, no cache hit occurs and the two calls are performed and memoized. The new DAG defining the call-tree for the second traversal is created and memoized as follows:

In this case, two cache hits occur due to the equal partial parameterization of visit-functions \( \lambda_{\text{Decl}} \) and \( \lambda_{\text{Use}} \). Thus, such partial computations are shared. Next, we present the memoized visit-function calls performed in the second traversal of the evaluator.
The equal partial parameterization of $\lambda_{Decl^2}$ and $\lambda_{Use^2}$ share, in this case, their total parameterization. Observe that this is exactly the same memoized behaviour presented in the visit-tree evaluator.

The standard function memoization can be directly used to achieve incremental evaluation for deforested attribute evaluators: we simply memoize the partial and the total parameterization of the visit-functions. This memoization of both parameterizations of the visit-functions, however, may yield inefficient evaluators. Observe that the partial parameterization is part of the total parameterization of a function. Consequently, redundant duplicate information is cached under this approach. This is easily observed in the previous example: the DAG representing the partial calls is duplicated in the first arguments of the memoized visit-function (total) calls. Next, we present a new approach to incremental attribute evaluation where such a redundancy is eliminated.

### 6.6 Decorating Shared Call-Trees

The different functional, incremental attribute evaluation approaches presented thus far are based on the idea of memoing function calls in a “global” memo table. These approaches, however, may induce a non-negligible interpretative overhead, since every time a memoized visit-function is called, we have to search for such a call in the memo table that contains all memoized calls. In this section, we shall introduce a new approach for incremental evaluation that reduces the overhead due to memoization by improving the locality of the memo table: the decoration of DAGs. The basic idea of our new approach is the following: the calls to visit-functions are memoized in the nodes of the DAG, instead of being memoized in the global memo table. To be more precise, calls to visit-functions are “memoized” in the context of the nodes where such calls may occur. Consequently, during attribute evaluation, when a call to a visit-function is performed (i.e., the function is totally parameterized). Then the incremental engine searches the local memo table associated with the node where the call occurs. Since that local memo table contains calls performed in the context of that particular node only, the searching operation is restricted to that set of memo entries. In this way, we achieve the desired locality of the memo table.

Informally speaking, this approach distributes the memo table over the nodes of the DAG, i.e., of the syntax tree. There is, however, a more fundamental difference with
6.6. Decorating Shared Call-Trees

respect to previous approaches: instead of simply memoing calls to visit-functions, we are now decorating DAGs with function calls. Thus, our new approach combines the standard decoration of attributed trees with the memoization of visit-function calls. Incremental evaluation is achieved as usual, *i.e.*, by reusing results of memoized visit-function calls. This approach can be used to achieve incremental evaluation both within the binding and visit-tree approaches. We shall return to this subject at the end of this section. For the memoment, we shall focus our presentation in the deforested attribute evaluators and in the decoration of shared call-trees. This new approach is based on the combination of the following ideas:

**Call-tree memoization:** The partial parameterization of the visit-functions is memoized using the hash-consing scheme. Thus, equal partial parameterizations of a visit-function are shared and, consequently, they are represented by the same call-tree structure, exactly as in the previous approach. Thus, we assign a *local memo table* to every (shared) node of the DAG in order to store values of attribute instances. Note the analogy with the standard attribute evaluation approach, where attribute instances are assigned to tree nodes to store attribute values.

**Decorated DAGs:** Attribute values are stored in the nodes of the call-tree DAG, very much like attribute values are stored in the nodes of a decorated tree. There is, however, a difference: within a decorated tree every node stores a single value for every attribute instance associated to the node (recall the definition presented in Section 2.2.1). In a decorated DAG, however, every shared node stores more than one value of an attribute instance associated to the node. Note that, shared nodes can be decorated in different contexts with different values for the same attribute instances.

**Functional Decoration:** The nodes of a DAG are decorated in a functional style. By functional style we mean that the memoized attribute values are “grouped” according to the interfaces induced by the AG scheduling algorithm. In other words, every node that is an instance of a production P for a particular visit, say v, memoizes on its local memo table the calls to the deforested visit-function \( \lambda_{P^v} \). Incremental evaluation is achieved by a standard memoization scheme performed on every local memo table.

To present this approach in detail, we proceed to discuss the deforested attribute evaluators. We start by recalling the signature of a deforested visit-function (see Section 4.1.3). For every production P and for each visit v, a visit-function with the following signature is generated:

\[
\lambda_{P^v} :: <type\_pp\_args(P, v)> \rightarrow (T \text{ inh}_1) \rightarrow \cdots \rightarrow (T \text{ inh}_k) \rightarrow (T \lambda_{P^{v+1}}, T \text{ syn}_1, \ldots, T \text{ syn}_l)
\]

where \( \text{inh}_i \) and \( \text{syn}_j \) are the inherited/synthesized attributes of visit v. The fragment \( type\_pp\_args \) defines the types of the arguments to which the visit-function is partially
parameterized in visit \( v \). \texttt{type\_pp\_args} and the function for the last visit are defined in Section 4.1.3, so, we do not repeat them here. Every visit \( v \) to a non-terminal symbol \( X \), with \( X = \text{lhs}(P) \), induces a new type \( X^v \). Next, we repeat its definition:

\[
\text{type } X^v = (T_{\text{inh}_1}) \to \cdots \to (T_{\text{inh}_k}) \to (X^{v+1}, T_{\text{syn}_1}, \ldots, T_{\text{syn}_l})
\]

The fragment \texttt{type\_pp\_args} defines the types within the tree structure that is hash-consed. That is, the primitive \texttt{memo} (presented in the previous section) constructs a shared tree node, where the types of the children of such a node are the types defined in \texttt{type\_pp\_args}. To this node we assign a memo table, say \( C \), where the total parameterizations of \( \lambda P^v \) are memoized. The total parameterization of the function is defined in the type \( X^v \). Note that \( X^v \) defines the types of the arguments (induced by the inherited attributes of visit \( v \)) and the types of the results of the function. The total parameterizations are memoized in the local memo tables of the node. Thus, the memo table assigned to nodes labelled with \( \lambda P^v \) is a set which elements have type \( X^v \), \textit{i.e.}, \( C :: [X^v] \).

It should be noticed that we are memoing the total parameterization locally to their the partial parameterizations. As a result, we do not have to repeat the memoization of all the arguments of the total parameterizations, as we did in the previous approaches. Only the arguments corresponding to the total parameterization have to be memoized, since the arguments of the partial parameterization are defined by the context of the node.

To explain in more detail how this new memoization scheme works, we shall analyse, once again, the decoration of sentence \( s_1 \). We assume that the DAG presented graphically in page 140 represents the memoization of the partial parameterization of the visit-functions for the first traversal. In our new approach we assign a memo table to every node of the DAG. Attribute evaluation is performed by totally parameterizing the call-tree structure defined by the DAG and decorating it with (memoized) function calls. Thus, evaluation starts with the following call performed in the DAG’s root node:

\[
\text{memo } \lambda_{\text{ConstIts}^1} (pp_2) (pp_3) [] 0
\]

The incremental engine searches for this call in the memo table assigned to node \( pp_1 \), \textit{i.e.}, where the call occurs (see the DAG of page 140). We assume that all the local memo tables are empty in the begining of the incremental decorations. As a result of computing the above visit-function, the hole DAG is evaluated and decorated. Figure 6.9 presents the decorated call-tree of the first traversal. The references \( pp_{21}, pp_{22}, \text{etc} \) refer to the memoized partial parameterization of the visit-functions for the second traversal.

Observe that the calls to the visit-function \( \lambda_{\text{Use}^1} \) share their partial parameterizations (recall that the two occurrences of statement “\texttt{use y}” in sentence \( s_1 \) induce two partial calls), but not their total parameterizations. Consequently, the shared node is decorated with two visit-function calls.
6.6. Decorating Shared Call-Trees

The first result of the calls are the references to the partial parameterization of the visit-functions for the following traversal. The shared call-tree for the second traversal was presented in page [141]. Note that, both calls to $\lambda_{Use}^1$ return, as the first result, the same value, i.e., reference $pp2_4$, because the two calls to $\lambda_{Use}^2$ share their partial parameterizations. In this case, however, they also share the total parameterization: they get the same remaining argument, i.e., the collected list of declarations $[x, y]$. Thus, the respective shared node is decorated with a single memoized call. The second call reuses the memoized one. The two visit-functions induced by production Use nicely present the ideas behind our approach: the sharing of partial parameterizations and the sharing/non-sharing of the total parameterizations in the second/first traversal, respectively. This is illustrated in the next figure.

The incremental evaluation of this approach is achieved by reusing the memoized total parameterizations in the nodes of the call-tree being traversed/decorated. The incremental behaviour presented by our new approach is similar to the memoization of the visit-tree and the deforested attribute evaluators presented in Sections 6.4 and 6.5 respectively. It
should be noticed that we are performing the same function calls. So, the same number of cache misses and hits is obtained in all these approaches. This is the case, for example, if we consider the transformation of sentence $s_1$ into $s_2$, as discussed previously. Although it provides the same potential incremental behaviour, our strategy is more efficient: it reduces the interpretative overhead of the incremental engine. The properties of decoration of shared call-trees are presented next.

- This approach reduces the interpretative overhead of the memoization scheme in two ways. First, it improves the locality of the cache, i.e., calls are memoized in the context where they can occur, and, as a consequence, when searching for a function call in the memo table, such a call is compared with calls previously performed by that particular visit-function in the context of that particular node. Second, it also reduces the number of equality tests that have to be performed when searching for entries in the cache: the searching algorithm does not need to test the visit-function applied in the memoized entries for equality, nor does it test for equality the tree that is the first argument of the visit-functions. Such values are specified by the context information of the local memo table.

