The Tree Processing Language

Defining the structure and behaviour of a tree

Name	E. Papegaaij
Email	e.papegaaij@alumnus.utwente.nl
Supervisors	dr. ir. Theo C. Ruys
	ir. Philip K.F. Hölzenspies
	dr. ir. Arend Rensink
Institute	Unversity of Twente
Chair	Formal Methods and Tools

Enschede, March 4, 2007

Abstract

Tree structures are commonly used in many applications. One of these is a compiler, in which the tree is called an abstract syntax tree (AST). Different techniques have been developed for building and working with ASTs. However, many of these techniques are limited in their applicability, require major effort to implement or introduce maintenance problems in an evolving application.

This thesis introduces the Tree Processing Language, a language for defining the structure of a tree and adding functionality to this tree. The compiler TPLc is used to produce the actual class hierarchy implementing the specified tree. TPL provides a clear separation between the structure of a tree, a *tree definition*, and behaviour of a tree, *logic specifications*. Different aspects of the behaviour of a tree can be provided in separate logic specifications, allowing a clear separation of concerns.

TPLc generates a heterogeneous tree structure with strictly typed children. Functionality in a logic specification is specified using the inheritance pattern. To allow different inheritance trees in different logic specifications, the inheritance pattern is enhanced with multiple inheritance. For languages that do not support multiple inheritance, the inheritance pattern with composition is developed.

To prove the applicability of TPL, TPLc is written in TPL. When compared with an implementation in Java, this implementation provides a better separation of concerns and is easier to maintain.

Samenvatting

In veel systemen worden boom structuren gebruikt. Eén van deze is een vertaler, waarin de boom een abstract syntax tree (AST) wordt genoemd. Er zijn verschillende technieken ontwikkeld om ASTs te construeren en er mee te werken. Echter, veel van deze technieken zijn beperkt in hun bruikbaarheid, vereizen veel moeite om te gebruiken of introduceren onderhouds problemen in een veranderd systeem.

Deze scriptie introduceert the Tree Processing Language, een taal voor het definiéren van een boom en het toekennen van functionaliteit aan deze boom. De vertaler TPLc produceert de uiteindelijke klasse hiërarchie, die the opgegeven boom implementeert. TPL biedt een duidelijk scheiding tussen de structuur van een boom, een *tree definition*, en het gedrag van een boom, *logic specifications*. Verschillende aspecten van het gedrag van een boom kunnen in losstaande logic specifications gegeven worden. Dit staat een duidelijke scheiding van gedrag toe.

TPLc genereert een heterogene boom structuur met strict getypeerde kinderen. Functionaliteit in een logic specification wordt gespecificeert met behulp van het inheritance pattern. Het inheritance pattern wordt uitgebreid met meervoudige overerving om het mogelijk te maken om verschillende overervingsbomen te gebruiken in verschillende logic specifications. For talen die geen meervoudige overerving ondersteunen is het inheritance pattern met compositie ontwikkeld.

Om de bruikbaarheid van TPL te bewijzen is TPLc geschreven in TPL. Wanneer deze versie vergeleken wordt met een implementatie in Java blijkt dat de versie in TPL een betere scheiding van gedrag heeft en beter te onderhouden is.

Preface

Compiler construction has always been one of my favourite fields of software engineering. In the past few years I've written several parsers and compilers. Of these compilers, the compiler for the functional programming language Tina has been the most challenging. I used a hand-written heterogeneous abstract syntax tree as underlying data structure. The most important algorithm applied onto this AST, the transformation of Tina into a core lambda expression language, was written as part of these AST node classes. However, the overwhelming number of AST classes (almost 100) made this approach increasingly difficult to maintain when other algorithms (such as a lambda lifter) where added. At that moment, it became clear that a more structured approach was required. To keep the development of an application, based on a heterogeneous tree, maintainable, different algorithms needed to be separated in different files. The development of TPL is an attempt to provide such an environment.

When I first approached Theo C. Ruys, my premiere supervisor, for an assignment, I has no idea I would be solving this problem, which had bothered me for a long time. At first, ambitious as I was, I proposed to design and implement a completely new parser generator. Luckily, Theo slowed me down a bit and directed me to focus on the real problem: the heterogeneous AST.

For his help in concreting the features of TPL, reading and correcting this thesis and his patience during the endless discussions we had last year, I would like to thank Theo C. Ruys, my premiere supervisor. His guiding helped me structure my thoughts, to be able to write them down. I would also like to thank Philip K.F. Hölzenspies for his help in writing and formatting this thesis. His knowledge of the English language has proven to be far better than mine. Last, but not least, I would like to thank Arend Rensink for having taken the time to examine this thesis.

Emond Papegaaij
Enschede, March 4, 2007

Contents

List of Figures

List of Tables

List of Grammar Fragments

List of Listings

Chapter 1

Introduction

Tree structures have been, and probably will be for a considerable time in the future, a widely used way of organising and working with data. Tree structures are used to represent the structure of an input file (concrete and abstract syntax trees), user interface components, the representation of HTML pages (the document object model), XML and many more. Due its wide acceptance, extensive research has been spent on working with tree structures.

This thesis is placed in the context of working with tree structures in an object-oriented programming environment. The main focus is on defining the runtime organisation of the tree and applying algorithms on this structure. The origin of the tree—the system responsible for constructing the tree structure—and the actual construction of the tree are discussed, but fall outside the main research area.

In this chapter, an introduction on compiler construction is given, in §1.1. This section shows how an abstract syntax tree is acquired, and what the typical operations are that need to be performed on an AST. §1.2 describes the problem statement of this thesis. Finally, the outline of this thesis is given in 1.3.

1.1 Compiler Construction and Abstract Syntax Trees

A multi-pass compiler performs the compilation of a source file in several stages. These stages will be discussed in this section. Compilation starts with reading a source file, and recognising the syntax of the input. Next, an abstract representation of this input is constructed. This is the abstract syntax tree. This AST is used in subsequent phases to perform context checking and code generation. More complex compilers might have more phases, such as optimisers.

Abstract syntax trees are also commonly used in other disciplines, such as communication (eg. a web browser) and source code refactoring in an integrated development environment (IDE) [Eclipse, 2007]. It is also possible that the abstract syntax tree is not the result of a parser reading an input file, but from speech, or from a graphical programming language. However, the most common usage is a compiler, which reads an input language.

1.1.1 Lexical Analysis and Parsing

In the first stage, the lexical analysis, the compiler reads the input file and produces a stream of tokens. Every token corresponds to a fragment, or construct, found in the input file, such as identifiers, literals, operators and keywords. These tokens are fed to a parser, which discovers (and checks) the structure of the input.

Writing a lexer (or scanner) and parser by hand is tedious, difficult and error prone. Many programs have been developed, which assist the developer in writing the lexer and parser. These tools often take a syntax specification in (E)BNF,

and generate a lexer and parser from this specification. Therefore, these tools are commonly called parser generators. Some of these tools are mentioned in §2.3.

Different strategies exist, on how a parser matches the input language, such as LALR and recursive descent parsing. However, a discussion of these is beyond the scope of this thesis[1].

1.1.2 Construction of the AST

In a multi-pass compiler, the task of the parser is to record the structure of the parsed input in an abstract syntax tree. This tree contains all relevant information from the input. What exactly is relevant information, depends on the subsequent phases. Normally, tokens, such as comma's and brackets, are discarded. Also, nesting of parser production rules is removed.

AST construction is exemplified with the grammar presented in fragment 1.1. This grammar matches simple expressions with addition and multiplication. The actual values are represented by numbers and identifiers. Expressions can be nested with brackets.

Grammar Fragment 1.1 Simple Expression

⟨*expression*⟩	::=	⟨*term*⟩ ('+' ⟨*term*⟩)?
⟨*term*⟩	::=	⟨*atom*⟩ ('*' ⟨*atom*⟩)?
⟨*atom*⟩	::=	'(' ⟨*expression*⟩ ')'
	|	⟨*number*⟩
	|	⟨*identifier*⟩
⟨*identifier*⟩	::=	A sequence of letters.
⟨*number*⟩	::=	A sequence of digits.

This grammar matches sentences such as '1', '1 + 1' and '(1 + a) * b'. The parse tree of the sentence '3 + 5 * (a + b)' is given in figure 1.1. This figure shows how the complete sentence is matched as an ⟨*expression*⟩. The ⟨*expression*⟩ consists of a ⟨*term*⟩, followed by the literal '+', again followed by a ⟨*term*⟩. The left ⟨*term*⟩ is a simple ⟨*atom*⟩, which in turn is a ⟨*number*⟩. The right ⟨*term*⟩ consists of two ⟨*atom*⟩s, separated by a '*'. This process is continued until all tokens (the bottom line of the figure) are matched.

The parse tree clearly shows the structure of the parsed text, but this structure is not very practical to work with. If an interpreter for this grammar is needed, a set of four constructs is sufficient: addition, multiplication, numbers and identifiers. The `Add` node adds the results of the left and right operands. This node is created when a '+' is matched in ⟨*expression*⟩. The `Multiply` node multiplies the left operand with the right. It is created when a '*' is matched in ⟨*term*⟩. A `Number` node is created when a ⟨*number*⟩ is matched, and yields the value of the number. Finally, the `Identifier` node, which is created when an ⟨*identifier*⟩ is matched, resolves the value in a symbol table.

When this approach is taken, the sentence '3 + 5 * (a + b)' yields the abstract syntax tree shown in figure 1.2. From '3' and '5', two `Number` nodes are created, and two `Identifier` nodes from 'a' and 'b'. The nested addition is converted into

[1] An explanation of various parsing algorithms, such as LR(k) and LL(k), can be found in [Aho et al., 1986].

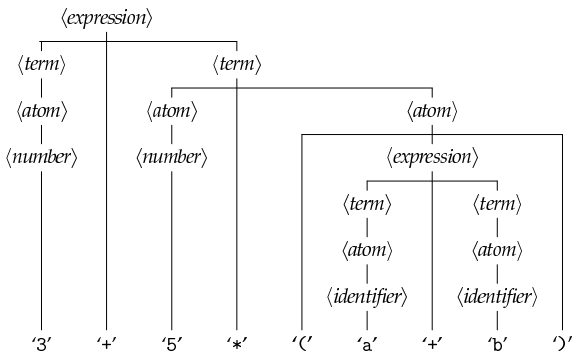

Figure 1.1: Example parse tree

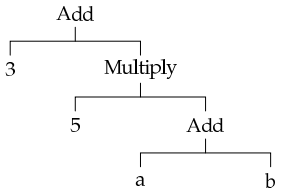

Figure 1.2: Example abstract syntax tree

an `Add` node. The brackets are discarded, because the structure of the tree itself is strong enough to indicate the nesting of `'a + b'`. Additional `Add` and `Multiply` nodes are created for the other addition and the multiplication. It is clear that the AST still follows the structure of the expression, but is not as verbose as the parse tree.

1.1.3 Context Checking and Code Generation

Parsing the input file and constructing the AST is only the first phase of the compilation process. The next phases typically consist of context checking, optimisation and code generation. These phases use the AST as underlying data structure.

With each phase, different demands are put on the AST. The context checker of a compiler often annotates the AST with additional information, such as types and references to the symbol table. The optimiser is likely to transform parts of the AST, to produce faster code with identical behaviour. Finally, the code generation reads the results of the preceding phases, to produce fast and correct code. These cases indicate, that it should be possible to add attributes to the AST nodes, transform parts of the tree and to traverse over the tree.

1.2 Problem Statement

Not only compilers use tree structures. Trees are a common data structure in many applications. Other examples are: component trees in a graphical user interface, most file systems and (simple) databases. Moreover, any data stored as XML can be represented as a tree structure. This can be the general purpose XML Document Object Model (DOM), but it is also possible to use a custom object structure to represent the data in the application. The requirements for working

with most of these trees are similar: they require attributes to be added to the tree nodes, and the nodes often need to contain behaviour.

In addition to the behaviour specific to a certain application, tree structure normally need functionality to construct the tree, to read the tree, for (string) serialisation and for copying. These operations are similar for different tree structures, but depend on the actual structure of the tree. A lot of effort is required to implement these operations, which need to be repeated for every tree structure. Also, these operations need to be kept in sync with the structure of the tree when this structure is modified.

A system is required to reduce the amount of work that needs to be done to implement a tree structure and to minimise maintenance problems. This system should consist of tool taking a specification and generating a tree structure. The tool should automatically generate functionality to construct, read, modify and copy the tree. The structure of the tree should be enforced using the type-system of the generation language. In addition to specifying the structure of the tree, it should be possible to specify behaviour of this tree (operations on the tree). A clear separation should be provided between different operations and between the operations and the structure. Finally, it should be possible to integrate the tool with a parser generator to construct the tree directly from a generated parser.

1.3 Outline

In this thesis, TPL and the tool TPLc are discussed. TPLc is the implementation of the tool described above. Below, the outline of this thesis is given.

Chapter 2. Related Work discusses techniques for working with tree structures that are currently available. It highlights the different organisations that are being used to design tree structures. It continues with a discussion on different techniques that have been developed to add functionality to a tree. Finally, a list of currently available parser generators and tree processors is given.

Chapter 3. The Tree Processing Language introduces TPL and gives the rationale behind this new language. The motivations for the decisions made in the design of TPL are explained. The architecture of the compiler TPLc is briefly touched. Also, an introduction on the language itself is given.

Chapter 4. Language Specification gives the specification of TPL. A formal specification of the syntax, in ENBF, is given, combined with an informal specification of the semantics.

Chapter 5. Design of TPLc presents the design of the compiler TPLc. The compiler is divided into six components, each of which is discussed.

Chapter 6. Conclusions and Future Work concludes this thesis. It discusses the contributions of the research, followed by some indications of future research.

Appendix A. Software Requirements Specification gives the requirements specification of TPL.

Appendix B. Testing explains the design of the testing framework used to test TPLc.

Appendix C. Improvements in the Final Version enumerates the improvements in the final version of TPLc over the prototype.

Appendix D. Visiting Methods and Attribute Grammars provides a discussion of the relation between visiting methods and attribute grammars.

Appendix E. Calculation Language Code Listings contains the code listings for the calculation language example, used in chapter 4.

Appendix F. Planning contains the original planning for the entire project.

Chapter 2
Related Work

The abstract syntax tree is a well known data structure, often used in compiler construction, but also in communication systems or integrated development environments. Due to its widespread use, many techniques exist for working with an AST. Some of these are supported by (or require) tools. These techniques, and there accompanying tools, are discussed in this chapter.

In this discussion, a distinction is made between the structure, or organisation, of an AST and the algorithms that are applied on an AST. First, an introduction of various, commonly used organisations of an AST are given in §2.1, followed by a discussion of various approaches of working with an AST in §2.2. Finally, §2.3 lists a set of tools, that are currently available for use in compiler construction.

2.1 The Organisation of an AST

Different organisations for an AST can be used. The two categories are the homogeneous AST and the heterogeneous AST. In the first, all nodes are of the same type (or data structure). In the second, different nodes can have different types. This section will discuss these categories, with their applications, advantages and disadvantages.

2.1.1 Homogeneous ASTs

In a homogeneous AST, every node has the same type (or, in an object-oriented environment, is of the same class). The major advantage of this approach is that only a single type needs to be developed. The popular parser generators ANTLR [Parr and Quong, 1995] and JavaCC [JavaCC, 2006] with JJTree [JJTree, 2006] all take this approach by default.

2.1.1.1 Using a Homogeneous AST

For a homogeneous AST, only a single node type needs to be developed. This simplicity is both its strength and its weakness. The single node type can be developed as part of a framework, which can be reused in different applications. This means no code has to be written to construct an AST, making it the ideal solution for simple applications, or when time is limited.

The node type, should at least be able to store the children of the node. This can be accomplished by adding a list structure as part of the node type, or the nodes themselves can form a linked list. In another design, a node contains a reference to its first child (if any), and a reference to its first sibling (if any). To be able to distinguish different parts of the tree, the node also contains a type field. For information such as identifiers, names and literals, the node type usually also contains a field, which can be filled with arbitrary data (such as a name, a number or a single character).

2.1.1.2 Problems

An AST is often decorated with additional information (attributes). This is especially useful in a multi pass compiler, to pass information from one stage to the next. To add attributes to nodes in a homogeneous AST, the node type will need support for these attributes. This is usually achieved by adding a dictionary to the node type. However, because all nodes are of the same type, this allows all nodes to contain attributes, even when they were not supposed to. Checking, whether a node is actually allowed to contain a certain attribute, can only be done at runtime, and requires additional code. Furthermore, the compiler can not guarantee type correctness of the attributes. Errors are difficult to detect, as they usually only surfaces when the erroneous attribute is used or set.

Another disadvantage is that algorithms can only be applied to the AST using a `switch` statement, as described in §2.2.1. Other techniques, such as the inheritance pattern (§2.2.2) and the visitor pattern (§2.2.3) depend on type polymorphism, which requires a heterogeneous AST.

A third problem surfaces when functionality for a node requires access to one or more children. For example, the context checker of a function declaration might need to extract the name of the function from the identifier node that represents its name. Cases like these are likely to be common, thus a solution, which allows reuse of code, is desirable. The code to extract the child node can be written in a utility method, which can be used throughout the application. However, these methods need to be kept in sync with the structure of the AST, introducing a maintenance hazard.

With a homogeneous AST, where all children are stored in a single list structure, it can also be difficult to cope with sub-structures with multiplicities other than 'exactly one'. When an optional subtree is omitted, all following subtrees shift one location towards the beginning of the list of children. A similar problem surfaces with lists. This makes it difficult to write the access methods mentioned before, because a search over the list of children is required, instead of a simple selection by index. It is also possible to restrict the structure of the AST, by disallowing optional nodes (and use a special 'absent node') and use container nodes for lists. However, this requires additional work during tree construction.

A homogeneous AST also limits the amount of type checking that can be performed by the compiler. For example, the compiler is unable to detect when the method written to retrieve the name of a function declaration is invoked with a variable declaration node supplied as argument. It is even possible that the method does not fail when given this node (for example, when it simply returns the first child). This makes these kinds of errors difficult to find and solve.

2.1.2 Heterogeneous ASTs

In a heterogeneous AST, different nodes can have a different type. In most heterogeneous trees, a distinct node type is used for every category of nodes, where a node category often relates to a production rule in the parser (see §1.1.2). Examples of such node types are: 'identifier', 'if statement' and 'method body'. This approach is taken by JJTree in 'multi' mode [JJTree, 2006], JTB [Tao and Palsberg, 2005], SableCC [Gagnon and Hendren, 1998] and most other tree builders, that build heterogeneous ASTs. ANTLR 3 [Parr, 2006] takes a slightly different approach. It allows any relation between node categories and node types, when a custom implementation of the `TreeAdaptor` (the component responsible for building the trees) is provided.

One of the most significant advantages of a heterogeneous AST is that different functionality and attributes can be provided for different node types. However,

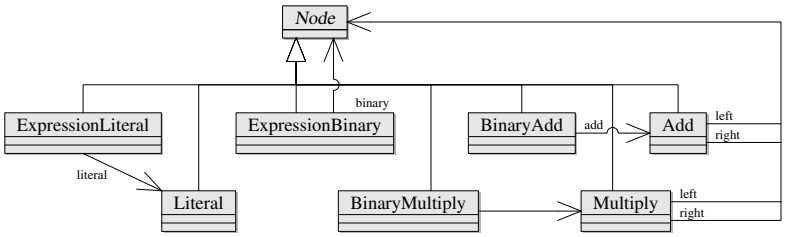

Figure 2.1: AST with a single inheritance level

Grammar Fragment 2.1 Expression example

⟨*expression*⟩	::=	⟨*literal*⟩
	|	⟨*binary*⟩
⟨*binary*⟩	::=	⟨*multiply*⟩
	|	⟨*add*⟩
⟨*literal*⟩	::=	A sequence of digits.
⟨*multiply*⟩	::=	⟨*expression*⟩ '*' ⟨*expression*⟩
⟨*add*⟩	::=	⟨*expression*⟩ '+' ⟨*expression*⟩

the large number of types can lead to maintenance problems.

This section first describes different designs used for heterogeneous ASTs. The biggest difference between these designs is the number of inheritance levels. This is followed by a discussion of the advantages of a heterogeneous AST over a homogeneous AST. Finally, the disadvantages of a heterogeneous AST are discussed.

2.1.2.1 Single Inheritance Level

In a AST organisation with a single inheritance level, all node classes directly extend from one common class, as displayed in figure 2.1, which shows the class organisation for grammar 2.1. Every node class represents an alternative from the grammar specification. This is the default design chosen by JJTree [Weatherley, 2002] in 'multi' mode (in which case all node classes directly extend from `SimpleNode`) and JTB [Tao and Palsberg, 2005] (where all node classes implement the `Node` interface and extend from `Object`).

The major disadvantage of this technique is that it makes little use of the power of inheritance. Although it is possible to add attributes and functionality to the node classes, similar node classes can not inherit these attributes and functionality. This can result in code duplication and makes the code difficult to maintain in a large application.

Another problem is that it is impossible to provide typed access to children when a production rule has several alternatives. Consider the grammar displayed in fragment 2.1. The classes corresponding to the production rules ⟨*multiply*⟩ and ⟨*add*⟩ should contain two references to an ⟨*expression*⟩. However, an ⟨*expression*⟩ can be a ⟨*literal*⟩ or ⟨*binary*⟩. The type of the reference, therefore, has to be an intersection of all these types. The only class satisfying this intersection is the common base class `Node`, extended by all nodes. This makes it impossible to distinguish between an expression and one of the other nodes. The same problem

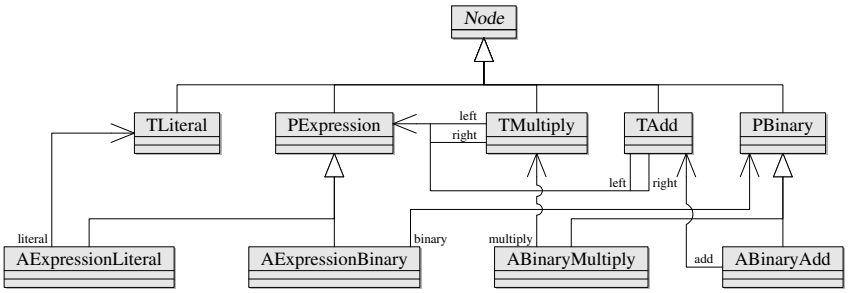

Figure 2.2: AST with inheritance on alternatives

occurs with the reference to a ⟨*binary*⟩ from `ExpressionBinary`.

2.1.2.2 Inheritance on Alternatives

A design commonly used by tools, which automatically generate the AST classes, is based on inheritance on alternatives. As with a single inheritance level, all nodes still extend from a single common node, but another level of inheritance is added. The first level of types represents the production rules, whereas the second level of types represents the alternatives of those production rules. Classes representing terminals (or tokens) are sometimes generated as direct subclasses of the common node. This is shown in figure 2.2, which shows the node classes for grammar fragment 2.1, with three terminals and two production rules, both with two alternatives. Other tools let terminals extend from a common terminal class, which is a subclass of the common root node class.

Inheritance on alternatives is used to resolve the problem described earlier and illustrated with grammar 2.1. It allows the construction of a strictly-typed AST. Both `left` and `right` references for `TMultiply` and `TAdd` are now references to the class `PExpression`, representing the ⟨*expression*⟩ production rule, and is extended by all classes representing the alternatives for this production rule. `AExpression-Binary` receives a reference to `PBinary`.

Inheritance on alternatives has a great resemblance with algebraic data types, where it is known as a recursive sum of products. The alternatives are the product types, with recursive references to the production rules, or sum types. Wang et al. [1997] describe how a definition in an 'abstract syntax description language' is compiled to several target languages. For the function language ML, they generate algebraic data types. The output for the object-oriented Java language uses a variation of inheritance on alternatives, without a common root node class. For C, a set of of `structs`, which contain a `union` over all alternatives, is generated.

The tools SableCC, described by Gagnon and Hendren [1998], JJForester, by Kuipers and Visser [2003], and ApiGen, by Van den Brand et al. [2005], all generate ASTs with inheritance on alternatives.

Classgen [Klein and Brandl, 2003] also generates trees with inheritance on alternatives. However, unlike the other tools, classgen can also generate product types that are not part of a sum. These classes directly extend from the common root class, without an intermediate level of inheritance. This is used for production rules with only a single alternative.

A disadvantage of inheritance on alternatives is that the inheritance tree is fixed. This makes it difficult to share attributes and functionality between nodes. Consider the case where common functionality is needed for all binary expressions. This can be accomplished by creating a ⟨*binary*⟩ production rule, for

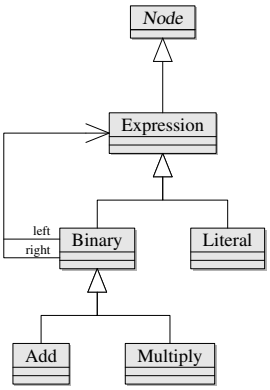

Figure 2.3: AST with custom defined inheritance

which all binary expressions are alternatives[1], and implementing the functionality as part of the type that corresponds to this production rule (`PBinary` in this case). However, this technique cannot be applied when functionality needs to be shared by all expressions. For example, when an instance variable is added to `PExpression`, which contains the type of the expression, then this variable cannot be easily accessed from `TLiteral`. Access to the variable needs to be provided by `AExpressionLiteral` when the `literal` reference is visited. This can be achieved by passing the variable itself, which will become cumbersome when multiple variables are required and does not work when `Literal` also needs access to methods in `PExpression`, or by passing a reference to `PExpression`, which results in encapsulation breaching—public accessor methods need to be provided for the attributes and the required methods also need to be declared public.

2.1.2.3 Custom Defined Inheritance

With the previous two techniques, the inheritance structure of the AST is determined by the tool generating the AST classes. With custom defined inheritance, the developer is free to specify the inheritance tree by hand. A possible inheritance tree for the grammar in fragment 2.1 is given in figure 2.3.

An obvious disadvantage is that this requires additional work from the developer. However, using a custom defined inheritance tree, it is possible to solve the problem with the expressions, illustrated in the previous section. The class `Literal`, representing a ⟨*literal*⟩ can now be defined as a subclass of the class for ⟨*expression*⟩, which gives it access to the instance variables and methods of `Expression`.

When a custom defined inheritance tree is used together with the inheritance pattern (§2.2.2), the developer is able to specify functionality at every level of inheritance. For example, when the inheritance tree of figure 2.3 is used, functionality can be specified in `Expression`, which is shared over all expressions. It is also possible to specify behaviour for just the binary expressions, or just for `Add`.

Custom defined inheritance is less common than the other designs, probably because of the added complexity. Treecc [Weatherley, 2002] allows the developer to specify any inheritance tree, although the syntax used is somewhat cryptic.

[1]Naturally this is only possible if the tool gives the developer the freedom to move all binary expressions under a single production rule. For example, in cases where the parser is used to indicate operator precedence, it is often not possible to move all binary expressions under a single rule, without loosing the operator priority.

2.1.2.4 Problems with a Heterogeneous AST

Writing the classes for a heterogeneous AST by hand is a tedious task, prone to errors. AST nodes often all need functionality such as accessor methods for the children, string serialisation and constructors with the children as arguments. Many of these need to be kept in sync with the structure of the AST. When changes to the structure of the AST are common, an error is easily made.

Also, it is difficult to utilise the heterogeneous structure of the AST, without affecting the maintainability of the system adversely. A single algorithm is likely to operate over several nodes. When the functionality for such an algorithm is added directly to the node classes (the inheritance pattern, §2.2.2), it will be spread across several files. Also, the code will be mixed with code for other algorithms and general node functionality. This makes it difficult to get a good overview of the code, and makes it hard to maintain. It is possible to implement algorithms as visitors [Gamma et al., 1995]. This moves all code and attributes, related to a single algorithm, to a single file. However, the visitor pattern also has its disadvantages, as described in §2.2.3.

2.1.3 Homogeneous versus Heterogeneous

Heterogeneous ASTs have a clear advantage over homogeneous ASTs, in that they allow the developer to define code for individual node classes. This solves the problems with adding and using attributes (see §2.1.1.2 on page 8) trivially. The solutions to the other limitations of a homogeneous AST will be explained here.

2.1.3.1 Type Polymorphism and Inheritance

A heterogeneous AST allows the use of type polymorphism. This makes it possible to use the inheritance pattern (§2.2.2) and the visitor pattern (§2.2.3), where with a homogeneous AST only the `switch` statement (§2.2.1) can be used.

Type polymorphism in object-oriented environments is based on subtyping. Therefore, the applicability of type polymorphism strongly depends on the design of the AST. A heterogeneous AST based on a single inheritance level only provides two levels of types: the common root node class and all other node classes. This requires operations to be specified on all node classes, or a single node. Although this is sufficient for the visitor pattern, it limits the usefulness of the inheritance pattern. A similar problem occurs with an AST based on inheritance on alternatives, where operations can only be specified per alternative, per production rule or for all node classes. When a custom defined inheritance tree is used, it is possible to use type polymorphism at every level of the inheritance tree. This makes it possible, for example, to specify operations on expressions, but also on binary expressions, statements, etc.