- It provides the same potential incremental behaviour as the visit-tree based attribute evaluator. By same potential incremental behavior we mean that it yields an evaluator that computes and reuses the same number of visit-function calls.

- It is an efficient technique to share computations.

- It efficiently handles the incremental evaluation of higher-order attribute grammars. The standard hash-consing scheme is used to share equal subtrees. Consequently, the decoration of different instances of higher-order attributes also share their decorations.

- Finally, hash-consing garbage collectors can be used to manage the cache: terms representing parts of the call-tree that have become garbage due to some tree transformation are discarded by standard garbage collectors. Under our new scheme of memoization, the calls applied to such “garbage” nodes are automatically discarded as soon as the node is discarded.

The functional decoration of DAGs is orthogonal to the memoization of the visit-function approach described in this chapter. Indeed, calls to visit-functions can also be memoized in the nodes of the DAGs under both the binding-tree and the visit-tree based attribute evaluators. The visit-tree approach yields solutions very similar to the ones we have discussed in this section. Under the binding-tree approach, however, a slightly different solution as to be considered. Recall that in the binding-tree approach the syntax tree does not change during attribute evaluation. As a result, when a multiple traversal evaluator is considered, different visit-functions are applied to the same nodes. Consequently, under our new memoization scheme, calls to different visit-functions will be memoized in
the local memo table of such nodes. In order to distinguish the different memoized calls, we need to store a reference to the visit-function that originates the call. This is the usual procedure when a global memo table is considered: the first element of a memo entry is the reference to the visit-function (see, for example, Figure 6.2). Moreover, the references to the visit-functions have to be tested for equality when searching for the entry in the cache. On the contrary, either in the visit-tree or in the deforested evaluators, we have an “one-to-one” correspondence between the node and the visit-function applied/associated to it. Thus, in our new purely functional evaluators the cache overhead is smaller: no reference to the visit-function has to be included in the memoized entry, nor is it tested for equality, obviously.

6.7 Projection of Attributes

In this chapter, we have discussed thus far techniques to improve the incremental behaviour of functional attribute evaluators, either by using functional evaluators that are more likely to share their computations or by reducing the interpretative overhead of the memoization scheme. This section presents an attribute grammar transformation technique which improves the incremental behaviour of their implementations. Thus, this transformation is orthogonal to all models of incremental evaluation.

A change that propagates its effects to all parts of the syntax tree causes inefficient incremental behaviour in all the models of incremental attribute evaluation [Pen94, SKS97a]. Nevertheless, incremental evaluation can be enhanced greatly by performing a simple transformation on the AG under consideration: the projection of attributes. Consider, for example, a block structured language. Typically, every inner block inherits the context of its outer block, so, any small change in that context requires the redecoration of the inner blocks, regardless of the irrelevance of the change, i.e., even if the change is confined to a symbol that is not mentioned in the inner blocks. However, if every block synthesizes a list of used variables, the inherited context could be projected on that list, yielding better incremental behaviour.

To explain this transformation, we return to the BLOCK language. The incremental evaluation of a BLOCK sentence gives a poor (incremental) performance after a change in a declaration: when the evaluator is executed to re-decorate the changed input, most of the visit-function calls have to be really computed since one of their arguments changes (i.e., the list of declarations and the environment changes). The single exceptions are the calls to the visit-functions that decorate (in the first traversal of the evaluator) the It nodes, in the path from the root of the tree to the changed node. Note also that both visits to inner blocks cannot be avoided since its inherited initial collection of declaration changes. Let us now add a new fragment to the BLOCK AG, which projects the inherited environment of each inner block on its list of identifiers.
The semantic function project projects the environment of the outer most block on the list of identifiers synthesized for each inner block. The semantic function getentries is a primitive function defined on environments: given an identifier (i.e., a key), it returns the entries associated with that identifier. Obviously, these two inductive functions can be efficiently defined within the higher-order attribute grammar formalism. Next, we extend the higher-order grammar $AG_3$ with a new fragment which defines the desired attribute projection. We modeled the two semantic functions as attributable attributes using the technique of accumulating parameters [Bir84a].

The previous projection forces an additional visit to the body of the blocks: a specific traversal is needed to synthesize the list of used identifiers. This extra visit is reflected in the visit-sequences computed by the AG scheduling algorithms. For example, the visit-functions based on visit-trees derived from the (three) induced visit sub-sequences of production Block look as follows:
The computation of the list of used identifiers for each block is the price to pay when decorating a block sentence from scratch. The real gain of this transformation is when incremental evaluation is considered: after a change in a declaration, the entire decoration of inner blocks can be reused now. To be more precise, the decoration of inner blocks which do not use the changed identifiers is totally reused: the three calls to three visit-functions result in three cache hits. The inner blocks that use the changed identifier do have to be redecorated. However, it is worthwhile to note that the incremental re-evaluation of such inner blocks is not affected by the additional traversal: their list of used identifiers does not change, and, thus, their first visit is reused.

It should also be noticed that defining the projection via a higher-order attribute has a key advantage: the projection is incrementally computed as an immediate result of our techniques, since it is now defined by a “standard” visit-function. For example, the function \texttt{visit\textsubscript{1}Uses}, in the previous evaluator.

### 6.8 Classification Criterion for Memoing Visit-Functions

Visit-function memoization is an efficient approach to incrementally evaluate attribute grammars. Calls to visit-functions are memoized with the single purpose of speeding up the incremental performance of the evaluator. However, the complete memoization of visit-functions can, in practice, quickly decrease the performance of the evaluator \cite{SKS97a}, in spite of reusing the maximum number of computations (which is optimal in this sense). Such decrease in performance is due to the fast growth of the function cache, which, on its turn, causes the decrease in performance of the searching operation. Thus, we need a criterion to decide whether the memoization of the calls to a particular visit-function is likely to improve the overall incremental behaviour of the evaluator, and not only the number of attribute values reused. Furthermore, we also need an algorithm that is able to manage efficiently the function cache.

Attribute grammar techniques are based on a global analysis of the productions, of their attribute occurrences and the respective induced dependencies of the AG under consideration. Thus, we may use global analysis to determine which visit-functions will be more profitable to memoize.

A different approach consists in the use of dynamic information: the memoization of visit-functions applied to small subtrees, for example, may not be considered because they
are likely to do little work. Next, we define a classification criterion based on these ideas.

- **Grammar based techniques**: It includes techniques that use characteristics of the grammar under consideration to statically determine which visit-function calls will be profitable to memoize.

- **Grammar independent techniques**: It includes techniques that are independent from the attribute grammar under consideration. These techniques use dynamic information to decide which calls to visit-functions to cache and which entries to discard from the cache. It includes, for example, all the probabilistic cache management strategies.

This section presents several criteria based on properties of the attribute grammars to determine which visit-functions would be profitable to memoize. After that, we present techniques to decide which memoized visit-function calls to discard from the function cache. We start by considering one characteristic of the context-free grammars to decide which visit-functions to memoize:

**Terminal Productions**: The calls to visit-functions induced by terminal productions are not memoized. We define terminal production as a production that does not contain non-terminal symbols on their right-hand side. A $\epsilon$-production, for example, is a terminal production.

A second criterion we define is based on properties of the attribute grammars.

**Visit-sub-sequences based Techniques**: These are techniques defined according to the properties of the visit-sub-sequences.

a) **Empty visit-sub-sequences**: No memoization of visit-functions is induced by empty visit-sub-sequences. By empty we mean visit-sub-sequences that do not contain any visit nor eval instruction.

b) **eval instructions**: Memoization of visit-functions is induced by visit-sub-sequences that contain eval instructions.

c) **visit instruction**: Memoization of visit-functions is induced by visit-sub-sequences that contain visit instructions.

We assume that eval instructions induced by copy rules are not considered by these criteria. These criteria can be used to define different incremental behaviours for the same AG. For example, we may focus the memoization on the visit-functions that do compute new attribute values by using criterion b). Or, on the contrary, we may focus the memoization on the traversals themselves and use criterion c). Such criteria can also be combined in order to specialise the incremental behaviour of the evaluator.
Let us return to our running example, the block grammar \(AG_1\), and define a criterion for its incremental evaluation. We wish to define an incremental evaluator that does not memoize calls to the visit-functions that do little work. Following this idea we use criterion \(a\) and \(b\). Furthermore, we combine it with the criterion that defines the non-memoization of calls to the visit-functions induced by terminal productions. As a result, we have a strict evaluator (based on binding or visit-trees) that memoizes calls to visit-functions induced by the following sub-sequences of Visit-Sequences: visit 1 to \(R\), visit 2 to \(\text{ConsIts}\), visit 1 to \(\text{Decl}\), visit 2 to \(\text{Use}\), visit 1 to \(\text{Block}\). That is, it memoizes the visit-functions that do compute semantic functions (see evaluators \(\text{Eval}\_\{\text{71}\}\) and \(\text{Eval}\_\{\text{82}\}\)).

**Memoization of eval Instructions:** A different criterion, also based on the visit-sequences, is to consider only the memoization of the eval instructions only. This corresponds to Pugh’s semantic function memoization.

### 6.9 Function Cache Updating Algorithms

In the previous section we have presented techniques that can avoid the memoization of calls to visit-functions. We shall now describe techniques to manage and discard entries from the function cache. The cache updating algorithms discussed in this section aims at managing the global function cache used by the standard visit-function memoization approach.

#### 6.9.1 LRU based Purging Strategy

One of the most commonly used cache replacement strategies is the *Least-Recently-Used* (LRU) algorithm: as the name indicates, it simply discards the least recently used entries from the cache. The algorithm is very simple and its update operation is fast. However, this simple algorithm *per se* is not suited for incremental attribute evaluation. Next, we explain why, using a simple example.

Without loss of generality, consider a two traversal attribute evaluator in which both traversals perform approximately the same number of calls to visit-functions. Let us assume that the cache can only store 50% of the calls needed to be computed when processing a particular input from scratch.