Inheritance allows reuse of functionality between similar nodes. However, as with type polymorphism, this is best used with a custom defined inheritance tree.

2.1.3.2 Selection of Child Nodes

It became apparent already that a heterogeneous AST with a single inheritance level is not strong enough for strictly typed access to children (this is illustrated with grammar 2.1). This is solved with the addition of another level of inheritance on alternatives.

However, type-safe access to child nodes requires the node types to contain typed references to the children. This makes it impossible to use a single list with all children. Also, the methods to access the children need to be typed, thus for

every child at least a method to read the child is needed, and most likely also methods to modify the children.

Sometimes these methods need to perform additional bookkeeping, when the tree is modified. For example, a reference to the parent needs to be set, when a node is added. These methods need to be provided for all children for all nodes. Not only is it a lot of work, to write all these methods; they need to be updated when the structure of the AST is modified. This creates a serious maintenance problem. A good solution is to generate the typed references and modification methods directly from an AST specification.

2.2 Applying Algorithms to an AST

In the introduction to compiler construction, it was already mentioned that algorithms need to be applied to the AST. Examples mentioned are context checkers, optimisers and code generators. These algorithms can be added as part of the AST node classes, or written in separate classes. Several different techniques have been developed, for interoperability of algorithms and a tree, or adding functionality to a tree. The most important will be discussed in this section. Many of the examples given in this section are based on the examples used by Palsberg and Jay [1998].

2.2.1 Checking the Node Type

One of the most straightforward approaches to apply an algorithm to an AST, is to create a large `switch` statement (or an `else-if` chain), and decide what code to run by looking at the type of a supplied node. This is also called 'a type case'. This is illustrated in listing 2.1. This technique is most commonly used when the AST is homogeneous, as most other techniques require a heterogeneous AST.

Although this algorithm is fast and simple, it has some drawbacks. The most obvious is the frequent need for type checks and type casts, which is regarded bad practise in object-oriented languages. Another problem is that the compiler can not check that all cases are covered. If a new node type is introduced, and not added to the algorithm, a `RuntimeException` will occur when the algorithm is used. It would be preferable if the compiler could inform the developer of a missing case. Because this is not possible, the `switch` statements will become increasingly hard to maintain when the system grows.

An advantage is that the algorithm can be implemented completely separate from the AST nodes. No changes need to be made to the node classes to implement a new algorithm. This makes it possible to add functionality without recompiling the AST node classes.

2.2.1.1 Using Type Cases

To be able to apply an algorithm to an AST, in a flexible manner, without having to write the `switch` statements by hand, tree walkers are often used. A tree walker is similar to a parser, except that the tree walker parses a two-dimensional tree structure, instead of a linear token stream. Actions can be executed when a node is matched. The tree walker generator is responsible for generating the code to match the nodes. Because the cases in the `switch` statements are generated automatically, they do not have to be updated by hand, when the structure of the AST changes. This ensures that the code is always in sync with the grammar and that actions are executed at the right places.

Listing 2.1: Checking the node type

```
interface List {
  public static final int NIL = 0;
  public static final int CONS = 1;

  int getType();
}

class Nil implements List {
  public int getType() {
    return NIL;
  }
}

class Cons implements List {
  int head;
  List tail;

  public int getType() {
    return CONS;
  }
}

  public String printList(List l) {
    switch (l.getType()) {
      case NIL:
        return "Nil";
      case CONS:
        Cons consL = (Cons) l;
        return "(Cons " + consL.head + " " + printList(consL.tail) + ")";
      default:
        throw new RuntimeException("Illegal List: " + l.getType());
    }
  }
```

Different approaches for tree walkers are available. ANTLR provides a true tree parser. Tree parsers in ANTLR 3 [Parr, 2006] flatten the tree to a stream of tree tokens, with additional `UP` and `DOWN` tokens to indicate nesting, and walk over the tree using a recursive descent parser. Because the tree parser is based on linear parser rather than tree traversal, it has some drawbacks. The linear structure of the token stream makes it difficult to skip subtrees. Functionality such as visiting a subtree more than once, requires manual marking and rewinding of the token stream to allow the same tokens to be matched more than once.

A different approach is taken by Kimwitu [Van Eijk et al., 1997] and Memphis [Memphis]. Both extend the syntax of C with pattern matching constructs. These patterns can be used to examine the tree, and execute actions, depending on the structure of the tree. Using these patterns, recursive algorithms over the AST can be constructed. The tools assist the developer in writing the `switch` statements, and are able to inform the developer of malformed patterns, or missing cases.

2.2.2 Inheritance Pattern

The previous technique does not use type polymorphism provided by object-oriented languages. An approach, that better utilises object-orientation, is called

the inheritance pattern (which is a variation on the interpreter pattern [Gamma et al., 1995]). It uses an abstract method in the base class, that needs to be implemented in all subclasses. This implies that the inheritance patterns requires a heterogeneous AST. The same `print` algorithm as before is given in listing 2.2, but now implemented with the inheritance pattern.

Listing 2.2: Inheritance pattern

```
interface List {
  String print();
}

class Nil implements List {
  public String print() {
    return "Nil";
  }
}

class Cons implements List {
  int head;
  List tail;
  public String print() {
    return "(Cons " + head + " " + tail.print() + ")";
  }
}
```

This approach is clearly more elegant than that of listing 2.1. All casts and type checks are eliminated. Also, the compiler will give an error when the `print` method is not implemented in all node classes. Normally, the inheritance pattern is applied by adding an abstract method in the root node type and providing an implementation in all subclasses. This implies that all node types need an implementation and all methods should take the same arguments and have the same result type. When the inheritance pattern is used together with a custom defined inheritance tree, it is possible to specify operations only on certain types, or to use different method arguments and return types for different node types.

However, the inheritance pattern suffers from a problem: a single algorithm is spread across several files. When multiple algorithms are added to the AST node classes, a single algorithm becomes hard to find between all other, unrelated code. Furthermore, when a new algorithm needs to be added, all node classes need to be modified. This makes it impossible to add functionality without recompiling the node classes, for which the source code of these classes needs to be available.

2.2.3 Visitor Pattern

The visitor pattern [Gamma et al., 1995] is a commonly used design pattern, which can be used to extend existing code with additional functionality. It uses an `accept` method on the nodes, to simulate a double dispatch. The `accept` method forwards the call to the visitor, which contains the functionality. This is illustrated in listing 2.3. All functionality related to the `print` algorithm is now in a single class. Also, new functionality can be added to AST nodes, without the need to recompile the AST node classes.

However, several disadvantages of the visitor pattern are already visible in this example. The first is the need for `accept` methods in all classes. This means the AST has to be prepared for the visitor pattern. If these methods are not present, they need to be added to all classes, something which may not be possible, when

Listing 2.3: Visitor pattern

```
interface List {
  void <T> T accept(Visitor<T> v);
}

class Nil implements List {
  public <T> T accept(Visitor<T> v) {
    v.visit(this);
  }
}

class Cons implements List {
  int head;
  List tail;
  public <T> T accept(Visitor<T> v) {
    v.visit(this);
  }
}

interface Visitor<T> {
  T visit(Nil node);
  T visit(Cons node);
}

class PrintVisitor implements Visitor<String> {
  String visit(Nil node) {
    return "Nil";
  }

  String visit(Cons node) {
    return "(Cons " + node.head + " " + node.tail.accept(this) + ")";
  }
}
```

the source code is not available. Another problem is that all `visit` methods need to have the same return type and method arguments (the latter are omitted from the `print` example). With parameterised types, it is possible to use different return types and method arguments for different visitors, but within a single visitor, they should still all be the same.

Another major problem with the visitor pattern is its inflexibility with respect to changes of the structure of the tree. An important part of the pattern is the `Visitor` interface, which defines a `visit` method for each node type. When a new node type added, a new `visit` method needs to be added, which necessitates the addition of this method to all visitors (implementations of the `Visitor` interface). This is an invasive change, which affects many classes.

More problems are mentioned by Hachani and Bardou [2002]. They indicate, that there is no clear separation between definitions required by the pattern and other definitions. A visitor class does not indicate that it contains functionality for other classes. Also, the visitor pattern suffers from encapsulation breaching. For example, when a visitor needs access to attributes, these attributes need public accessor methods. This makes the attributes part of the public interface of the class, which may not be desirable.

Several attempts have been made to improve the visitor pattern. Palsberg and Jay [1998] describe the `WalkAbout` class, which can traverse object structures, with-

out relying on an `accept` method. This is further improved by Bravenboer and Visser [2001] by separating the visitor in two parts: (1) the visitor itself, containing the functionality of the algorithm and (2) a guide, describing the traversal order over the tree. A generic guide is also presented, which uses an algorithm similar to that of the `WalkAbout` class.

However, both suffer from a large performance penalty, because Java reflection is used to discover and visit the children. An implementation with dedicated methods (the inheritance pattern) or the visitor pattern outperforms the original `WalkAbout` class by several orders of magnitude (although the visitor pattern implementation is somewhat slower than the inheritance pattern). Bravenboer and Visser [2001] manage to improve the efficiency of the `WalkAbout` algorithm significantly, using a caching mechanism, but the difference between the `WalkAbout` and a normal visitor still is 2 orders of magnitude.

A different approach is taken by Visser [2001]. This approach is further discussed by Van Deursen and Visser [2004]. They describe JJTraveler, a system of small, almost trivial visitors, which can be combined to build complex behaviour. Many of these visitors can be written in a generic form and are included as a standard library. Examples of these combinators are `Identity`, `Sequence` and `All`. The first does nothing. The `Sequence` combinator takes two visitors and performs one after the other. `All` applies a visitor to every immediate subtree sequentially.

The AST class generator of JJForester [Kuipers and Visser, 2003] is extended to generate the syntax dependent visitor classes, including the `Fwd` combinator, which is used to build the bridge between generic and syntax dependent combinators. Using these combinators, they are able to construct complex syntax analysis tools with relatively little code. What is even more important; many of these algorithms can be (partially) written in a generic form, which makes it possible to reuse them in applications with a different AST.

However, these visitor combinators are still based on the standard visitor pattern. Therefore, they still suffer from the same drawbacks: `accept` methods are needed in all classes and all methods need identical signatures. To be able to write generic visitor combinators, which can be reused on different ASTs, a variation of the staggered visitor pattern [Vlissides, 1999] is used. This pattern separates the visitor pattern in a generic framework and an application specific part. It requires all nodes to extend from a generic framework node class. This generic node class defines two methods: one that returns the number of children, and one that returns a child, using its index. Implementations of these methods need to be provided in the node classes. When a framework with JJTraveler support is used, such as JJForester, all required methods are automatically generated. When the node classes are written by hand, or generated by a framework which does not support JJTraveler, all node classes need to be modified. Sometimes, it may not even be possible to apply these changes (for example, to change the superclass of all node classes), limiting the applicability of the visitor combinator framework.

2.2.4 Multiple Dispatch

The traditional visitor pattern requires an `accept` method on all nodes to perform a double dispatch. There are, however, also languages with native support for multiple dispatch. In these languages, the actual method, to which the execution is dispatched, is chosen not only on the first argument, but on several. This allows a visitor to dispatch execution to the correct visit method directly, without the need for an `accept` method in the node class.

MultiJava [Clifton et al., 2000] extends Java with open classes and multiple dispatch. Open classes allow the addition of functionality to a class, without

subclassing it. A valid Java program still is a valid MultiJava program, with the same semantics. MultiJava programs are compiled to Java bytecode, which runs on a standard Java virtual machine.

The Nice programming language (formally Bossa) [Bonniot, 2000] is more than an extension to Java. The syntax is still similar to Java, but has many enhancements. One of these enhancements is the multimethod. A multimethod can be used to add functionality to an existing class, without subclassing it (similar to open classes). Multimethods also support multiple dispatch.

2.2.5 Aspect-Orientation

Functionality that needs to be added to a tree can be seen as behaviour which crosscuts the AST classes. Aspect-orientation can be used to specify this crosscutting behaviour. The impact of aspect-orientation on compiler development is discussed by Wu et al. [2006]. In this paper, it becomes clear that aspect-orientation solves many of the traditional problems of compiler development. Aspect-orientation is able to provide the visitor pattern in a much cleaner way.

However, as indicated by Wu et al. [2005], it is not able to lift all limitations of the object-oriented visitor pattern. The visitor pattern is still difficult to use when the structure of the AST changes frequently. To overcome this difficulty, an approach, the compiler matrix, is introduced, with which it becomes possible to change an implementation between the inheritance pattern and the visitor pattern. This greatly reduces maintenance problems when both operations and AST structure are subject to frequent changes. This does, however, add another tool to the build process. The AST classes are generated with TLG [Bryant and Lee, 2002], functionality is added using the compiler matrix and, finally, the code is compiled using AspectJ. Little or no documentation is available for the first two applications and support for all three is limited.

Treecc, by Weatherley [2002], provides a limited aspect-oriented language specifically tailored to compiler development. It checks for complete coverage of defined operations and allows easy transition between the inheritance pattern and the visitor pattern. With Treecc, it is possible to specify the structure of the AST and the functionality in different modules. However, attributes need to be declared as part of the structure, and can not be declared as part of the module with the functionality, which requires the attribute. Also, the syntax of Treecc is somewhat cryptic and difficult to understand for someone with little or no experience with Yacc and C.

2.2.6 Attribute Grammars

Attribute grammars are introduced by Knuth [1968]. A system is described in which attributes and semantic rules can be attached to a grammar definition. A distinction is made between inherited attributes, which are evaluated from the top down, and synthesised attributes, which are evaluated from the bottom up. The semantic rules are used to specify the relation between different attributes. Attribute grammars are often used as formal specification of the semantics of a grammar. This means that the specification can also be used as the implementation.

An example of the `print` algorithm, as attribute grammar, is given in grammar fragment 2.2. This grammar only uses synthesised attributes. The resulting string is constructed from the bottom up in a single pass over the tree. A more complex example is given in grammar fragment D.1 on page 94, which combines inherited and synthesised attributes, and requires multiple passes over the tree.

Grammar Fragment 2.2 Attribute grammar

⟨*list*⟩	::=	'Nil'
	|	'Cons' ⟨*number*⟩ ⟨*list*⟩
⟨*number*⟩	::=	A sequence of digits.

list : Nil.	[*v* **of** *list* = 'Nil';]
$list_1$: Cons *number* $list_2$.	[*v* **of** $list_1$ = '(Cons ' ++ *v* **of** *number* ++ ' ' ++ *v* **of** $list_2$ ++ ')';]
number : number_token.	[*v* **of** *number* = *value* **of** number_token;]

SLADE [SLADE, 2002] provides a very limited attribute grammar evaluator, which requires the attributes to be evaluated in a single pass. SLADE has been used at the University of Twente, for the course compiler construction, and can only be used to write compilers.

The Attribute Grammar System, initially developed by Swierstra et al. [1998], is a general purpose attribute evaluator. An attribute grammar, in which semantic functions are described through Haskell expressions, is compiled into a Haskell program. The Attribute Grammar System fully supports attribute grammars with inherited and synthesised attributes and grammars which require multiple passes. This is achieved through the use of lazy evaluation in Haskell, which automatically resolves the order in which the attributes need to be evaluated.

2.3 Available Parser Generators and Tree Processors

A great variety of parser generators and tree processors is available. Some parser generators can also generate tree processors, other tree processor generators are provided as separate applications. This section discusses a small selection of parser and tree processor generators. The chapter is slightly biased towards Java systems. This is because Java is a popular programming language in the academic world and because TPL is implemented in Java. This section will not discuss the advantages and disadvantages of the tools, because most are already covered in the other parts of this chapter.

2.3.1 Lex and Yacc

A very popular lexer and parser generator combination is Lex and Yacc. Many Lex and Yacc compliant alternatives are also available, such as Flex and Bison. Lex and Yacc generate C/C++ code for lexing and parsing respectively. Actions can be embedded in the grammar, which can be used to construct an AST. However, Yacc does not provide support for ASTs. These can either be constructed using hand written code, or with a separate tree processor tool, such as Memphis [Memphis] and Kimwitu [Van Eijk et al., 1997]. Both extend the C syntax with constructs that can be used to specify the structure of the AST. To perform operations on this tree, both languages provide pattern matching constructs, which can be used to create `switch`-like statements.

2.3.2 ANTLR

ANTLR [Parr and Quong, 1995] is a popular parser generator written in Java. From the website [ANTLR, 2006]:

> ANTLR, ANother Tool for Language Recognition, (formerly PCCTS) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions. ANTLR provides excellent support for tree construction, tree walking, and translation.

ASTs can be directly constructed from the parser. ANTLR uses a homogeneous AST by default, but it is possible to specify different classes for different AST nodes, creating a heterogeneous AST. The tree walkers used by ANTLR are based on a recursive descent parser. The tree is flattened to a stream of tree nodes. `UP` and `DOWN` tokens are inserted, to indicate nesting of nodes. Actions can be inserted in the tree parser, to perform operations on the tree.

2.3.3 JavaCC

JavaCC is another very popular parser generator written in Java. From the website [JavaCC, 2006]:

> Java Compiler Compiler™ (JavaCC™) is the most popular parser generator for use with Java™ applications. [...] In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc.

Unlike ANTLR, JavaCC does not support target languages other than Java. Also, no tree walker is included. By default, JavaCC does not construct ASTs. However, different tree building preprocessors exist. These preprocessors generate Java actions in the grammar to construct the trees.

2.3.3.1 JJTree

JJTree is a tree building preprocessor that comes with the default JavaCC distribution [JJTree, 2006].

> JJTree is a preprocessor for JavaCC [tm] that inserts parse tree building actions at various places in the JavaCC source. The output of JJTree is run through JavaCC to create the parser.

JJTree will generate a homogeneous AST by default, but it is possible to generate heterogeneous ASTs. JJTree does not generate any visitors, but can generate the required visit methods in the AST nodes.

2.3.3.2 JTB

JTB [Tao and Palsberg, 2005] also is a tree builder preprocessor, but is less known. JTB always generates a heterogeneous AST that is partially, strictly typed. Only single, required nodes are strictly typed. This is caused by the fact that JTB generates an AST with a single inheritance level.

In addition to the AST node classes JTB also generates several visitors (with and without arguments and return values), which visit the tree in depth-first order. Methods in these visitors can be overridden to provide custom functionality.

2.3.4 SLADE

SLADE is a system that has been used for the course compiler construction at the University of Twente. From the website [SLADE, 2002]:

> The SLADE system is a development environment for programming languages. The main restriction on the nature of the programming language that is developed is the Application Programming Interface, abbreviated to API. A general-purpose API is included in the system, but the user is free to add and change the routines in the API to suit his or her needs. The SLADE system is written in, and generates, C code. A compiler that is generated by SLADE generates code for a VIrtual Machine called VIM.

SLADE does not generate an AST. All actions are executed directly from the parser. SLADE only supports single-pass-compilers. An interesting feature of SLADE is its attribute evaluator engine. Attributes can be passed up and down the parse tree.

2.3.5 SableCC

SableCC [Gagnon and Hendren, 1998] is a parser generator written in Java that generates parsers in Java.

> SableCC is an object-oriented framework that generates compilers (and interpreters) in the Java programming language. This framework is based on two fundamental design decisions. Firstly, the framework uses object-oriented techniques to automatically build a strictly typed abstract syntax tree. Secondly, the framework generates tree walker classes using an extended version of the visitor design pattern which enables the implementation of actions on the nodes of the abstract syntax tree using inheritance. These two design decisions lead to a tool that supports a shorter development cycle for constructing compilers.

The most interesting feature of SableCC is its strictly typed AST. All operations on the AST are type-safe and all sub-nodes are stored as typed fields. SableCC uses inheritance on alternatives for the AST node classes. All token classes are subclasses of the `Token` class. The names of classes for production rules are prefixed with a '`P`', and an '`A`' is used for alternatives.

2.3.6 JFlex, CUP and Classgen

JFlex [Klein, 2005], CUP [Flexeder et al., 2006] and classgen [Klein and Brandl, 2003] are a scanner generator, parser generator and a class generator respectively. These three tools are designed to work together, although they can also be used separately. All three generate Java code.

JFlex and CUP together form a parser generation framework, but do not have support for AST construction. Classgen can be used to generate the AST classes. On the website, classgen is described as:

> classgen is a 100% Java, platform-independent class generator. Though it was designed with compiler design in mind it's very compact and flexible specification language allows to generate classes for all purposes. For the compiler designer it offers some more extremely useful features. It has built in support for the visitor design pattern and gets along perfectly with the scanner generator JFlex and the parser generator CUP.

Attributes and methods can be added to the AST node classes. Classgen can also prepare the AST node classes for the use of the visitor pattern. Classgen uses inheritance on alternatives for the generated AST node classes.

2.3.7 Treecc

Treecc [Weatherley, 2002] is a tree generator, with support for several target languages, including Java and C#.

> The treecc program is designed to assist in the development of compilers and other language-based tools. It manages the generation of code to handle abstract syntax trees and operations upon the trees. [...] Treecc was originally written to assist with the development of the C# compiler for the Portable.NET project. However, it is much more general than that and can be used in any system that makes heavy use of abstract syntax trees and the operations upon them.

Treecc does not provide a parser generator, but can be used in conjunction with many parser generators. As with Classgen, Treecc allows the declaration of node attributes and functionality, but Treecc offers more robust support for 'operations'. It is able to check if an operation is defined for all nodes, even when multiple arguments are used. Both the inheritance and visitor pattern are supported. The developer is able to fully define the inheritance tree of the AST node classes.

Chapter 3

The Tree Processing Language

The Tree Processing Language (or TPL) is a domain specific language for specifying trees and behaviour on trees. Applications written in TPL can be compiled to tree structures with the tool TPLc. TPL's typical application is in the field of compiler construction, but it can be used in any system which uses tree structures. TPL allows the developer to specify a strictly typed tree, annotate the tree with attributes and add functionality to this tree.

TPL tries to minimise the effort required by the developer to develop a tree structure, by generating as much of the code as possible. The organisation used in this generated code focuses on static type checking, performed by the compiler, and code reuse. TPL does not focus on how the tree is constructed at runtime. Although, it is possible to let an AST, written in TPL, be constructed by a parser. Support for ANTLR 3 is provided.

This chapter will discuss the added value of TPL. In §3.1, the techniques discussed in the previous chapter are evaluated, and careful choices are made, on which TPL is based. Next, an overview of TPL's architecture is given in §3.2. Followed by an explanation of the structure of the tree in §3.3 and the addition of functionality to the tree in §3.4. Finally, §3.5 gives a detailed view of the inheritance pattern, as it is used by TPLc.

3.1 Rationale

Language processing, and in particular compiler construction, is a complex field of engineering. The previous chapter mentions many techniques which are available to the developer. However, many of these techniques suffer from maintenance problems or lack the flexibility required in a large language processing system. In this section, a combination of techniques is discussed, which support the construction of a maintainable and extensible language processor.

Conventional compiler construction is often based on the syntax definition of the source language. The suggested approach is to use the abstract syntax tree as base for the compiler. The compiler becomes an operation on the abstract syntax tree. This approach can also be taken in other systems that operates on a tree structure. The tree does not need to be an abstract syntax tree, but can be any tree-like data structure. For example, it can be a component tree in a user interface.

Working with a large tree structure, with many different kinds of nodes, is a task with much repetition. Node creation, inspection and manipulation often requires code for different node classes, which is similar in nature, but has subtle differences. Also, these pieces of code often have a strong reliance on the runtime structure of the tree and need to be updated whenever this structure changes. Therefore, as much of the tree (types and code) as possible should be automatically generated from a specification. The tool that generates the tree code should not fix the choice for an implementation language for the tree; however to limit

the complexity of the tool, it is demanded that an object-oriented language is used.

As already mentioned in chapter 2, a tree has two main distinctive features: its structure and its behaviour (or functionality). This separation will also be present in the discussion in this section. First, the structure of the tree will be discussed in §3.1.1, followed by a discussion of the behaviour of the tree in §3.1.2.

3.1.1 Structure of the Tree

§2.1 discusses the two categories of AST organisation: homogeneous, where every node is of the same type, and heterogeneous, where nodes can have different types. Homogeneous trees are especially useful in smaller systems, or when the tree is developed by hand. Heterogeneous trees, on the other hand, are cumbersome to create by hand, due to the large number of different types, but allow stricter type checking and more advanced techniques of adding behaviour to the tree. Automatically generating the tree types and management code eliminates the primary disadvantage of a heterogeneous tree. Therefore, the choice is made to use a heterogeneous tree.

From the three commonly used organisations for a heterogeneous tree (see §2.1.2), custom defined inheritance provides the most flexibility with respect to code reuse (especially when the inheritance pattern is used) and is able to support a tree with strictly typed children. However, to specify a tree with custom defined inheritance, it is needed to specify the node types, their super types, the children, and the types of the children, whereas with the other two approaches the entire structure of the tree can be determined from a grammar specification (or something similar). The added benefits are considered to outweigh the additional work that has to be performed when specifying the tree.

3.1.2 Behaviour of the Tree

In §2.2 six different approaches are discussed with which the behaviour of a tree can be specified. The first three, checking the node type (§2.2.1), the inheritance pattern (§2.2.2) and the visitor pattern (§2.2.3) can be used in most object-oriented languages. Multiple dispatch (§2.2.4) and aspect-orientation (§2.2.5) require features, which are not present in most object-oriented languages. Finally, attribute grammars (§2.2.6) are usually evaluated with an interpreter, or a preprocessor is used to produce compilable code from the grammar, which evaluates the attribute grammar.

The decision to require a typical object-oriented language as target language eliminates the multiple dispatch and aspect-oriented approaches. The other approaches all have their own field of applicability. A switch statement is very useful for short algorithms, which can be implemented inline, in a single method. The visitor pattern is very useful for adding behaviour to a tree, when the source code is not available (or cannot be modified). This is not possible with the inheritance pattern, but the latter does not suffer from encapsulation breaching and allows different method arguments and return types to be used for different parts of the tree. This makes the inheritance pattern a good candidate to use for specifying behaviour when the source code of the tree nodes is available. Finally, attribute grammars allow a formal specification of the behaviour of a tree. However, they need to be compiled into a representation that can be evaluated by the target language.

Due to the different applications of the different techniques, it would be preferable if all where supported by the tree generator. Support for the visitor pattern requires `accept` methods in all node classes and a visitor interface, with `visit`

methods for all node classes. These are straightforward to generate and are therefore included. The other three approaches are more difficult to integrate and are discussed below.

3.1.2.1 Attribute Grammars

A typical object-oriented language does not provide the features required for support for attribute grammars. Attribute grammars require a dependency analysis to determine the order in which the attributes need to be evaluated. An exception is the class of L-attributed grammars (see [Alblas and Nymeyer, 1996, Section 9.2]), which can be evaluated in a single top-down pass over the tree.

Support for L-attributed grammars is straightforward to provide, because they have a great resemblance with method invocations: the inherited attributes are provided through method arguments and the synthesised attributes are returned by the method invocation. To be able to return multiple synthesised attributes, the methods should be able to return multiple values. Although this is not supported by most object-oriented languages, it can easily be emulated by wrapping the values in a tuple object.

The tree generator should be able to evaluate L-attributed grammars; thus provide multiple return values. Other classes of attribute grammars require a dependency analysis, which is difficult to provide in object oriented languages. A preprocessor, which converts an attribute grammar into a representation which can be read by the tree generator, can solve this. However, this preprocessor will not be part of the tree generator tool.

3.1.2.2 Query/Test Language

A query/test language can be used to select parts of the tree and to test whether the tree conforms to a certain structure. This can also be used to implement node type checks, as discussed in §2.2.1. XPath [Clark and DeRose, 1999] is an example of such a language. Naturally, these queries and tests can also be written by hand, but the advantage of an advanced query/test language is that the representation usually is more compact, easier to understand and can be checked by the compiler.

The tree generator tool should incorporate a query/test language with at least the following features: node selection, iteration over nodes, and type cases with pattern matching.

Node Selection It should be easy to locate nodes deeper down the tree, but also higher up the tree. Queries that are often used are the selection of a child, a direct ancestor or the node itself. These queries should therefore be easy to write and clear. More complicated queries, which include node type checking, or other conditions, are less common, but can be useful. The tool should support a query language that is powerful enough to express these more complicated selections.

It is often possible to determine the exact type of the nodes returned by a query, for example when a typed child is selected. Typed queries allow better type checking when the generated classes are compiled by the target language's compiler. Therefore, the tool should support typed queries where possible.

Iteration over Nodes A query can often return multiple nodes. Iterating over these nodes is a common operation. For example, the context checker of a method declaration probably needs to iterate over the specified method arguments. Because this operation is used frequently, the tool should provide support for iteration over nodes. This can be combined with the node selection, where the iterator iterates over the results of a query.