As a result of processing this input from scratch, the function cache contains the entries which correspond to the calls performed in the second traversal of the evaluator: after the first traversal the cache is full, and, consequently, during the second traversal the LRU algorithm replaces the old entries (corresponding to the first traversal) by the new entries (the the visit-functions being called in the second traversal). Let us proceed to perform a slight change in the input program. The incremental re-evaluation of the input presents a poor incremental behaviour: the evaluator begins with its first traversal and no function call is found in the cache. Consequently, the LRU algorithm replaces the old entries (the ones performed in the previous second traversal) by the new ones (the first traversal). When
the evaluator starts its second traversal, then, the function cache contains the entries from
the previous pass. In other words, the function cache always contains the visit-functions
applied in the “other” traversal. As a result, no visit-function call is actually reused
\[SKS97a\].

### 6.9.2 Obsolete based Purging Strategy

This section presents a possible strategy to manage the function cache which is similar
to the schemes used by garbage collectors: (cache) entries that have become garbage are
purged (from the cache). These obsolete entries correspond to calls to functions that would
not be created when evaluating the current input from scratch. Such entries are defined
next.

**Definition 6.1 (obsolete visit-function call)** A memoized visit-function call is consid-
ered obsolete if this function call is not made during an evaluation from scratch of the syntax
tree currently under consideration.

The obsolete based purging strategy discards obsolete visit-function calls from the func-
tion cache. It should be noticed, however, that under our model of incremental evaluation
an obsolete visit-function call can always be reused in a future re-evaluation of the input.
Typically, this happens after the user performs an undo operation.

The obsolete based purging strategy is based on the idea that visit-function calls, ap-
plied to subtrees which are not part of the tree being edited, have a low probability to be
reused and, thus, are discarded. This is exactly the same idea used by the hash-consing
garbage collectors: terms that are not part of the tree being edited are discarded as well.
This purging strategy mimics the traditional approach of incremental evaluation, i.e., only
attribute values assigned to nodes of the tree being edited are kept in memory. In our
approach, those attribute values are simply the arguments/results of the memoized calls
kept in the caches. Next, we present a simple algorithm that manages both the term cache
and the function cache using this strategy.

**Algorithm**

1. **Term Cache**
   - Mark all nodes in the tree being edited

2. **Function Cache**
   - Purge obsolete entries:
     - Discard visit-function calls applied to unmarked nodes
   - Mark all terms reachable from the function cache
   - Hash-consing garbage collection

Although this strategy seems appealing at first sight, a complication arises when higher-
order attributes have to be considered. Note that higher-order attributes are computed
during attribute evaluation and, as a result, the induced higher-order trees are not part
of the tree being edited. In other words, higher-order trees are not reached from the root
of the tree being edited. So, obsolete higher-order trees and the visit-function calls that
decorate them are not considered by the previous algorithm.
6.9.3 ULE based Purging Strategy

The algorithm presented in the previous section aims at determining the set of entries in the cache that have a low probability to be reused in future re-evaluations of the input: the obsolete entries. On the contrary, in this section, we present an algorithm that focuses on the entries that have a high probability to be reused in next re-evaluations of the input. The algorithm, the so-called Used in Last Evaluation (ULE), assumes that the visit-function calls reused in the re-decoration of the syntax tree under consideration have a high probability of being reused in the next re-decoration. In other words, the ULE algorithm assumes that there is some locality on the sequence of tree transformations.

A straightforward cache updating strategy is to keep only in the cache the entries corresponding to the calls to the visit-functions actually performed during the incremental evaluation of the syntax tree currently under consideration. This purging algorithm is very simple: during an incremental evaluation of the input, it simply constructs a new cache containing the calls to visit-functions that result in cache misses and hits. To be more precise, consider that the syntax tree \( t' \) results from a tree transformation at node \( N \) in tree \( t \), and let \( \Delta \) be the affected attribute instances. Let \( new(t', C) \) be the set of visit-function calls that have to be computed to evaluate instances in \( \Delta \), i.e., the cache misses, and let \( used(t', C) \) be the set of memoized calls that are reused when decorating \( t' \). Then, the new cache obtained when processing \( t' \) is \( C' = new(t', C) \cup used(t', C) \). Under this approach, the number of memo entries of \( C' \) is smaller than the cache that would be obtained when processing \( t' \) from scratch.

As a result of this strategy, the cache \( C' \) is “specialized” in tree transformations occurring in subtree rooted \( N \): it only contains the entries that are reused when re-decorating the syntax tree after a transformation in subtree rooted \( N \). Thus, it induces a small interpretative overhead when the transformations are local to that subtree. Note that, this strategy keeps in the cache the visit-functions applied in the root of the subtrees in the path from the root to the changed subtree: they correspond to cache hits, i.e., the set \( used(t', C) \).

A problem arises when a tree transformation is performed in one of such subtrees. Since only the visit-functions applied to the root of the subtree are memoized, the complete subtree has to be re-evaluated from scratch: no memoized calls decorating the subtree are found in the cache \([SKS97a]\). Not surprisingly, this straightforward algorithm gives a poor incremental performance after the user changes the focus of his interactions.

In order to solve the inefficiency of this algorithm on this particular situation, we propose the use of a dynamic memoization criterion to decide which calls to cache: the size of the subtree can be used to determine whether a visit-function call is purged from the cache or not. The trees (i.e., the DAGs) are constructed bottom-up, thus, the size can be automatically computed at tree construction time. Consequently, a more efficient ULE based purging strategy keeps in the cache the visit-function calls that are applied to large subtrees. Furthermore, the purging strategy can be parameterized with the size that defines the minimum tree-size for the visit-function to be kept in the cache. For example, it can be a percentage of the total syntax tree being decorated \([SKS97a]\).
Chapter 7

**LRC: A Generator for Purely Functional Language-based Systems**

**Summary**

This chapter presents the attribute grammar based system LRC. The architecture of the system is described in detail. The higher-order attribute grammar processor and the visit-function generator are described as well. The construction of a language-based environment is defined. Finally, the performance of the purely functional attribute evaluators is analysed.

As part of our research on deriving purely functional attribute evaluators for higher-order attribute grammars, we have implemented our techniques in the LRC system [KS98]. The LRC system is a generator for purely functional language-based systems developed at Utrecht University. Matthijs Kuiper started developing LRC during his research on parallel attribute evaluation [Kui89]. After that, Vogt, Swierstra and Kuiper [VSK89] continued the development of LRC for their work on higher-order attribute grammars. Maarten Pennings [Pen94] conducted the first implementation experiments on incremental evaluation. This thesis continues the work of Pennings.

The LRC system accepts as input a higher-order attribute grammar and generates purely functional attribute evaluators. These attribute evaluators are based on the techniques described in Chapters 3 and 4. Thus, LRC generates strict, multiple traversal attribute evaluators and lazy attribute evaluators. Efficient incremental attribute evaluation is obtained via function memoization, using the techniques described in Chapter 6. The LRC system produces C and HASKELL based attribute evaluators.

### 7.1 The Architecture of LRC

The LRC system consists of three main components: the *higher-order attribute grammar processor*, which processes the HAG specification; the *visit-function generator*, which
produces the functional attribute evaluators; and the runtime system, which contains an incremental evaluator engine and a visualization engine for supporting incremental evaluation and visualization of graphical user interfaces, respectively. Figure 7.1 shows the architecture of the LRC system.

Figure 7.1: The Architecture of LRC.

7.1.1 The Higher-Order Attribute Grammar Processor

The higher-order attribute grammar processor, written by Matthijs Kuiper, is the front-end of the LRC system. It processes attribute grammar specifications written in a superset of Ssl, the Synthesizer Specification Language [RT89]. It extends Ssl with two new features. First, it provides support for higher-ordeness. Second, it provides support for describing user interactions. We have incorporated the generation of Haskell code in the front-end of LRC.
From a HAG specification, four different things are produced: a specification for a scanner generator, a specification for a parser generator, the set of semantic functions and an abstract description of the attribute grammar under consideration. The specifications produced for the scanner and parser generators are based on regular expressions and context-free grammars, respectively. LRC is not tied to any particular scanner or parser generator tools. Nevertheless, for portability reasons, the current version of LRC uses LEX/YACC based tools. The set of semantic functions is generated in one of the target languages of LRC: C or Haskell. So, the semantic function processor translates the semantic functions written in SSL into one of these languages. The abstract attribute grammar is an intermediate language that abstractly describes the HAG: it defines the non-terminal symbols, the attributable attributes, the productions, the attribute occurrences and the dependencies among them. Roughly speaking, it only contains the parts of a HAG needed by the standard AG algorithms: the attribute occurrences and their dependencies. This representation is independent of any particular notation to describe AGs. Thus, it makes the LRC architecture highly modular: new front and back-end components can easily be “plugged in” to the system. This language can also be seen as an abstract representation of a lazy circular program: circular definitions occur in a form of attribute dependencies. Besides generating code for the remaining components of the system, the front-end of LRC also performs the name analysis and type checking of the source attribute grammar. That is, any semantic error and type error is detected by this processor (e.g., use of non-defined attributes, type errors in the semantic equations, etc).

This processor is written in the LRC specification language. So, the system can bootstrap itself. The HAG describing the LRC bootstrap grammar consists of \( \approx 20000 \) lines of a higher-order attribute grammar specification, and, as far as we know, this is the largest HAG ever written. AG ordered scheduling algorithms assign 11 visits to some non-terminals of the grammar. In other words, a strict, purely functional evaluator performs 11 traversals over the syntax tree to compute its meaning. As we have discussed in this thesis, it would be extremely complex to write such program by hand.