Type Cases Type cases, or node type checks, often are a convenient way of specifying algorithms on a tree. The biggest disadvantage is the effort that it takes to keep these checks in sync with the actual structure of the tree during development. Therefore, the language should support automatically generated (and checked) type cases, similar to those seen in Memphis [Memphis] and Kimwitu [Van Eijk et al., 1997]. This includes pattern matching constructs, for more complicated checks.

3.1.2.3 Inheritance Pattern

As stated before, the inheritance pattern provides a flexible way of expressing the behaviour of a tree. In the standard inheritance pattern, an abstract method is added to the base class and implemented in the concrete classes, extending from this base class. When a custom defined inheritance tree is used, this can easily be extended to a form where the abstract method is added to a subclass of the base class and only applies to the subclasses of this class.

However, with this approach, the inheritance pattern still follows the inheritance tree of the node classes. This inheritance tree is designed to make it possible to build a strictly typed tree and may not be suitable for all operations that need to be defined over the tree. For example, consider a user interface component tree with visible and invisible[1] components. Only visible components require a `paintComponent` method and the invisible components do not. This could be realised by creating a `VisibleComponent` node class and letting all visible components extend from this class. However, this moves implementation details of the behaviour of the tree to the structure of the tree. Not only does this violate the separation of concerns, it may also be impossible to realise, when different operations need to be applied to disjoint subtrees of the node classes.

To solve these problems, the inheritance tree of a single behaviour should be definable independently of the inheritance tree of the node classes or any other behaviour. Not only does this allow the definition of operations on disjoint subtrees of node classes, but it also makes it easier to specify common functionality. This is explained in §3.5.

3.2 Architectural Overview

The flow of information for an application developed with TPLc is displayed in figure 3.1. A tree definition and several logic specifications are taken by TPLc to produce the implementation of the AST. All source files (including additional support files) are compiled by the compiler to produce binaries (or class files for Java).

The tree definition describes the structure of the AST, and is discussed in §3.3. The logic specifications are used to add functionality to a tree, which is discussed in §3.4.

The figure also includes the generation of a parser with ANTLR, which is optional. ANTLR is used to generate a parser from the parser specification. In addition, ANTLR also generates a token file, which contains the mapping from token names to internal numbers. With TPL it is possible to give a mapping from the token names in the token file to the AST node classes. This can be used in the parser to construct the AST instance using ANTLR's native tree construction mechanism.

[1] Invisible indicates that the component does not have a graphical representation at all, not that it is currently hidden.

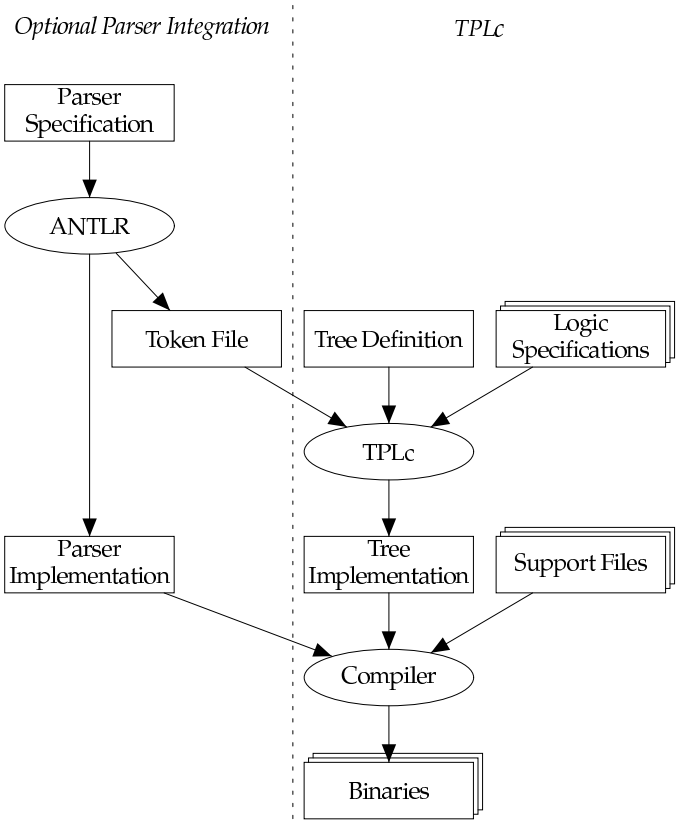

Figure 3.1: Architectural Overview

Listing 3.1: Example tree definition

```
package expression;

abstract Expression;
abstract Binary(Expression left, Expression right) extends Expression;

Literal(Integer value) extends Expression;
Multiply extends Binary;
Add extends Binary;
```

The resulting AST, generated by TPLc, can be used throughout the application. It is possible to let ANTLR construct the AST, but the AST can also be constructed in different ways. Functionality added to the tree, is available to the application using the tree. A typical compiler would use several logic specifications for functionality, such as a context checker and a code generator. First, the file is parsed, and an AST constructed. Next, the context checker is invoked. Finally, the code generator is called.

3.3 Tree Structure

TPL uses a separate file, a *tree definition*, to describe the structure of the AST. An example of such a file is given in listing 3.1, which defines an AST for expressions.

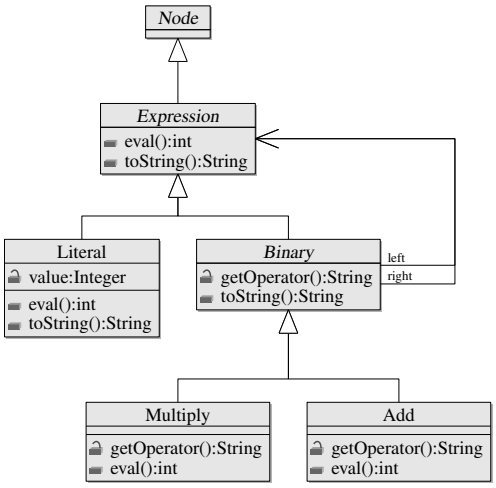

Figure 3.2: Functionality added to the AST nodes

The resulting class hierarchy is displayed in figure 2.3 on page 11. The AST has a heterogeneous structure, with custom defined inheritance, as described in §2.1.2.3. Not only the inheritance tree, but also the names and types of children are specified in this file. Two kinds of children are supported by TPL: nodes and values. The first is used to form the runtime structure of the AST. The second can be any data type, as long as it can be constructed from a string. When the AST is constructed by a parser, the parser will construct the value from the token. Typical applications are enumerations, such as operator types, and literals, such as strings and integers.

All child nodes in the AST are strictly typed. The developer needs to specify the exact types of these children. Methods to access the children are automatically generated by TPLc. Constructors, to build ASTs manually, are also added.

3.4 Adding Functionality

The tree structure, described in §3.3, only forms the basis of the entire AST structure. Functionality is added to this tree using *logic specifications*. Multiple logic specifications can be used, which makes it possible to separate different algorithms. Every logic specification can specify its own inheritance tree, onto which the inheritance pattern is applied. Attributes can also be specified in a logic specification and are added with the functionality.

Two exemplary logic specifications, that specify functionality for the tree definition in listing 3.1, are given in listing 3.2 and 3.3. The first implements an interpreter and the second a pretty printer. The resulting class hierarchy is displayed in figure 3.2[2].

In the interpreter specification, specifications for the node classes `Expression`, `Literal`, `Multiply` and `Add` can be seen. The specification adds an abstract `eval` method to the `Expression` node class. This method is implemented in the other nodes, to return the correct results. For `Literal`, the result is the value. Both `Multiply` and `Add` recursively invoke the `eval` method on the left and right children and use these values to evaluate the result. In this example, it can be seen

[2]This class diagram is a simplification of the class structure actually generated. A description of the actual structure can be found in §3.5.4

Listing 3.2: A simple interpreter

```
/** An interpreter for the expression language. */
package expression.interpreter;

abstract Expression {
  interface public int eval();
}

Literal extends Expression {
  interface public int eval() {
    return #value;
  }
}

Multiply extends Expression {
  interface public int eval() {
    return #left.eval() * #right.eval();
  }
}

Add extends Expression {
  interface public int eval() {
    return #left.eval() + #right.eval();
  }
}
```

that the inheritance tree of the interpreter differs from that of the tree definition: both `Multiply` and `Add` extend from `Expression`, instead of `Binary`. The `Binary` node is not needed at all.

For the pretty printer, a superclass is used, which does not have a corresponding node class; `Printable`. Another interesting feature is the use of a `protected` method in `Binary`, which returns a string representation of the operator. This allows the common behaviour of the pretty printer for binary expressions to be shared by all binary expressions. The difference between an `interface` and an `implementation` method is that an `implementation` method is not exposed through the public interface of a node class. This is explained further in §3.5.4.

A disadvantage of the inheritance pattern is that it is not possible to add functionality without recompiling the AST node classes. Therefore, TPLc also adds `accept` methods to the AST node classes, for use with the visitor pattern. A `Visitor` interface is also generated. TPLc does not provide any default implementations of the visitor pattern, such as a depth-first visitor. With TPL, the visitor pattern should only be used to add behaviour to the tree, when the source code is not available. During normal development, the logic specifications should be used to add behaviour to the tree.

3.4.1 Attribute Grammars

As mentioned before, support for L-attributed grammars is desirable. TPL provides this support through visiting methods, which can return multiple values. When every AST node has exactly one visiting method and these nodes are visited in depth first order, then the evaluation of these methods corresponds to the evaluation of a L-attributed grammar. However, TPL allows the addition of multiple visiting methods to a single AST node. This can be used to evaluate more

Listing 3.3: A pretty printer

```
/** A pretty printer for the expression language. */
package expression.pretty;

abstract Printable {
  interface public String toString();
}

abstract Expression extends Printable;

Literal extends Expression {
  interface public String toString() {
    return #value.toString();
  }
}

abstract Binary extends Expression {
  interface public String toString() {
    return #left.toString() + " " + getOperator() + " " + #right.toString();
  }

  implementation abstract protected String getOperator();
}

Multiply extends Binary {
  implementation protected String getOperator() {
    return "*";
  }
}

Add extends Binary {
  implementation protected String getOperator() {
    return "+";
  }
}
```

complicated attribute grammars, although the developer has to determine the dependencies between the attributes by hand and coordinate the evaluation. For a deeper discussion of attribute grammars, refer to appendix D.

3.4.2 Test/Query Language

A query language is included in TPL. This language can be used to select nodes and to iterate over nodes. Some simple examples can already be observed in listings 3.2 and 3.3. `#left`, `#right` and `#value` are used to select the left and right children, or the value of a node. With a `#foreach` statement, it is possible to iterate over a node set that results from a query.

Some more complex examples are shown in listing 3.4. The first returns the parent of the parent of the current node. The second query matches the parent of every node under the current node; in other words: all non-leaf nodes under the current node. Where `/` indicates a single step, `//` indicates an arbitrary number of steps. With the third query, all nodes of type `Identifier` anywhere under the children `statements` are matched. The `[Type1, Type2, ...]` notation is used

Listing 3.4: Query examples

```
#parent/parent
#this//parent
#statements//*[Identifier]
#*//*[AssignExpression]/var
```

to filter nodes of certain types. The asterisk character, `*`, serves as a wildcard to match all children of a node. The final query combines most features to return the left-hand-side of every assignment expression.

TPLc automatically determines the most specific type of the results of a search, to which the nodes will conform. For example, for the last query in listing 3.4, TPLc determines that the nodes will all be of type `Identifier`, as the `var` child of `AssignExpression` is of type `Identifier`. This makes it possible for the compiler of the target language to perform static type checking. TPLc includes a checker, which generates an error when a query always returns the empty set.

Another feature of queries, supported by TPL, is the compilation and execution of queries at runtime. This allows an application (or the user of an application) to construct and execute queries when the application is running. Naturally, static type checking is not possible on runtime queries.

TPL does not support type cases (yet), as seen in in Memphis [Memphis] and Kimwitu [Van Eijk et al., 1997]. Support for type cases is discussed in §6.3.2.

3.5 Advanced Usage of the Inheritance Pattern

TPL uses the inheritance pattern to add functionality to the node classes. However, due to the ability to allow different inheritance trees for different logic specifications, it is not trivial to apply the inheritance pattern. This section discusses the structure generated by TPLc to support different inheritance trees for a single tree structure.

First, an example is introduced, which illustrates why the standard inheritance pattern is not strong enough to solve some of the problems faced, when implementing behaviour of a tree. This is followed by a possible solution, which uses multiple inheritance, in §3.5.2. Next, in §3.5.3, a solution is presented, which allows the generation of different inheritance trees in a language that does not support multiple inheritance. Finally, in §3.5.4, the example presented in §3.4 is revisited and the actually generated code is discussed.

3.5.1 Example

In many functional programming languages a technique, called pattern matching, is used. Several examples of pattern matching are given in listing 3.5. The factorial example uses `0` and `n`. This results in the first function to be selected when `fac` is called with `0` as parameter and the second to be selected in all other cases. In those other cases, the value of the parameter is bound to `n`, which can be used in the right-hand-side of the function. The other functions use nested patterns. The outer pattern, `x:xs`, splits a list into its head and its tail. An identifier (as in the factorial function) and a wildcard are nested inside this pattern. The identifier and the wildcard both match all values, but the wildcard does not bind the value to an identifier.

Listing 3.5: Pattern matching examples

```
fac 0           = 1
fac n           = n * fac (n-1)

listhead (x:_)   = x
listtail (_:xs)  = xs
```

The tree node for a function (in this simplified example) should contain a name, a pattern and an expression. The node classes for the different patterns will be a subclass of the `Pattern` node class and a similar structure can be used for the expression node classes. However, pattern can often be used as expressions. In this example, only the wildcard pattern is not a valid expression. It would be desirable if code could be shared between patterns and expressions. It is not possible to make the `Expression` node class a subclass of `Pattern`, because not all expressions are patterns (for example, multiplication is not a valid pattern). This results in two separate inheritance subtrees, one for patterns and one for expressions. The inheritance pattern only allows code to be shared through the common base class(es) of these trees. However, code added to the first common superclass of `Pattern` and `Expression` will be shared with all expressions and patterns. This makes it difficult to implement behaviour, such as a pretty printer, for a single pair of a pattern and an expression. Either the behaviour needs to be implemented in both classes or is available to all patterns and expressions. An implementation which implements the common behaviour in the base class is illustrated in figure 3.3.

3.5.2 Inheritance Pattern with Multiple Inheritance

To solve the problem discussed above, multiple inheritance can be used. A class diagram of the same tree, as displayed in figure 3.3, is shown in figure 3.4, but now multiple inheritance is used to share the functionality. The class diagram shows that both `LiteralPattern` and `LiteralExpression` inherit from `PrintLiteral` thereby inheriting the literal printing behaviour. The behaviour can be shared without moving it up in the inheritance 'tree'.