### 7.1.2 The Visit-Function Generator

The visit-function generator is the back-end of the LRC system and its attribute evaluator generator. As shown in figure 7.1 it processes the abstract attribute grammar, produced by the front-end, and generates purely functional attribute evaluators. It produces C and Haskell based attribute evaluators. The evaluators differ in their model of evaluation: LRC produces strict, multiple traversal attribute evaluators, and lazy attribute evaluators. The former are C or Haskell based evaluators, while the later are Haskell based evaluators only.

**Lazy Attribute Evaluators:** LRC generates lazy attribute evaluators according to the mappings presented in Sections 3.5 and 4.2. The evaluators are expressed as circular programs under the lazy Haskell language. In order to guarantee termination of such
evaluators we use the circularity test implemented in Lrc to check for (static) circularities in the grammar under consideration. If circularities are detected, a warning message is produced. Nevertheless, the circular program is generated, and it is up to writer of the attribute grammar to ensure the correctness of such a program.

**Strict, Multiple Traversal Attribute Evaluators:** The generation of strict, multiple traversal attribute evaluators follows the steps presented in Chapters 3 and 4. First, the visit-sequences are computed. After that, the attribute lifetime analysis is performed and, finally, the mapping into visit-functions is performed. Next, we describe how these steps are implemented in Lrc.

The visit-sequence generator checks whether the attribute grammar is an ordered attribute grammar or not. If the grammar is ordered, the static scheduling of attribute evaluation is performed and the annotated visit-sub-sequences computed. Pennings implemented the two scheduling algorithms discussed in this thesis: the standard Kastens’s ordered scheduling algorithm and the chained scheduling algorithm (his own algorithm).

**Attribute grammar lifetime analysis:** The visit-sub-sequences are the basis of the attribute lifetime analysis. Basically, this analysis determines the set of attribute occurrences that have to be preserved during different traversals of the evaluator. Two attribute lifetime analysis have been implemented: Pennings implemented the binding analysis and its emptiness test, in order to induce the binding-tree data types. Binding analysis is the basis of the binding-tree approach. We have implemented the attribute lifetime time analysis described in Section 4.1.1. As a result of our analysis the sets alive and inspect are computed.

The generation of visit-functions: The three strict and purely functional implementations for HAGs discussed in this thesis have been implemented in Lrc. Pennings implemented the mapping from HAGs into binding-tree based attribute evaluators presented in Section 3.6.6. This implementation generates C programs which use the incremental engine included in Lrc. We have incorporated in Lrc the mappings into visit-tree based attribute evaluators and λ-attribute evaluators, presented in Sections 4.1.2 and 4.1.3. Furthermore, we have included the generation of Haskell based attribute evaluators. The binding-tree, visit-tree and deforested evaluators presented in this thesis were automatically produced by Lrc from an AG specification. The C version of the visit-tree based evaluator and the deforested evaluator are not yet incorporated in the back-end of Lrc. We have implemented also an algorithm that eliminates redundant copy rules from the evaluators’ code. Next, we present the textual code produced by Lrc for the deforested visit-function $\lambda_{R^1}$ of evaluator $Eval_{4,87}$.
7.1. The Architecture of Lrc

\[
\text{lambda}_R_1 \ t_{Its}_1 = (x_{errs}_1) \\
\text{where} \\
x_{lev}_1 = 0 \\
x_{dcli}_1 = [] \\
(t_{Its}_2, x_{dclo}_1) = t_{Its}_1 x_{dcli}_1 x_{lev}_1 \\
(x_{errs}_1) = t_{Its}_2 \ x_{dclo}_1
\]

Now, we will briefly describe other features we have incorporated into the visit-function generator:

**Sliced Attribute Evaluators:** The visit-function generator produces sliced attribute evaluators. A *program slice* consists of the parts of a program that potentially affect the values computed at some point of interest, referred to as a *slicing criterion* [Tip94]. Following the example of Horwitz and Reps [HR92] we have defined the slicing algorithm in terms of operations on a dependency graph that represents the programs to slice. In our case, the graphs are the dependency graphs induced by the (abstract) attribute grammar being processed. We have implemented a *backward slice* technique, i.e., we compute the slices by way of a backward traversal of the graph, starting at the slicing criterion. As slicing criterion we consider a subset of the synthesized attributes of the root symbol of the AG. Thus, the sliced evaluator includes only the computations needed to evaluate the synthesized attributes of the AG’s root considered in the slice.

To present in more detail the sliced attribute evaluators produced by Lrc, we consider the block grammar \( AG_1 \) and define a sliced attribute evaluator. First, we have to define the slicing criterion. In order to have a more interesting slice, let us assume that the root symbol \( P \) synthesizes attribute \( dclo \), which defines the list of identifiers declared at the outermost block. Let us assume that we are now interested in computing the list of declarations of the outermost block of a block program and not its list of errors. Next, we present the deforested visit-function \( \lambda_{R_1} \) derived from a backward slicing of the \( AG_1 \) with respect to the slicing criterion “\( P.dclo \)”. The components of attribute evaluator that are “shadowed” are not in the slice.

\[
\lambda_{R_1} \lambda_{Its} = (dclo_1, \ errors_1) \\
\text{where} \\
(lev_1) = 0 \\
(del_1) = [] \\
(\lambda_{Its_2}, dclo_1) = \lambda_{Its_1} \ deli_1 lev_1 \\
\text{errors}_1 = \lambda_{Its_2} \ dclo_1
\]

We did not not yet include any support to specify the slicing criterion in the front-end of the system. The slicing criterion is defined directly in the input of the visit-function generator, i.e., by just changing the set of synthesized attributes of the root symbol defined in the abstract attribute grammar.

**Parallel Evaluation:** Lrc includes a parallelism detector which identifies independent visit-
function calls. It corresponds to independent visit instructions within the visit-sequence paradigm as proposed by Jourdan [Jou91]. Our purely functional attribute evaluators are amenable for parallel evaluation since the visit-functions do not have side effects. The Haskell based evaluators can be annotated with parallel directives.

Let us consider again the block grammar $AG_1$ and use the parallel detector to determine which visits can be performed in parallel. Next, we present the induced deforested visit-function where potential parallelism is detected.

$\lambda_{\text{ConsIts}} \cdot \lambda_{\text{It}^1} \cdot \lambda_{\text{Its}^2} \cdot \text{dclo} \cdot \text{lev} = (\lambda_{\text{ConsIts}^2} \cdot \lambda_{\text{It}^2} \cdot \lambda_{\text{Its}^2}^2, \text{dclo}_2)$

where $(\lambda_{\text{It}^2, \text{dclo}_1} = \lambda_{\text{It}^1} \cdot \text{dclo} \cdot \text{lev})$

$(\lambda_{\text{Its}^2^2, \text{dclo}_2} = \lambda_{\text{Its}^1} \cdot \text{dclo}_1 \cdot \text{lev})$

$\lambda_{\text{ConsIts}^2} \cdot \lambda_{\text{It}^2} \cdot \lambda_{\text{Its}^2}^2 \cdot \text{env} = \text{errs}$

where $\text{errs}_1 = \lambda_{\text{It}^2} \cdot \text{env}$

$\text{errs}_2 = \lambda_{\text{Its}^2}^2 \cdot \text{env}$

$\text{errs} = \text{errs}_1 + \text{errs}_2$

<table>
<thead>
<tr>
<th>Dependencies detected:</th>
<th>The visits can be computed in Parallel!</th>
</tr>
</thead>
</table>

The parallelism detector infers that in the first visit to production $\text{ConsIts}$, the visits to its children cannot be performed in parallel: one argument to the call $\lambda_{\text{It}^1}$ is computed by $\lambda_{\text{It}^1}$. The second visit, however, can perform the two visits in parallel: the visits are independent. It is worthwhile to recall that it is during the second traversal to a block that the evaluator descends to its inner blocks. Thus, the parallelism detector has determined that inner blocks can be decorated in parallel. Observe that, under the style of circular programming this potential parallelism is hidden within the circular definitions: the (sequential) collection of the environment and the (parallel) computation of the errors are merged in one single function (see $\text{Eval}_{\text{Eval}}$ and $\text{Eval}_{\text{Eval}}$). Consequently, no parallelism is detected for the block circular attribute evaluators.

**Readable Attribute Evaluators**: LRC produces concise and readable attribute evaluators: the Haskell based attribute evaluators. Moreover, it generates a (colored) \textsc{LaTeX} version of the Haskell based attribute evaluators. Actually, the attribute evaluators presented in this thesis were automatically produced by LRC. The \textsc{LaTeX} evaluators use different colours and fonts for different entities of the evaluators. For example, when slicing an AG, it “shadows” the computations that are “sliced out” from the evaluators code, exactly has presented above.

### 7.1.3 The Runtime System

The runtime system consists of three components: the **incremental evaluator engine**, the **visualization engine** and the **garbage collector**.

**The Incremental Engine**: This is the part of LRC that makes incremental evaluation possible. The incremental engine is a C based implementation of the standard function memoization techniques described in Chapter 6. That is, it implements data constructor memoization and visit-function memoization. Two separate caches are used: the term cache
and the function cache. Both caches are defined as “global” hash tables and are implemented as an array of lists of term nodes. Pennings constructed the C incremental engine. For a detailed discussion about the implementation of the incremental engine of LRC we refer the reader to Pennings’ thesis ([Pen94], Chapter 6). We have updated the incremental engine with different cache management strategies. The LRU and the ULE strategies defined in Sections 6.9.1 and 6.9.3 respectively, are implemented in LRC [SKS97a]. Furthermore, the incremental engine can be parameterized with the size of the subtrees that determine whether a visit-function call is memoized, or not. Unfortunately, we have not yet implemented the new model of incremental evaluation based on the decoration of shared call-trees in the incremental engine of LRC.

**Advanced Graphical User Interface**: The LRC system includes a modern graphical user interface engine. The attribute evaluators, or to be more precise, the tools produced by LRC have advanced graphical user interfaces. The interface is described within the attribute grammar formalism. During attribute evaluation one of the attributes being synthesized is an abstract description of the interface. Thus, the interface of LRC generated tools is computed. The interface is presented on the screen by the visualizer. This component is itself incremental: after a change to the syntax tree under consideration, the interface is incrementally evaluated by the incremental engine (like any other attribute value), and the visualizer (incrementally) updates the screen parts that have changed. It computes the difference between the current visual structure presented on the screen and the new visual structure induced by the change. This difference is translated into incremental updates of the screen. The advanced graphical user interface is supported in the C based attribute evaluators only.

A set of visual objects is pre-defined in the LRC prelude that can easily be included in the HAG specification. Those objects correspond to standard graphical user interface objects, like menus, buttons, etc. A “textual object” that displays the unparsing of a syntax tree is used to offer more traditional textual editing. The unparsed representation of the syntax tree is induced by the unparsing rules defined in the HAG. Those rules are written in Ssl code. The graphical objects can be combined horizontally or vertically in order to form the interface of the tool. Moreover, a set of properties can be defined for each of the objects, which specifies its size, colour, fonts, etc. In the next section, we describe some of these visual objects.

*The garbage collector*: The Boehm-Demers-Weiser conservative garbage collector [BW88] is used to reclaim unreachable memory.

### 7.2 A Language-based Environment for the BLOCK Language

To illustrate how modern language-based environments are easily defined in LRC, we shall extend the specification of the BLOCK language with the advanced graphical objects
included in the LRC prelude and supported by the runtime system. To show the notation of LRC specification language, we present the textual code exactly as written in LRC.

We start by defining a traditional language-based editor for the BLOCK language. Such an editor displays a “pretty printed” version of the syntax tree that represents the input. The prelude of LRC contains the constructor Unparse to provide this facility. Thus, to obtain a language-based editor for BLOCK, we simply need to extend our grammar with a new attribute that synthesizes the visual object, where the constructor Unparse is used to compute values of this visual object. Next, we present the LRC specification (left) and the synthesized visual object (right) that is presented to the user as one of the results of processing the BLOCK example sentence (shown is page 20).

```lrc
Prog { syn lrc_visuals guiObjects};
Prog : Root
{ Prog.guiObjects =
    let editor_frame = Unparse(&Stats.ast);
    in (Toplevel(editor_frame,"editor","Block Editor")::LrcVisuals0);
};
```

The root symbol Prog synthesizes an attribute that represents the list of graphical user interface objects defined in the attribute grammar. The type lrc_visuals is the predefined type for the visual objects to be displayed by LRC. The constructor Unparse takes as argument a reference to the abstract syntax tree to be displayed. This tree is displayed according to the unparse rules defined for each of its productions. As a result of using the Unparse constructor, the editor of the BLOCK language allows the user to perform traditional language-based editing. A Toplevel constructor displays a frame in a window. It takes three arguments: the frame, a name and the title. The title is displayed at the top of the window. The name must be unique within a list of toplevels for one tree.

We can also specify the size of the frame where the syntax tree is to be displayed. We define a frame with 40 columns and 20 lines with the Size constructor as follows:

```lrc
editor_frame = Size(Unparse(&Stats.ast),40,20)
```

The root symbol of the BLOCK AG synthesizes the attribute errs. So, let us define a visual object to display this attribute as well. We have two alternatives here. Either we display the errors within the abstract syntax tree or we display them in a different frame. In the first case, the language-based environment displays the value of the attribute in the program being manipulated, i.e., it points to the user where, in the input text, the error occurs. This is the approach we follow in the editor that we will present in Figure 7.2. In the example we present here, we take the second approach, i.e., we define a new Unparse constructor to display the unparsing of attribute errs. Furthermore, we combine both unparsing frames into one by using the VList frame combinator. Next, we show the let expression only:
7.2. A Language-based Environment for the BLOCK Language

```haskell
let editor_frame = Unparse(&Stats.ast) ;
error_frame = Unparse(&Prog.errs) ;
vcomb_frame = VList (editor_frame :: error_frame :: Frames0);
in (Toplevel(vcomb_frame,"edit","Block Editor")::LrcVisuals0);
```

We proceed now to define a more advanced visual object. We add a *push button* to the user interface in order to allow the user to add statements to the input BLOCK program by pressing a simple button. We use, in this case, the constructor `PushButton` of the LRC prelude. This constructor has a single argument: the string to be displayed in the button. Next, we extend with a local attribute defining a *push-button*. The BLOCK environment shown in the right is the result of combining vertically the push-button with the previous two unparsing frames.

![Block Editor](image)

The `PushButton` constructor displays a push button only. To assign an *action* to the displayed button, however, we have to specify such an action in the AG. The LRC prelude includes a set of pre-defined *event-handler* constructors to specify how user interactions with the graphical objects are translated into changes in the object being edited. The constructor `ButtonPress` is the event-handler associated with `PushButton`. Next, we show a possible action associated with this event-handler.

```haskell
Progs : Root
{ local frame button_add_entry;
  button_add_entry = PushButton("Add Statement");
};
```

The `Lrc bind` expression is used to specify how user interactions are handled by the language-based environment. In this case it simply defines that every time the push button "Add Statement" is pressed, the subtree rooted `Its` is transformed into `ConsIts (Decl("a"),NilIts)`. Note that this event-handler constructor is defined in the context of a `NilIts` production. Thus, a new declaration is added at the end of the input program.

We have presented two ways to specify the user interaction with the environments produced by LRC, *i.e.*, by traditional editing within the `Unparse` constructor, and by direct manipulation of pre-defined visual objects. There is a third way to specify user interactions: by defining *tree transformations*. Tree transformations are a feature provided by the SSL language. For each production of the grammar (*i.e.*, constructor) we can
associate an expression defining how the production is translated. Next, we associate several transformation rules to non-terminal \textit{It}. The first rule, for example, defines that if the user selects the transformation name "USE" in the context of a production \textit{Decl}, then the production is transformed into a \textit{Use} production. The identifier declared/used in the statement is represented by the pattern variable \textit{n} and does not change. In the environments produced by LRC the list of possible transformations is presented to the user in a pop-up menu, as shown in the environment on the right.

\begin{verbatim}
transform It
    on "USE " Decl(n): Use(n),
    on "BLOCK" Decl(n): Block(ConsIts(Decl(n),NilIts))
    on "DECL " Use(n) : Decl(n),
    on "BLOCK" Use(n) : Block(ConsIts(Use(n),NilIts))
\end{verbatim}

The visual objects defined in the LRC prelude, and the different ways of interaction with the tools generated by LRC makes it possible to define easily powerful interactive systems with LRC. For example, the BiB\TeX language-based environment presented in Chapter 1 (Figure 1.1) was produced by LRC from an attribute grammar specification. The complete environment is specified in 931 lines of LRC code.

### 7.2.1 Applications Developed using LRC

The LRC system has been used to develop several kinds of language-based environments, compilers and other tools [KS98]. LRC is used at Utrecht and Minho University to support courses on implementation of programming languages and on compiler construction. It has also been used in several industrial renovation projects. Next, we briefly show a list of applications which we have developed with LRC.

- The LRC itself, \textit{i.e.}, the LRC bootstrap grammar.
- The microC \textit{compiler}: A language-based environment for a micro C like compiler. This simple compiler will be used as the benchmark of our incremental evaluator in the next section.
- \textit{Unix find command}: An interactive interface to the Unix find command [KS98].
- \textit{XML language-based environment}: An environment to define XML documents.
- \textit{Pretty Printing Combinators}: We have used LRC to produce a strict, purely functional implementation of a pretty printing combinator library [SAS98].
7.3 Performance of the Incremental Engine

In this section we discuss the performance of the incremental engine of LRC. Unfortunately, the visit-tree and the deforested approach have not been incorporated yet in the c back-end of the visit-function generator. This fact prevents us from extensively profiling the different strict, attribute evaluators generated by LRC. Consequently, we cannot compare accurately the performance of incremental evaluation under the binding-tree, visit-tree and deforested evaluation approaches. Furthermore, the decoration of shared call-trees is not incorporated into the incremental engine of LRC. Thus, it forbids us to compare the interpretative overhead induced both by a standard global memoization table scheme, and by our new approach based on the decoration of DAGs, using local memo tables.

Nevertheless, we have implemented a visit-tree based evaluator in the c language in order to compare its performance with the binding-tree one. That is, we have transformed a c based binding-tree evaluator produced by LRC into a c based visit-tree evaluator. This section presents the results obtained with the incremental engine of LRC.

7.3.1 Binding-Tree versus Visit-Tree Performance Results

In order to conduct meaningful tests, we need to define a realistic input. Language-based environments and their specifications via attribute grammars are based on the processing of a formal language. Thus, to profile such environments, we have to choose a realistic language, and its grammar specification.

The microC language: The microC language describes a tiny c based programming language. This language was used in the Department of Computer Science of University of Minho, to teach first year students basic algorithms, and their representation in a stack based assembly language called Msp. We have constructed a language-based environment for microC with LRC. Figure 7.2 shows the language-based environment of microC. This environment is specified by a higher-order attribute grammar that consists of 24 non-terminal symbols, 75 productions and 1570 lines of LRC code. The grammar describes the name analysis and the type checking of the language. The meaning of a microC program is defined as a mapping into the Msp assembly code (the window in the right of Figure 7.2).

Having defined the language and its AG specification, we need to define a realistic input program for the attribute evaluator. The microC program used as input for these benchmarks consists of the exercises proposed to the students. Part of the input program is displayed in Figure 7.2.

We proceed now to discuss the attribute evaluator produced by LRC from the attribute grammar specification.