The inheritance pattern with multiple inheritance greatly increases the reusability of functionality, with the disadvantage of a more complicated design. Many classes have to be maintained and, in large systems, it can become difficult to get a good overview of the relations between these classes. However, with TPL, development is based in the tree definition and the logic specifications. The final class structure is generated and the burden of maintaining these class is taken by TPLc.

3.5.3 Inheritance Pattern with Composition

Not all languages provide multiple inheritance. For example, the very popular languages Java and C# do not. For these languages, a different solution is needed. As with the multiple inheritance approach, the behaviour is implemented in separate classes. However, these classes are not linked to the tree node classes through inheritance, but through composition; the node classes hold an instance of the classes implementing the behaviour. This is illustrated in figure 3.5. `AbstractCommonNode` provides an abstract `print` method, through which the pretty print behaviour can be accessed. This method is implemented in the

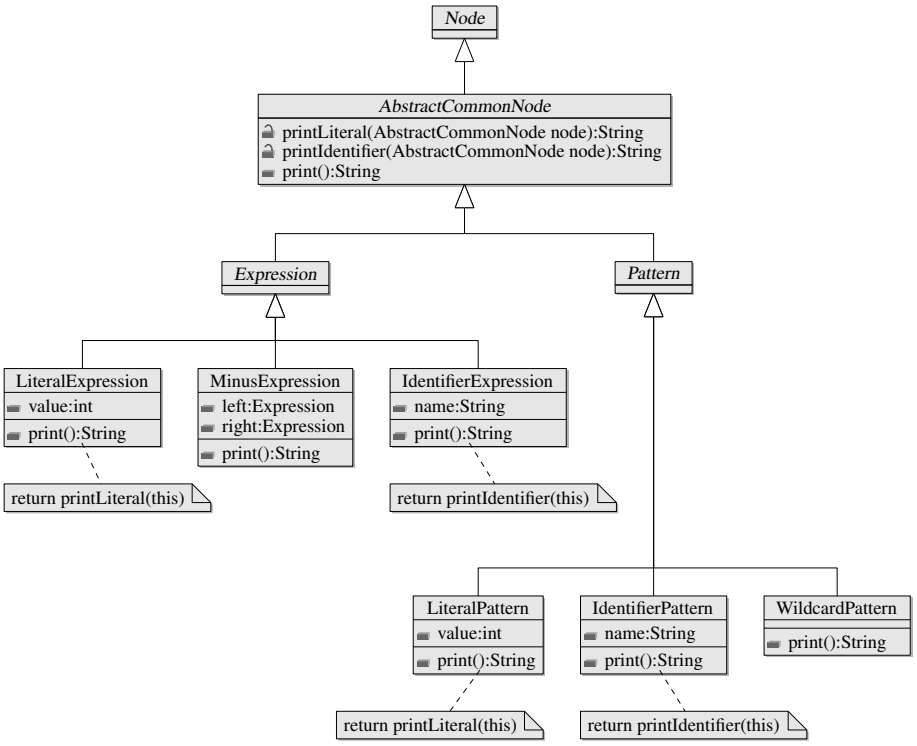

Figure 3.3: Behaviour with the inheritance pattern

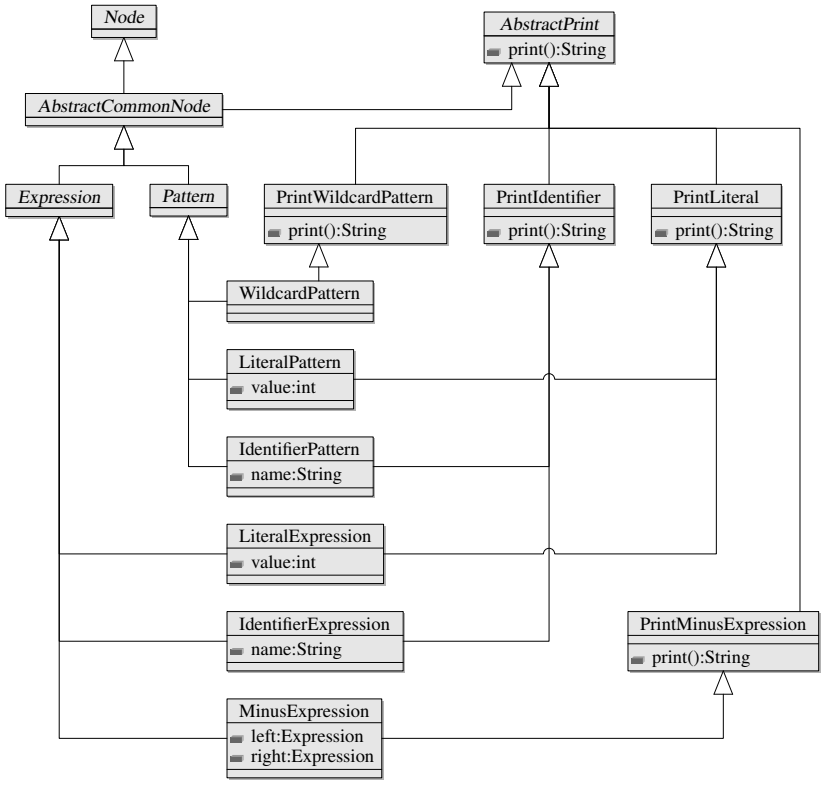

Figure 3.4: Inheritance pattern with multiple inheritance

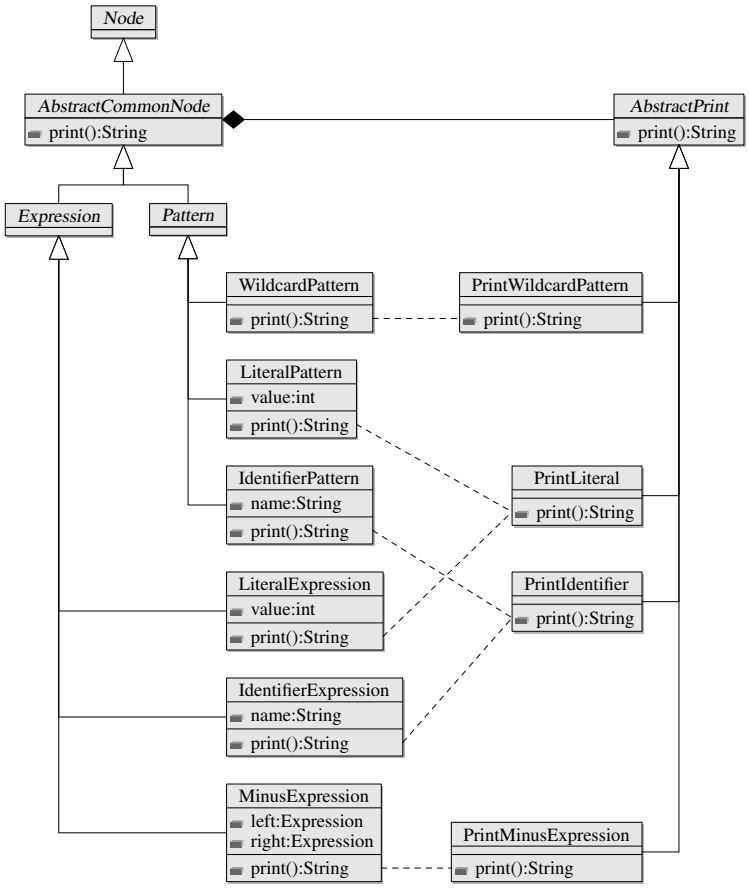

Figure 3.5: Inheritance pattern with composition

subclasses of `AbstractCommonNode`. `MinusExpression` and `WildcardPattern` provide their own implementation of the `print` method, in the classes `PrintMinusExpression` and `PrintWildcardExpression` respectively. `LiteralExpression` and `LiteralPattern` share the same implementation of the `print` method, through `PrintLiteral`. The node classes `IdentifierExpression` and `IdentifierPattern`, share the behaviour in `PrintIdentifier`. The inheritance pattern itself is applied to the composition nodes, containing the behaviour.

Although it is possible to share behaviour across nodes in different parts of the inheritance tree, using the inheritance pattern with composition, it is not possible to access this behaviour through a common interface. This is not a problem with the pretty print functionality, because all nodes are printable, however other behaviour may only be applicable to subtrees of the tree. For example, consider a user interface framework, with different components. Some of these components might have the ability to fire user events, for which listeners can be registered, through a method called `addEventListener`. Even though these components all have this method, it is not possible to access this behaviour without relying on the exact type of the component, because the method is not part of a common type of all the components. This can be solved by moving the method upward in the inheritance tree (to the root), but this results in all components sharing the method, even when a component cannot fire user events. A more elegant solution is to specify the method in an interface and to have the various components implement this interface. This is illustrated in figure 3.6, which also shows

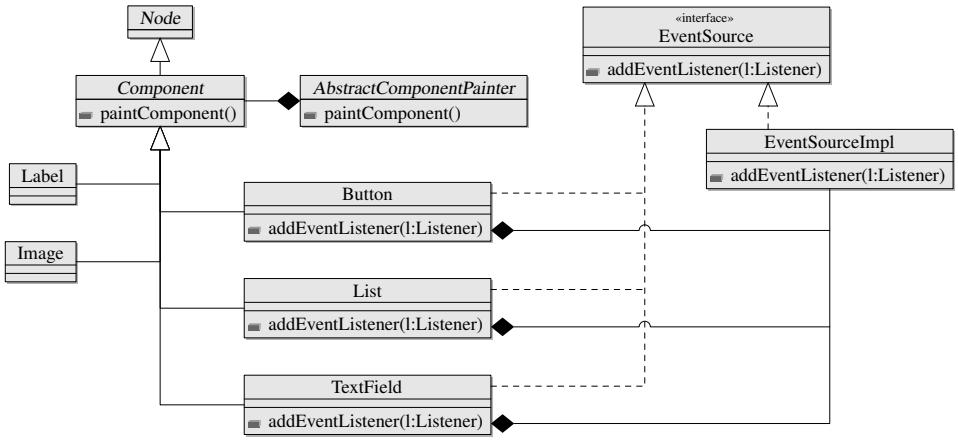

Figure 3.6: Inheritance pattern with composition and interfaces

a second composition, used to paint the components. For simplicity, the concrete implementations of the abstract class `AbstractComponentPainter` are omitted.

The inheritance pattern with composition allows the construction of different inheritance trees in languages that do not support multiple inheritance. A disadvantage is the added runtime overhead. A single tree node instance will be composed out of several objects. Also, calls to these objects have to be forwarded through the node classes. A solution would be to merge inheritance trees when possible. This is discussed further in §6.3.3.

3.5.4 Expression Language Revisited

In §3.4, an example was presented which implemented an interpreter and a pretty printer for an expression language. In that section, only the external behaviour of the logic specification was discussed. The foregoing shows the actual structure of the generated code. In this section, the actually generated code for the example is discussed.

Figure 3.7 shows the generated classes for a language that does not support multiple inheritance. The classes on the left represent the AST nodes, the classes in the middle the pretty printer and the classes on the right the interpreter. These examples clearly show the inheritance pattern with composition, as it is applied by TPLc. The interfaces `IPrettyPrintable` and `IInterpreterExpression` specify the required methods for the pretty printer and interpreter respectively. These methods are implemented by the various `Pretty` and `Interpreter` classes. To make the behaviour available to the AST node classes, `Expression` contains a `InterpreterExpression` and a `PrettyExpression`. The other AST node classes fill these slots with instances of their corresponding functionality classes.

The class diagram also shows the difference between an `interface` method and an `implementation` method. Methods declared with the `interface` keyword, such as `eval`, are accessible on the AST nodes, whereas methods declared with `implementation`, such as `getOperator`, are not. The `eval` method is part of the `IInterpreterExpression` interface, which is implemented by `Expression`. This makes it possible to invoke `eval` on `Expression`. The `getOperator` method is only part of the classes that implement the pretty printer. It is not possible to invoke that particular method on one of the AST nodes.

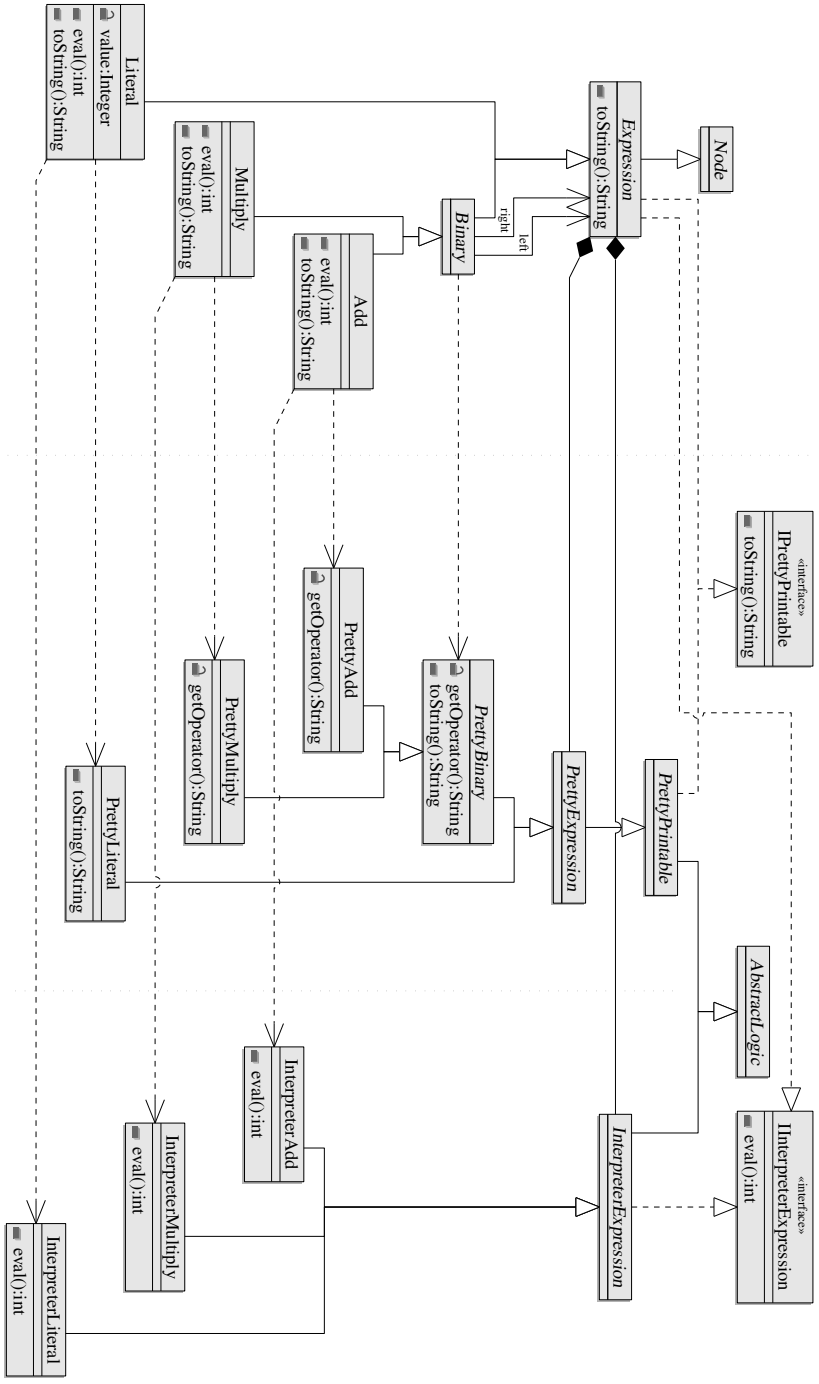

Figure 3.7: Functionality added to the AST nodes

Chapter 4
Language Specification

In this chapter, a specification of the syntax and semantics of TPL is given. §4.1 introduces TPL An example, that will be used throughout this chapter to exemplify the constructs of TPL, is given in §4.2. Next, in §4.3, a set of frequently used syntactical elements is discussed. §4.4 discusses the syntax and semantics of the tree definition, followed by the syntax and semantics logic specification in §4.5.