Attribute Evaluators: The binding and the visit-tree based attribute evaluators generated by LRC for the microC HAG perform two traversals over the abstract syntax tree. Two traversals are required because microC does not force a declare-before-use discipline, exactly as in the BLOCK language. Thus, declaration of global variables and definition of
functions are not required to occur before their first use. The body of the functions, however, is decorated in a single traversal. This occurs because no nesting of functions is allowed in microC. The binding-tree approach uses a single binding-tree to glue together the two traversal functions, while the visit-tree approach uses a visit-tree. Both evaluators use the same syntax tree in their first traversal. It is worthwhile to note that under a two traversal scheme the binding-tree and the visit-tree approach induce the same number of intermediate trees: a single binding or visit-tree, respectively. Thus, the overhead due to construct/memoize during evaluation a large number of intermediate data structures used by the binding-tree approach is not reflected in such evaluators.

Next, we present results obtained when executing the binding-tree and the visit-tree attribute evaluators. We present the number of cache misses (functions evaluated), cache hits (functions reused), the number of equality tests performed between (shared) terms and the execution time in seconds. We have clocked the execution time on a plain Silicon Graphics workstation.
The above table presents results obtained both with exhaustive evaluation, i.e., without memoization of the calls to the visit-functions, and with incremental evaluation, i.e., with memoization of the visit-function calls. The results of exhaustive evaluation obtained with the binding and the visit-tree evaluator are very similar. Under a two traversal evaluation scheme, the binding-tree approach does not induce a large overhead due to the large number of intermediate trees that are constructed during evaluation to glue the different traversals of the evaluator. In this case, the results of exhaustive evaluation under the binding and the visit-tree approach are very similar. It should be noticed, however, that the visit-functions that perform the second traversal of the binding-tree evaluator get an additional argument: the binding-tree gluing the traversal functions.

The results of the incremental evaluation refer to the processing of the input from scratch, i.e., the memo tables are empty at the beginning of evaluation. In the results of incremental evaluation we consider two evaluators, the binding and the visit-tree based evaluators, and two configurations of the function cache. These configurations differ in the size of the hash array used to implement the cache. Recall that the cache is implemented as an array of collision lists. As expected, the interpretative overhead of the function cache is affected by the configuration of the hash array: decreasing the size of the hash array induces a greater execution time in both evaluators. This decreasing in performance is due to the increasing in the number of equality tests that are performed when searching the cache. In the visit-tree evaluator this number increases 10 times. The configuration of the cache does not influence the number of cache misses and hits. The visit-tree induces fewer misses than the binding-tree approach. That is, fewer visit-functions have to be computed (8% fewer functions are computed). As we have explained in Section 6.4, the visit-trees are being specialized for each individual traversal of the evaluator and, thus, they are more likely to share subtrees and, consequently, to share their decorations. Furthermore, the number of equality tests performed under the visit-tree approach is considerably smaller than with the binding-tree one. The binding-tree induces an additional argument to the visit-functions, and, as a result, additional equality tests have to be performed when searching such calls in the cache. The visit-tree approach presents a better incremental behaviour in both configurations of the cache.

*edit actions*: To profile the incremental behaviour of our attribute evaluators we have considered two kinds of modification to the input microC: we consider a modification local to
the body of a function and a modification that is global to the whole program. To be more precise, we modified the program in two ways: by adding a statement to a function and by adding a global variable. Furthermore, we performed such modification at the beginning and at the end of the input program. Note that, our model of incremental evaluation is sensitive not only to the modification itself, but also to where such modification occurs in the input: the visit-functions applied to the nodes in the path from the root to the modified subtree have to be re-evaluated. Next, we present the result of the incremental evaluation after such four modifications.

<table>
<thead>
<tr>
<th>Edit Action</th>
<th>Misses</th>
<th>Binding-Tree</th>
<th>Time</th>
<th>Misses</th>
<th>Visit-Tree</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Hits</td>
<td>Tests</td>
<td></td>
<td>Hits</td>
<td>Tests</td>
</tr>
<tr>
<td>Add a statement</td>
<td>13</td>
<td>16</td>
<td>336</td>
<td>0.00</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>(function: beginning)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add a statement</td>
<td>434</td>
<td>432</td>
<td>14376</td>
<td>0.03</td>
<td>434</td>
<td>429</td>
</tr>
<tr>
<td>(function: end)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add a global variable</td>
<td>4072</td>
<td>1828</td>
<td>172234</td>
<td>0.34</td>
<td>3950</td>
<td>914</td>
</tr>
<tr>
<td>(at the beginning)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add a global variable</td>
<td>3874</td>
<td>1949</td>
<td>165674</td>
<td>0.26</td>
<td>3752</td>
<td>1034</td>
</tr>
<tr>
<td>(at the end)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As expected, the incremental evaluators handle local changes extremely well. The execution time in both evaluators is almost negligible. Note, however, the small increase in the number of misses when the local change occurs at the end of the input. On the contrary, adding a global variable gives poor incremental behaviour. Although it gives poor results, the evaluators perform slightly better if the modification occurs at the end of the input. Note that the evaluator is collecting the environment during its first traversal. As a result, only the visit-functions that collect the environment after adding the global variable are really affected by the modification. Thus, the evaluator reuses more visit-functions if the modification occurs at the end of the first traversal.

Observe also that the number of visit-functions computed after adding a global variable is similar to the number obtained when evaluation from scratch is considered. This tends to happen specially when the modification occurs at the beginning of the input program. In this case, the execution times obtained with incremental evaluation and evaluation from scratch are equal. No gain is obtained with the incremental evaluation since the same visit-functions are being computed. On the contrary, the exhaustive attribute evaluator achieves a better execution time, since it is not influenced by the interpretative overhead due to the memoization scheme.

### 7.3.2 Projection of Attributes

The incremental behaviour of the \texttt{MICROC} environment can be greatly improved if we consider the grammar transformation presented in Section 6.7. Thus, we have transformed the AG in order to project the attribute that defines the environment passed to the body of the \texttt{MICROC} functions. The environment is projected on the list of identifiers used
by the functions. This transformation induces an additional traversal to the body of the functions, \textit{i.e.}, the traversal that collects the used identifiers. As a result, the bodies of the MICROC programs are now being decorated in two traversals. The next table shows the results obtained.

<table>
<thead>
<tr>
<th>Edit Action</th>
<th>Binding-Tree</th>
<th>Visit-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Misses</td>
<td>Hits</td>
</tr>
<tr>
<td>\textit{Add a statement}</td>
<td>17</td>
<td>23</td>
</tr>
<tr>
<td>(function: beginning)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{Add a statement}</td>
<td>443</td>
<td>441</td>
</tr>
<tr>
<td>(function: end)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{Add a global variable}</td>
<td>852</td>
<td>704</td>
</tr>
<tr>
<td>(at the beginning)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{Add a global variable}</td>
<td>644</td>
<td>813</td>
</tr>
<tr>
<td>(at the end)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The projection of the environment drastically improved the performance of both incremental evaluators after a global modification in the input. The number of cache misses and equality tests decreased considerably. As a result, using the projection of attributes, the incremental evaluator gives a speed-up of 2 for global changes when compared to the exhaustive evaluator. Local changes keep being handled extremely well.

Let us analyse in more detail the performance of the visit-tree based attribute evaluator. The projection of attributes is implemented in the MICROC AG as a semantic function, and not by attribution. Consequently, the projection function is computed non-incrementally. The effects of this can be seen in the above table by comparing the performance of the visit-tree evaluator in the case of adding a statement to function occurring at the end of the program with adding a global variable at the end of the program (second and fourth row, respectively). The number of visit-functions evaluated do not differ much (432 versus 522), but the execution times do (0.03 versus 0.10). The difference in the execution time is due to the evaluation of the semantic functions which project the attributes: after the change to a global variable the environment is projected in all the functions of the input program.
Chapter 8

Conclusions

This thesis discussed purely functional implementations of attribute grammars. We have presented techniques to implement attribute grammars in both a strict and a lazy functional programming language. In Chapter 4 we have introduced two new strict, functional implementations for attribute grammars: firstly, we have defined the visit-tree based attribute evaluators, which use a visit-tree data structure to glue together the different traversal functions of the evaluators. Secondly, we have introduced deforested attribute evaluators, which are data type free attribute evaluators. We presented also a deforested implementation for lazy attribute evaluators.

We have focused our research on the strict, functional attribute evaluators and on their incremental evaluation. In Chapter 6 we presented techniques for incremental attribute evaluation based on standard function memoization. The visit-tree based attribute evaluators improve the incremental behaviour of previous functional approaches in two ways: they reduce the number of memoized functions that have to be computed, and they induce a smaller interpretative overhead, since their visit-functions do not rely on any additional arguments to convey information between different traversals. Furthermore, we have introduced a new incremental evaluation scheme, i.e., the decoration of shared call-trees, that appears to reduce considerably the interpretative overhead due to memoization. We have also presented several techniques for improving the incremental behaviour of the evaluators. The projection of attributes does improve the incremental behaviour considerably. The first experimental results, presented in Chapter 7, are encouraging.
In Chapter 5, we introduced generic attribute grammars which are an extension to the classic attribute grammar formalism. This extension provides a component based style of programming within the attribute grammar formalism. Generic attribute grammar components are parameterized with grammar symbols and semantic functions. Such components are analysed and compiled independently.

8.1 Future Work

We have presented a transformation for attribute grammars which projects attributes to improve the incremental behaviour of the attribute evaluators. Such a transformation, however, should be detected by an attribute grammar analysis, and the grammar should automatically be “projected”. Such analysis deserves more study.

We have also studied purely functional attribute evaluation and their incremental evaluation model. Another attractive model to speedup attribute evaluation is to use parallel evaluation. Although we have included a parallelism detector, which annotates the Haskell based evaluators with parallel directives, we did not focus on parallel evaluation. Functional programming in general, and, our purely functional attribute evaluators in particular, seem amenable to parallel evaluation: the evaluators are side-effect free. Besides, our attribute evaluators do not rely on any possible large number of data structure used to convey information between different traversal functions, like in previous functional approaches. The use of parallelism together with incremental evaluation is still an open problem.

Compression algorithms can improve their compression rate by using knowledge about the structure of the data to compress. These data compressors, rather than compressing the textual data, compress the abstract syntax tree that represents the structure of such a data. In our model of attribute evaluation, a very efficient representation of abstract syntax trees is used: the syntax trees are collapsed into minimal direct acyclic graphs, due to the hash-consing scheme. Thus, this seems an appealing application for LRC: based on an attribute grammar specification of the language, an efficient data compressor can be automatically generated. Such a program would use the DAG representation of the input to achieve efficient data compression.

The LRC system is now an easy-to-use system, which produces language-based environments that have powerful and modern graphical user interfaces. Moreover, the front-end of LRC system is written in LRC itself. However, an advanced interactive tool for LRC, written in LRC, of course, is still missing. In fact, a language-based environment for LRC is a major lack of the system.

8.2 Final Remarks

We stated in the very first chapter of this thesis that attribute grammars and lazy functional programming are closely related. In fact, writing an attribute grammar can be
viewed as writing a lazy functional program. In this thesis we discussed several examples where the attribute grammar formalism yields correct and elegant solutions, and the functional program yields counterintuitive, complex and large solutions.

Indeed, we are convinced that the style of programming with attribute grammars helps the programmer to construct better functional programs. Thus, the question that arises immediately is whether it would be possible to incorporate the elegant style of attribute grammar writing directly within a functional programming language. Recent developments based on the concept of extensible records are working in this direction [dMPJvW99]. We hope that the continuing developments on functional languages and attribute grammars will make it possible to have functional programming languages supporting the attribute grammar style of programming. Would it not be nice if we could have functional programs analyse and transform themselves into the strict, deforested attribute evaluators presented in this thesis?

Finally, we expect recent advances in language-based environments to allow the writing of programs to become even more agreeable and effective.


Index

Δ, 122, 153
block, 20, 36, 38, 40, 65, 126, 161
λ-attribute evaluators, 83
abstract attribute grammar, 157
abstract context-free grammar, 13, 14, 17, 22
abstract structure, 14
abstract grammar, 14, see abstract context-free grammar, 22
abstract syntax tree, 14, 22
generic, 104
accumulating parameters, 148
ACFG, 14
affected, see Δ
AG, 15
alphabet, 12
arranged orderly attribute grammars, 65
AST, 14
attributable attribute, 31, 33, 34
generated attribute, 35, 37
attribute, 15, 16, 34
defined, 17
higher-order, 33
inherited, 15, 34, 104
input, 15
instances, 17
local, 15, 34, 104
occurrence, 15, 16, 104
output, 15
synthesized, 15, 34, 104
used, 17
attribute dependency graph, 18, 29
attribute equation, see semantic equations
attribute evaluation, 17
attribute evaluator, 18, 31
binding-tree, 69, 123, 165
deforested, 83, 139, 158
exhaustive, 122
generic, 103
incremental, 122, 129, 130
visit-tree, 76, 133, 158, 165
attribute grammar, 15
arranged orderly, 65
circular, 19
complete, 17
declarative, 15
equivalent, 18, 35
first-order, 33, 123
genetic, 104
higher-order, 33, 34, 156
L-ordered, 62, 65, 109
non-circular, 19
ordered, 63, 130
partitionable, 61, 110
well-defined, 18, 29, 35, 106
attributed tree, 17, 130
attribute rules, 15, 24
axiom, 12, 15, 21, 103, 115
Backus Naur Formalism, 21
binary shape-tree, 45
binding analysis, 69
binding-trees, 69
BNF, 21
Bochmann normal form, 17, 35, 63
cache hit, 127, 131, 136, 141
cache miss, 126, 127, 129
catamorphisms, 52
CCFG, 13
circular, 19
complete, 17
declarative, 15
equivalent, 18, 35
first-order, 33, 123
genetic, 104
higher-order, 33, 34, 156
L-ordered, 62, 65, 109
non-circular, 19
ordered, 63, 130
partitionable, 61, 110
well-defined, 18, 29, 35, 106
attributed tree, 17, 130
attribute rules, 15, 24
axiom, 12, 15, 21, 103, 115
Backus Naur Formalism, 21
binary shape-tree, 45
binding analysis, 69
binding-trees, 69
BNF, 21
Bochmann normal form, 17, 35, 63
cache hit, 127, 131, 136, 141
cache miss, 126, 127, 129
catamorphisms, 52
CCFG, 13
CFG, 12
Chained Scheduling Algorithm, 65, 158
change propagator algorithm, 122, 123
changed, 122
circular attribute grammar, 19, 56
circular program, 49, 52, 56, 59, 65, 157

circular programs, 157
deforested, 88
strictification, 58
well-formed, 57
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32
circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17
circular programs, 157
components, 32

circular program, 49, 52, 56, 59, 65, 157
circularity test, 19, 58, 158
complete attribute grammar, 17
complete context-free grammar, 13, 17

factorized binding-trees, 69
first-order attribute grammar, 33, 37, 123
flow type, 111, 117
function cache, 126, 127, 131, 134, 136
   137, 149, 151, 152, 161
fuse, 30

GAE, 112
GCFG, 103
GENAG, 102
generalized fold operators, 52
generic attribute evaluator, 103
generic attribute grammar, 104
generic attribution rules, 104
generic context-free grammar, 104
generic semantic function, 105
generic symbol, 102, 103, 106, 108, 112

HAG, 34
hash-consing, 124
higher-order
   attribute, see attributable attributes, 130
   attribute grammar, see attribute grammar
   attribution rules, 34
   reducing, 38
   trees, 33, 38, 130
ill-defined attribute grammar, 19
inconsistent, 122
incremental attribute evaluator, 122
change propagator, 122
optimal, 122
visit-function memoization, 123
incremental evaluator engine, 156, 160
input, 35
inter-traversal attribute dependency, 68
interface, 61
ITAD, 68

Kastens’s ordered scheduling algorithm, 158
L-ordered attribute grammar, 57, 62, 65, 109, 110

DAG, 124
data constructor memoization, 160
decorated tree, 18, 122, 143
deforested, 73
deforested visit-function, 84
depends, 17
derivation tree, 14
direct acyclic graphs, 124
directly derives, 12
equal, 122
equivalent attribute grammar, 18
exhaustive attribute evaluator, 122, 132
   168
INDEX

language, 12, 20, 107, 165
leaf-trees, 44
least fixed point, 57
Least-Recently-Used, 151
left-hand side symbol, 12, 15, 26, 29, 117
lexical analyser, 13
LRC, 48, 137, 155
LRU, 151, 161
meaning, 18, 19, 56, 108, 157
memo entry, 125
memo table, 125, 139, 142, 144
memoized functions, 122
newborn, 122
non-circular, 19
non-circular attribute grammar, 19
non-terminal symbol, 12
  accessible, 12
  derivable, 12
  instance, 15
  occurrence, 12
non-terminals, see non-terminal symbol
OAG, 64
obsolete based purging strategy, 152
optimal incremental attribute evaluator, 122
ordered attribute grammar, 64, 158
output, 35
parse tree, 14
parse-time attribute evaluation, 96
parser combinators, 39
parsers, 13
parsing functions, 39
partitionable attribute grammars, 61
production, 12
  ϵ-production, 12, 21, 150
  applied on, 12, 14
  instance, 15
  left-hand side symbol, 12
  right-hand side symbols, 12
  terminal production, 12

program slice, 159
pseudo-terminal symbol, 22, 75, 108
recursive descendent parsers, 39
reduce, 90
regular expressions, 13
repmin, 44
repminerror, 53
retained, 122
right-hand side symbols, 12, 15, 26, 29
runtime system, 156
scanner, 30
semantic domain, 15, 34
semantic equation, 15, 16, 25, 34, 37, 105, 117
semantic function, 16
semantic function processor, 157
sentence, 12, 18, 20
split-trees, 138
start symbol, see axiom, 103
static parameters, 105
strictification of circular programs, 58
syntactic compositionality, 32, 102
syntactic reference, 16, 26
term cache, 126, 128, 132, 135, 152, 160
term constructor function, 13
terminal production, 150
terminal symbol, 12
  literal symbols, 13, 41
  occurrence, 12
  pseudo-terminal symbols, 13
terminals, see terminal symbol
tree decoration, see attribute evaluation
type I circularity, 63
type II circularity, 63
type III circularity, 65
types, 13, 15, 34
ULE, 153, 161
undecorated tree, 17
unshared tree, 130
upward remote attribute occurrences, 29
INDEX

Used in Last Evaluation, 153
visit-function generator, 155
visit-function memoization, 160
visit-functions, 69, 123, 125, 126, 158
visit-sequences, 66, 74, 113, 158
visit-sub-sequences, 66, 158
visit-tree, 76
    attribute evaluator, 76
    data types, 77
visualization engine, 156
vocabulary, 12
well-defined attribute grammar, 18, 62
well-formed lazy evaluator, 57, 110
Samenvatting

Dit proefschrift behandelt de specificatie en de efficiënte implementatie van programmeeromgevingen die het schrijven van programma’s in een of andere programmeertaal ondersteunen. Kennis over een taal, die is ingebouwd in een dergelijke omgeving, kan de gebruiker ondersteunen in het programmeerproces. In dit soort kennis vallen twee aspecten te onderscheiden: kennis van de structuur van de taal en kennis van de betekenis van een taal. Hiervan gebruik makend kan de programmeeromgeving eigenschappen van programma’s berekenen en aan de gebruiker presenteren. Daarnaast kan ook aangegeven worden waar de gebruiker zich niet aan de taaldefinitie houdt, en dus bezig is een illegaal programma te construeren. Een gevolg van het gebruik van een taalspecifieke programmeeromgeving is dat dit niet alleen de correctheid van de geconstrueerde programma’s ten goede komt, maar ook nog eens het productieproces aanzienlijk versnelt.