4.1 Introduction

TPL is a language in which the structure and functionality of a tree can be described. From this specification, a set of tree node classes is generated. TPL supports the generation of trees in different languages. The chosen language is called the target language[1]. TPL is especially useful for abstract syntax trees, that can be constructed by a parser front-end, but other applications (such as a user interface component tree) are also possible.

A specification in TPL consists of two parts: a single tree definition and zero or more logic specifications. The tree definition describes the type and structure of the AST, whereas a logic specification describes attributes and functionality that are part of a certain node.

4.2 Tutorial

In this section, a running example—a simple calculation language—will be presented. For this calculation language, a set of logic specifications is developed. The first is a simple context checker. It checks whether identifiers are declared before they are used. The second logic specification is an interpreter, which evaluates the input. The last specification is a pretty printer. Fragments of this example will also be used in this chapter to illustrate the concepts of TPL. The full code listing of the calculation language is available in appendix E.

The syntax of the calculation language is defined in grammar fragment 4.1. For this syntax, a grammar specification in ANTLR is developed. This grammar specification is provided in §E.3.

4.2.1 Tree Definition

The syntax shows that a ⟨*program*⟩ can contain zero or more ⟨*declaration*⟩s, and always has a single ⟨*statement-block*⟩. A ⟨*declaration*⟩ starts with the '`var`' keyword, followed by a single ⟨*identifier*⟩. The structure of the node types, that correspond to these production rules, is given in listing 4.1. Here a `Program` node type is declared, which contains zero or more `Declaration` nodes and a single

[1] Although TPL is setup to be language independent, a back-end needs to be available for the selected target language. Currently, only the Java target is supported.

Grammar Fragment 4.1 Example calculation language

⟨*program*⟩	::=	(⟨*declaration*⟩)* ⟨*statement-block*⟩
⟨*declaration*⟩	::=	'var' ⟨*identifier*⟩
⟨*statement*⟩	::=	⟨*expression*⟩ ';'
	|	⟨*if-statement*⟩
	|	⟨*print-statement*⟩
⟨*expression*⟩	::=	⟨*identifier*⟩
	|	⟨*number*⟩
	|	⟨*expression*⟩ '+' ⟨*expression*⟩
	|	⟨*expression*⟩ '-' ⟨*expression*⟩
	|	⟨*identifier*⟩ ':=' ⟨*expression*⟩
⟨*if-statement*⟩	::=	'if' '(' ⟨*expression*⟩ ')' '{' ⟨*statement-block*⟩ '}' 'else' '{' ⟨*statement-block*⟩ '}' ';'
⟨*print-statement*⟩	::=	'print' '(' ⟨*expression*⟩ ')' ';'
⟨*statement-block*⟩	::=	(⟨*statement*⟩)*
⟨*identifier*⟩	::=	A sequence of letters.
⟨*number*⟩	::=	A sequence of digits.

`StatementBlock`. The `Declaration` node only contains an `Identifier`. The 'var' literal is not part of the abstract syntax tree, because it adds no information to the `Declaration`. Where the 'var' token is needed in the parser to recognise a ⟨*declaration*⟩, the `Declaration` AST node can be distinguished by its type alone.

Listing 4.1: Program and Declaration tree nodes

```
Program(Declaration* declarations, StatementBlock statements) -> PROGRAM;
Declaration(Identifier variable) -> DECLARATION;
```

The next step is to declare the nodes for statements. To represent the commonality between the different statements, all statements extend from an abstract base `Statement` class. The node definitions for the statements is given in listing 4.2.

Listing 4.2: Statement tree nodes

```
abstract Statement;
IfStatement(Expression condition, StatementBlock thenPart,
    StatementBlock elsePart) extends Statement -> IF;
PrintStatement(Expression message) extends Statement -> PRINT;
StatementBlock(Statement* statements) extends Statement -> BLOCK;
```

Listing 4.3 shows the node definitions for the expressions of the calculation language. The first line shows that all expressions are also statements, something that can be deduced from the first alternative of the ⟨*statement*⟩ production rule. It

also shows AddExpression and MinusExpression inheriting the left and right children from BinaryExpression. AssignExpression also inherits these children, but renames left to var and changes its type to Identifier.

Listing 4.3: Expression tree nodes

```
abstract Expression extends Statement;
abstract BinaryExpression(Expression left, Expression right) extends Expression;
AddExpression extends BinaryExpression -> ADD;
MinusExpression extends BinaryExpression -> MINUS;
AssignExpression(left as Identifier var) extends BinaryExpression -> ASSIGN;
```

Finally, the literals are defined in listing 4.4. The first line defines a Literal class, parameterised with a type T[2]. This parameter represents the type of the value of the node. Both Number and Identifier subclass this node, where the first provides Integer for T and the latter provides String. In addition, both modify the type of the value.

Listing 4.4: Literal tree nodes

```
abstract Literal<T>(T value) extends Expression;
Number(value as Integer) extends Literal<Integer> -> NUMBER;
Identifier(value as String) extends Literal<String> -> IDENTIFIER;
```

4.2.2 Interpreter

The interpreter is implemented as a logic specification. The execution starts in the run method in Program. The Program node is shown in listing 4.5. The run method invokes the visiting method of Program. This method creates a variable store, visits all declarations and finally visits the statements child. The visiting method of Declaration adds the declared variable to the store, initialised to 0.

The Program node calls the visiting method of Statement. This method is declared in listing 4.6. Again, Statement is declared abstract and it provides the signature for the visiting methods of all statements. For example, IfStatement implements this method, where it checks the result of the condition and, depending on that result, visits thenPart or elsePart.

Listing 4.7 shows the implementation of the Expression and AssignExpression nodes. The main difference between a statement and an expression is that the former does not return a value. Therefore, Expression adds an abstract visiting method which does return a value. The visiting method inherited from Statement is implemented by invoking the new method and discarding the result. AssignExpression implements the new visiting method by evaluating the right-hand-side and storing the result in the store. Notice that AssignExpression directly extends Expression; there is no BinaryExpression.

4.2.3 Pretty Printer

In listing 4.8, the base classes of the pretty printer are shown. The class Printable is declared to be part of the base node class Node. Printable specifies an abstract visiting method that is responsible for pretty printing the node. It also overrides

[2]The use of type parameters is a feature of the target language Java, not of TPL

Listing 4.5: Interpreter for Program and Declaration

```
Program {
  interface public void run() {
    #this.visit();
  }

  visit() {
    Map<String, Integer> store = new HashMap<String, Integer>();
    #foreach(#decl <- #declarations) {
      #decl.visit(store);
    }
    #statements.visit(store);
  }
}

Declaration {
  visit(Map<String, Integer> store) {
    store.put(#variable.visitWrite(), 0);
  }
}
```

Listing 4.6: Interpreter for statements

```
abstract Statement {
  visit(Map<String, Integer> store);
}

IfStatement extends Statement {
  visit(Map<String, Integer> store) {
    if (#condition.visitExpr(store) != 0) {
      #thenPart.visit(store);
    } else {
      #elsePart.visit(store);
    }
  }
}
```

Listing 4.7: Interpreter for expressions

```
abstract Expression extends Statement {
  visit(Map<String, Integer> store) {
    #this.visitExpr(store);
  }

  int visitExpr (Map<String, Integer> store);
}

AssignExpression extends Expression {
  int visitExpr(Map<String, Integer> store) {
    int value = #right.visitExpr(store);
    store.put(#var.visitWrite(), value);
    return value;
  }
}
```

`toString` to call this visiting method. Another class, `IndentedConstruct` is introduced. This class contains a utility method to produce the correct indentation. Notice that this class is not part of an AST node.

Listing 4.8: Pretty printer base declarations

```
Node: abstract Printable {
  interface public String toString() {
    return #this.visit(0);
  }

  String visit(int indentation);
}

abstract IndentedConstruct extends Printable {
  implementation protected String writeIndentation(int indentation) {
    String ret = "";
    for (int count=0; count<indentation; count++) {
      ret += "  ";
    }
    return ret;
  }
}
```

The declarations for the statements of the pretty printer contain no interesting parts and are therefore skipped. In listing 4.9, the implementation for `AssignExpression` is shown. The `Expression` class extends `IndentedConstruct`, inheriting the `toString` and `writeIndentation` methods. The implementation of the visiting method generates the correct indentation and calls a special visiting method for expressions, which does not generate any indentation. `BinaryExpression` implements this method for all binary expressions. The `left` and `right` operand are printed, with the operator in between. Finally, `AddExpression` provides the `getOperator` method, specified by `BinaryExpression`, and returns the string `"+"`.

4.2.4 Context Checker

The most interesting part of the context checker for the calculation language is the visiting method for `Program`. This method returns the number of errors detected in the program. The declaration of this method is given in listing 4.10. The method can be split into two parts: checking the declarations and checking the usage. The actual checks are performed in the `visitDeclare` and `visitUse` methods.

This visiting method uses more complex node selections than the previous methods. Until now, only selections of the current node (`#this`) and of children (`#child`) where used. `#declarations/variable` returns the `variable` child of every declaration in `declarations`. The other node selection matches any child that is an `Identifier` (because of `*[Identifier]`) anywhere in the tree (`//`) under `statements`.

4.2.5 Running the Example

To run the example, a driver is needed. This driver parses the input file and constructs the AST. Next, the appropriate methods are called on the AST. The driver for the calculation language is given in §E.2. The most important line,

Listing 4.9: Pretty printer BinaryExpression

```
abstract Expression extends IndentedConstruct {
  String visit(int indentation) {
    return writeIndentation(indentation) + #this.visitExpr() + ";\n";
  }

  String visitExpr();
}

abstract BinaryExpression extends Expression {
  implementation abstract protected String getOperator();

  String visitExpr() {
    return #left.visitExpr() + " " + getOperator() + " " + #right.visitExpr();
  }
}

AddExpression extends BinaryExpression {
  implementation protected String getOperator() {
    return "+";
  }
}
```

Listing 4.10: Context checker for Program

```
  int visit() {
    int errors = 0;
    Set<String> symbols = new HashSet<String>();
    #foreach(#curId <- #declarations/variable) {
      errors += #curId.visitDeclare(symbols);
    }
    #foreach(#curId <- #statements//*[Identifier]) {
      errors += #curId.visitUse(symbols);
    }
    return errors;
  }
```

Listing 4.11: Setting the TreeAdaptor

```
    calcParser.setTreeAdaptor(
        new TPLTreeAdaptor(new calc.PackageDescription()));
```

concerning TPL, is where the `TreeAdaptor` is set, as displayed in listing 4.11. The `TreeAdaptor` is responsible for constructing the AST. `TPLTreeAdaptor` allows the construction of an AST written in TPL. It uses the information provided by the given `PackageDescription`, which is generated by TPLc.

Before the example can be compiled, the driver, parser, tree definition and logic specifications all need to be placed in a subdirectory called `calc`. Also, the class path needs to be setup correctly. ANTLR v3 and TPLc are required. When the environment is configured, the example can be compiled:

```
# java org.antlr.Tool calc/Calc.g
ANTLR Parser Generator  Version 3.0b6 (??)  1989-2007
# java -jar tplc.jar -report -parser calc.CalcParser calc/calc.tree \\
    calc/calccontext.logic calc/calcinterpreter.logic calc/calcpretty.logic
TPLc Tree Compiler 0.99
Parsing calc/calc.tree... done.
Parsing calc/calccontext.logic... done.
Parsing calc/calcinterpreter.logic... done.
Parsing calc/calcpretty.logic... done.
Checking tree definition... done.
Checking logic specifications... done.
Generating code... done.
# javac calc/CalcDriver.java
Note: ./calc/CalcParser.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

The first step calls ANTLR, which generates the lexer, the parser and a token file (a list of all tokens used in the grammar). Next, TPLc is called. The '`-parser calc.CalcParser`' option is used to indicate that the token file, generated by ANTLR, for `CalcParser` should be read. TPLc generates all AST classes. Finally, everything is compiled with the Java compiler. Now the example can be run:

```
# java calc.CalcDriver input.calc
Program is:
var a;
var b;
a := 3;
b := 5;
if (a - b) {
  print (a + b);
} else {
  print (a - b);
}