Omdat er zeer veel verschillende programmeertalen bestaan is het ondoenlijk voor elk van deze talen met de hand een programmeeromgeving uit te programmeren, en we proberen dus een dergelijke omgeving te genereren uit een formele beschrijving van de programmeertaal. Formeel is een taal gedefinieerd als een verzameling zinnen, maar het systeem baseren op louter een opsomming van deze zinnen is een zinloze exercitie; we hebben een formalisme nodig waarin we dergelijke, in principe onbegrensd, verzamelingen kunnen beschrijven; Attributengrammatica’s bieden deze mogelijkheid. Een attributengrammatica bestaat uit een context-vrije grammatica en een verzameling attribuutdefinities en regels hoe deze attributen te berekenen. De context-vrije grammatica legt de structuur van de taal vast, en de attributen worden gebruikt om een betekenis toe kennen aan een zin uit die taal. Uitgaande van een dergelijk formalisme komt de vraag op hoe we hieruit op een efficiënte manier een programmeeromgeving kunnen genereren. Een belangrijk aspect hierbij is dat programmeeromgevingen interactieve systemen zijn die “real-time” reageren op acties van de gebruiker. Omdat de gebruikte analyses van het programma onder constructie vaak duur zijn, is het van belang dat dergelijke systemen niet telkens opnieuw een volledige analyse uitvoeren, maar daarbij zoveel mogelijk hergebruiken van wat al bekend is uit eerdere analyses.

Dit proefschrift behandelt nieuwe technieken om attributengrammatica’s te implementeren, en wel gebruik makend van het functie begrip. Een belangrijke eigenschap van een zuiver functionele implementatie is dat incrementeel uit te rekenen specificaties gemakkelijk zijn af te leiden uit een attributengrammatica. Een incrementele attributen-grammatica-evaluator is in staat alle eerder berekende informatie, die niet ongeldig is
gemaakt door de laatste gebruikersinteractie, opnieuw te gebruiken. In de gekozen zuiver functionele benadering bestaat een evaluator uit een verzameling functies, en wordt de gewenste incrementaliteit bereikt door aanroepen van deze functies te memoreren; dat wil zeggen dat de functies de argumenten waarop ze ooit zijn toegepast onthouden, samen met het toen berekende resultaat.

Vrijwel altijd beschrijven attributengrammatica’s algoritmen die een aantel keren over de ontleedboom, die de door de gebruiker gegeven input representeert, lopen. In een zuiver functionele implementatie, gebaseerd op strikte functies, dienen de functies informatie tussen de verschillende bezoeken aan elkaar over te dragen. Eerdere implementaties langs deze weg gebruikten een groot aantal tussenliggende datastructuren om deze informatie te presenteren. In dit proefschrift worden twee nieuwe manieren geïntroduceerd waarop deze informatie doorgegeven kan worden: de eerste introduceert een zogenaamde visit-tree als verbindend element tussen de verschillende bezoeken aan een knoop, terwijl de tweede, gebruik makend van de ontbossings-techniek het gebruik van deze tussenliggende structuren geheel onzichtbaar maakt. De uiteindelijke evaluatoren maken geen gebruik meer van speciale data types. Een belangrijke eigenschap van de gebruikte afbeeldingen is dat er geen extra argumenten en resultaten toegevoegd worden aan de functies die de attributen uiteindelijk uitrekenen. Een gevolg hiervan is dat we in onze evaluatoren alleen maar te maken hebben met een kleine overhead tengevolge van het gebruikte memoisatie schema voor functieaanroepen. Tenslotte presenteren we een aantal technieken om het incrementele gedrag en de functionele implementaties daarvan verder te verbeteren.

De afwezigheid van expliciete data types maakt dat de resulterende implementaties in hoge mate modular zijn. Teneinde hiervan gebruik te maken introduceren we zogenaamde generieke attributengrammatica’s. Een generieke attributengrammatica is een component die ontworpen is om gemakkelijk opnieuw gebruikt te worden, gemakkelijk is samen te stellen met andere componenten, en gemakkelijk te begrijpen is. We geven aan hoe generieke attributengrammatica’s afgebeeld kunnen worden op de eerder geïntroduceerde functionele implementaties. Omdat een dergelijke component geen data types importeert of exporteert kunnen we een dergelijke component ook stapsgewijs uitbreiden zonder een globaal geïntroduceerd data type aan te hoeven passen.

Als onderdeel van het onderzoek naar geschikte incrementele implementaties hebben we de gebruikte technieken geïmplementeerd in het LRC systeem. In dit proefschrift geven we de eerste experimentele resultaten bij gebruik van de incrementele module van dat systeem.
Curriculum Vitæ

João Alexandre Baptista Vieira Saraiva

21 September 1967  Born in Póvoa de Lanhoso, Portugal

             University of Minho, Portugal

1991 – 1993  Master’s degree in Computer Science
             University of Minho, Portugal

1993 – 1999  Teaching Assistant
             University of Minho, Portugal

1995 – 1999  Ph.D. Student (on leave from University of Minho)
             University of Utrecht
Titles in the IPA Dissertation Series

*The State Operator in Process Algebra*
**J. O. Blanco**
Faculty of Mathematics and Computing Science, TUE, 1996-1

*Transformational Development of Data-Parallel Algorithms*
**A. M. Geerling**
Faculty of Mathematics and Computer Science, KUN, 1996-2

*Interactive Functional Programs: Models, Methods, and Implementation*
**P. M. Achten**
Faculty of Mathematics and Computer Science, KUN, 1996-3

*Parallel Local Search*
**M. G. A. Verhoeven**
Faculty of Mathematics and Computing Science, TUE, 1996-4

*The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*
**M. H. G. K. Kesseler**
Faculty of Mathematics and Computer Science, KUN, 1996-5

*Distributed Algorithms for Hard Real-Time Systems*
**D. Alstein**
Faculty of Mathematics and Computing Science, TUE, 1996-6

*Communication, Synchronization, and Fault-Tolerance*
**J. H. Hoepman**
Faculty of Mathematics and Computer Science, UvA, 1996-7

*Reductivity Arguments and Program Construction*
**H. Doornbos**
Faculty of Mathematics and Computing Science, TUE, 1996-8

*Functorial Operational Semantics and its Denotational Dual*
**D. Turi**
Faculty of Mathematics and Computer Science, VUA, 1996-9

*Single-Rail Handshake Circuits*
**A. M. G. Peeters**
Faculty of Mathematics and Computing Science, TUE, 1996-10

*A Systems Engineering Specification Formalism*
**N. W. A. Arends**
Faculty of Mechanical Engineering, TUE, 1996-11
Normalisation in Lambda Calculus and its Relation to Type Inference
P. Severi de Santiago
Faculty of Mathematics and Computing Science, TUE, 1996-12

Abstract Interpretation and Partition Refinement for Model Checking
D. R. Dams
Faculty of Mathematics and Computing Science, TUE, 1996-13

Topological Dualities in Semantics
M. M. Bonsangue
Faculty of Mathematics and Computer Science, VUA, 1996-14

Algorithms for Graphs of Small Treewidth
B. L. E. de Fluiter
Faculty of Mathematics and Computer Science, UU, 1997-01

Process-algebraic Transformations in Context
W. T. M. Kars
Faculty of Computer Science, UT, 1997-02

A Generic Theory of Data Types
P. F. Hoogendijk
Faculty of Mathematics and Computing Science, TUE, 1997-03

The Evolution of Type Theory in Logic and Mathematics
T. D. L. Laan
Faculty of Mathematics and Computing Science, TUE, 1997-04

Preservation of Termination for Explicit Substitution
C. J. Bloo
Faculty of Mathematics and Computing Science, TUE, 1997-05

Discrete-Time Process Algebra
J. J. Vereijken
Faculty of Mathematics and Computing Science, TUE, 1997-06

A Functional Approach to Syntax and Typing
F. A. M. van den Beuken
Faculty of Mathematics and Informatics, KUN, 1997-07

Ins and Outs in Refusal Testing
A.W. Heerink
Faculty of Computer Science, UT, 1998-01

A Discrete-Event Simulator for Systems Engineering
G. Naumoski and W. Alberts  
Faculty of Mechanical Engineering, TUE, 1998-02

*Scheduling with Communication for Multiprocessor Computation*  
**J. Verriet**  
Faculty of Mathematics and Computer Science, UU, 1998-03

*An Asynchronous Low-Power 80C51 Microcontroller*  
**J. S. H. van Gageldonk**  
Faculty of Mathematics and Computing Science, TUE, 1998-04

*In Terms of Nets: System Design with Petri Nets and Process Algebra*  
**A. A. Basten**  
Faculty of Mathematics and Computing Science, TUE, 1998-05

*Inductive Datatypes with Laws and Subtyping – A Relational Model*  
**E. Voermans**  
Faculty of Mathematics and Computing Science, TUE, 1999-01

*Towards Probabilistic Unification-based Parsing*  
**H. ter Doest**  
Faculty of Computer Science, UT, 1999-02

*Algorithms for the Simulation of Surface Processes*  
**J.P.L. Segers**  
Faculty of Mathematics and Computing Science, TUE, 1999-03

*Recombinative Evolutionary Search*  
**C.H.M. van Kemenade**  
LIACS, faculty of Mathematics and Natural Sciences, Leiden University, 1999-04

*Learning Reliability: a Study on Indecisiveness in Sample Selection*  
**E.I. Barakova**  
Faculty of Mathematics and Natural Sciences, RUG, 1999-05

*Schedulere Optimization in Real-Time Distributed Databases*  
**M.P. Bodlaender**  
Faculty of Mathematics and Computing Science, TUE, 1999-06

*Message Sequence Chart: Syntax and Semantics*  
**M.A. Reniers**  
Faculty of Mathematics and Computing Science, TUE, 1999-07