No errors found.
Running...
8
```

4.3 Common Syntactical Elements

TPL supports the generation of trees in different languages. The selected language is called the target language. However, only parts of the code are generated, the remaining code is taken from the tree definition and logic specifications. This means that both tree definition and logic specifications need support for fragments of the target languages, embedded in these grammars. Examples of these fragments are types, identifiers and method bodies and signatures.

The elements defined in grammar fragment 4.2 are fragments of the target language, used in both the tree definition and logic specifications. These elements are matched using their 'official' syntax; the same syntax as is used by the compiler of the target language. Examples for the Java target can be found in §5.3.1.

Grammar Fragment 4.2 Simple elements in the target language

⟨*identifier*⟩	::=	An identifier as defined by the target language.
⟨*target-class*⟩	::=	A class type in the target language.
⟨*target-type*⟩	::=	Any possible type in the target language, including classes, arrays and primitives.
⟨*package-name*⟩	::=	The name of a package or module in the target language.

In addition to these simple elements, that can normally be matched by their syntax definitions, TPL also uses some composite elements. These elements can contain large fragments of code in the target language, which makes them difficult to match using their exact syntax. These elements are therefore matched as (restricted) free form text. The composite elements are defined in grammar fragment 4.3.

Grammar Fragment 4.3 Composite elements in the target language

⟨*expression-action*⟩	::=	An expression in the target language.
⟨*method-sig-action*⟩	::=	A method signature in the target language.
⟨*method-body-action*⟩	::=	The body of a method in the target language, enhanced with macros (see §4.5.8).

TPL has some more syntactical elements that are used in both the tree definition and logic specification. These elements define headers and comments. The effect of these elements depends on the context in which they are used. Their syntax is defined in grammar fragment 4.4.

Grammar Fragment 4.4 Common elements

⟨*header*⟩	::=	`'header' '{'` ⟨*method-body-action*⟩▲4.3 `'}'`
⟨*node-header*⟩	::=	`'nodeheader' '{'` ⟨*method-body-action*⟩▲4.3 `'}'`
⟨*comment*⟩	::=	`'/*'` ⟨*any-char*⟩ `'*/'`

4.4 Tree Definition

The tree definition describes the structure and typing of the AST nodes. It contains a number of node definitions, which define a heterogeneous tree structure. Every node definition corresponds to a class. These node classes can be declared to contain children. The inheritance relation between the node classes can be

freely defined. When a parser is used as front-end, the tree definition can be used to specify the relation between token types, as used by the parser, and AST node classes. The tree definition only specifies the structure of the tree, it does not contain any behaviour.

This section defines the formal syntax and informal semantics of the tree definition of TPL. It begins with an explanation of the structure of a single AST node, in §4.4.1. This is followed by a discussion of types and node types in §4.4.2. Next, in §4.4.3, the top level structure of the tree definition is discussed. How a node is defined is explained in §4.4.4. TPLc generates a set of AST classes. The organisation of these classes is discussed in §4.4.5. The constructs, to which comments can be added, are discussed in §4.4.6. Finally, in §4.4.7, the coupling with a parser, in particular ANTLR, is explained.

4.4.1 The AST Node

A tree is constructed from several nodes. A distinction is made between internal nodes and leaf nodes. An internal node contains one or more children, whereas a leaf node does not contain any children at all. An example of an internal node is a binary expression, which contains two children. An example of a leaf node is a literal, such as an integer. In addition, an AST node also contains a reference to its parent.

A node can also contain a value. This value can be used to add information to the AST, which can not be encoded (easily) as a tree structure. Examples are literals (such as numbers and strings), names of identifiers, but also enumerations (such as operator types). When a parser is used as front-end, then the value of a node must have a type for which instances can be constructed from strings (see §4.4.7). This allows the parser to automatically instantiate the values of the nodes.

Child nodes are subtrees that are located under the node itself. Child nodes are identified by their name. For example, the binary expression 'plus' has two child nodes: left and right. In the case of the binary expression, the number of children is exactly two, but other multiplicities are also possible. A function definition for example might have zero or more modifiers, exactly one name, zero or more arguments and optionally a return type.

4.4.2 Types

The tree definition is used to specify the type and structure of the AST. Every node, defined in a tree definition, defines a new type: a node class. These are classes, defined in the target language, that extend the `Node` class. Only node classes can be used as children of a node.

TPL is based on a strictly typed tree. This means that for every child, the type is known. References to the children are typed and all access to the children is performed through these typed references. This makes it impossible to construct trees that do not conform to the structure defined in the tree definition.

When a parser, such as ANTLR, is used as a front-end, this parser often has its own notion of types of the AST nodes, commonly called token types. These token types are related to node classes, but are not the same. TPL allows a mapping from token types to the node class, that should be constructed when a token of the given type is matched.

Types in TPL are represented by the types in the selected target language. This implies that types in TPL follow the semantics of types in the target language.

4.4.3 Top Level Structure

A complete tree definition contains a package declaration, node definitions and optionally a header. This is specified in grammar fragment 4.5.

Grammar Fragment 4.5 Tree definition top level

⟨*tree-definition*⟩	::=	⟨*comment*⟩▲4.4.4? ⟨*package-declaration*⟩ ⟨*header*⟩▲4.4.4? ⟨*node-def*⟩▼4.6*
⟨*package-declaration*⟩	::=	'package' ⟨*package-name*⟩▲4.4.2 ';'

The `package` specification specifies the target package for the generated AST node classes. How this package is interpreted, depends on the target language. For Java, a `package` construct is added to the class and the file is placed in the corresponding directory. The `header` section is included in the header of every generated node class. This can, for example, be used for imports.

4.4.4 Node Definition

The formal syntax of a node definition is given in grammar fragment 4.6. A node definition consists at least of the node class that is declared. This class can extend another node class. If nothing is specified, it will extend the base node class `Node`. Optionally, one or more token types can be specified, for which the node class should be used. It is an error to specify the same token type in multiple node definitions.

Grammar Fragment 4.6 Node definition

⟨*node-def*⟩	::=	⟨*node-header*⟩▲4.4.4? ⟨*comment*⟩▲4.4.4? 'abstract'? ⟨*target-class*⟩▲4.4.2 ⟨*param-list*⟩▼4.7? ⟨*extends-clause*⟩? ⟨*token-types*⟩? ';'
⟨*extends-clause*⟩	::=	'extends' ⟨*target-class*⟩▲4.4.2
⟨*token-types*⟩	::=	'->' ⟨*token-type*⟩ (',' ⟨*token-type*⟩)*
⟨*token-type*⟩	::=	⟨*identifier*⟩▲4.4.2

A node class can be declared `abstract`. If it is, no instances of it can be made. It will only serve as a base class for other node classes. This also makes it impossible to specify token types for the node class.

Node definitions can have a parameter list. The formal syntax of the parameter list is given in grammar fragment 4.7. This parameter list defines the structure of the node. It defines whether the node has child nodes and/or a value. A node class extending another node class, for which the structure is already defined, will inherit this structure. It is an error to redefine the structure of a node class, that already inherited its structure, except when using the `as` keyword. Only node classes with a defined structure can be coupled to a token type.

With the `as` keyword, it is possible to modify a parameter, inherited from a superclass. Two types of modifications are allowed: (1) creating an alias and

Grammar Fragment 4.7 Tree node parameters

⟨*param-list*⟩	::=	'(' ⟨*parameters*⟩? ')'
⟨*parameters*⟩	::=	⟨*commented-param*⟩ (',' ⟨*commented-param*⟩)*
⟨*commented-param*⟩	::=	⟨*comment*⟩▲4.4.4? ⟨*param*⟩
⟨*param*⟩	::=	⟨*target-class*⟩▲4.4.2 'value'
	|	'value' 'as' ⟨*target-class*⟩▲4.4.2
	|	⟨*target-class*⟩▲4.4.2 ⟨*multiplicity*⟩? ⟨*identifier*⟩▲4.4.2
	|	⟨*identifier*⟩▲4.4.2 'as' ⟨*target-class*⟩▲4.4.2 ⟨*identifier*⟩▲4.4.2?
⟨*multiplicity*⟩	::=	'?' | '+' | '*'

(2) narrowing the type (using a subclass of the type). The optional ⟨*identifier*⟩ can be used to create an alias, which will make a child available under a different name. The child will also still be accessible under the old name. When the type of a child is narrowed in a subclass, the access to the child in this subclass will be of the new type.

Parameters, that define child nodes, can have a multiplicity defined. The supported multiplicities are: `?` for zero or one, `+` for one or more and `*` for zero or more. A node with no multiplicity defined, will have the multiplicity of exactly one.

The parameter keyword `value` is used to specify the type of the value of a node. At most a single definition of `value` can be present in a structure definition, and it should be the first. When a parser is used as front-end, the value will automatically be set. The parsed string, corresponding to the token, will be converted to an instance of the correct type (see §4.4.7).

When the AST is the result of a parser run, special care should be taken in defining the order of the children of a node class. Most importantly, the children should be defined in the same order as they are added by the parser. Furthermore, subsequent child node definitions should be unambiguous. This is explained in more detail in §4.4.7.

Listing 4.12: Node definitions

```
Program(Declaration* declarations, StatementBlock statements) -> PROGRAM;
abstract BinaryExpression(Expression left, Expression right) extends Expression;
AssignExpression(left as Identifier var) extends BinaryExpression -> ASSIGN;
abstract Literal<T>(T value) extends Expression;
Number(value as Integer) extends Literal<Integer> -> NUMBER;
```

Listing 4.12 show several node definitions, taken from the calculation language. The first defines a node called `Program`, which contains zero or more declarations and a `StatementBlock`. `BinaryExpression` is an `abstract` base definition, for all expressions that have a left and right operand. One of these expressions is `AssignExpression`, which redefines `left` as an `Identifier`. `Literal` is an `Expression` with a value. The type of the value is specified as a type parameter (a Java 5 feature). `Number` provides `Integer` for this type parameter. It also specifies that the value is now an `Integer`. This is needed, because TPL does not know anything about the types used. When omitted, `Number` will get a `value` with the incorrect type `T`.

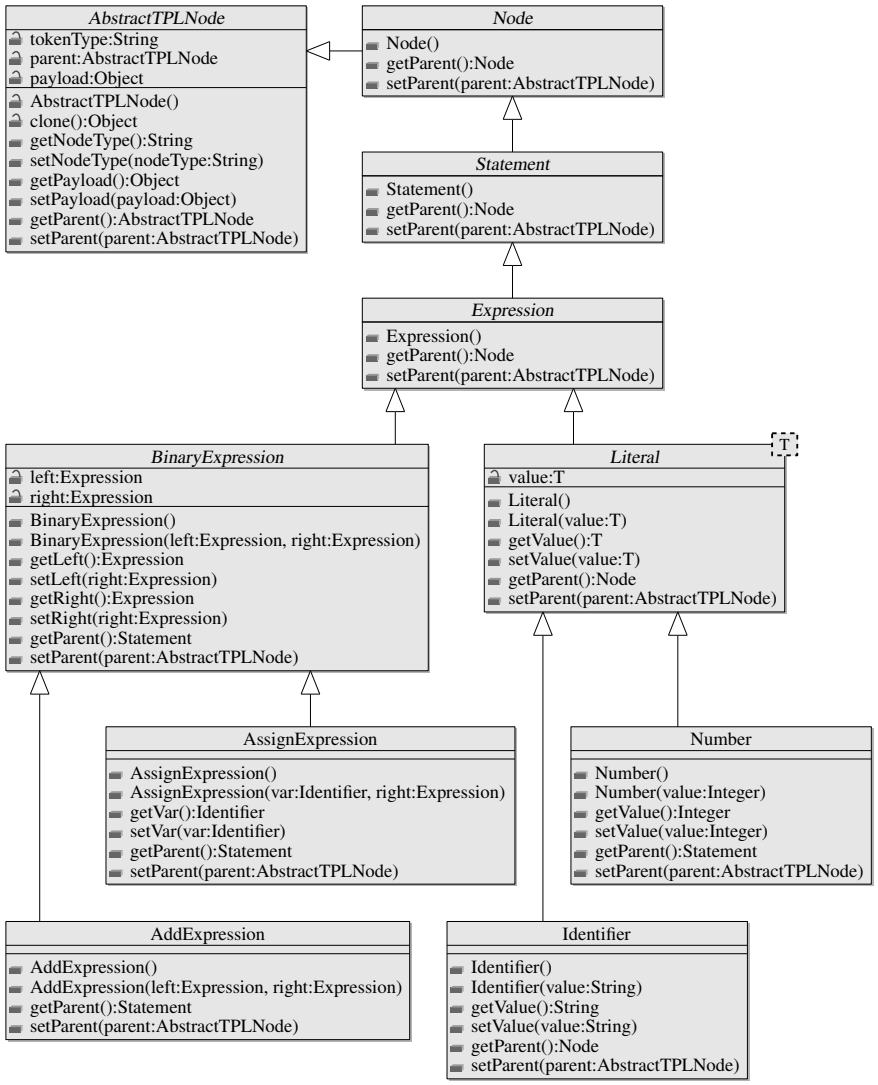

Figure 4.1: Class diagram for tree definition

4.4.5 The Structure of the Generated AST

The tree definition defines the structure of the AST. The different constructs are used to generate a node class hierarchy. An example of such a generated class hierarchy is displayed in figure 4.1.

For every node definition, a single class is generated. These classes are placed in the declared package. The classes all indirectly extend from the `AbstractTPLNode` class, which provides the basic functionality for all node types. When node A extends node B, node class C_A, generated from A's node definition, will be a subclass of C_B, generated from B's node definition. A node type that is declared `abstract` results in an `abstract` node class.

Every generated node class has the following components:

Constructors At least a constructor with no arguments is generated. When the structure of a node is defined, an additional constructor is generated that takes all children as arguments.

Typed Fields A node class, that defines a structure, has instance fields, that contain the references to the children of that node. Fields are generated for the value and all child nodes. These fields are of the type specified in the definition. All fields are `protected`.

Accessor methods For every component of the declared structure, a set of accessor methods is generated. For the multiplicities 'exactly-one' and 'zero-or-one', this will be a `get` and `set` method. For the multiplicities 'zero-or-more' and 'one-or-more', an `add`, `remove` and `set` method is generated. The `add` and `set` methods all call `setParent` on the new child, updating the parent of the node.

Parent accessor Every node that can have a parent (i.e. a node type that is not used as a child in any of the node definitions) receives `getParent` and `setParent` methods, which provide access to the parent of the node. The type of node returned by the `getParent` method is the most specific type that still contains all possible parents (see below). `setParent` checks at runtime if the specified parent is of the correct type[3]. When the node can not have a parent, both `getParent` and `setParent` are overridden to throw an exception.

Annotations Every node class is annotated with the `Structure` annotation, which describes the structure of the node. This annotation contains information about the children of the node and the token types to which the node is bound. When the target language does not support annotations, a static field can be used to store the information.

For nodes with a multiplicity of 'zero-or-more' or 'one-or-more' a list is generated. The two other multiplicities are compiled into a single instance. An empty list or `null` is used to indicate a missing node for the two cases respectively.

Modifications to the structure, made with `as`, are taken into account when the nodes are generated. When an alias is created, extra accessor methods are added. This can be seen in the `AssignExpression` class in figure 4.1. When the type is narrowed, this modified type is used in the constructor and accessor methods. This is also illustrated by the `AssignExpression` class, where the constructor takes an `Identifier`, and both `getVar` and `setVar` use `Identifier`, not `Expression`.

4.4.5.1 Parent Type

The `getParent` method returns the parent of a node, if it exists. Nodes often only occur in a fixed context, making it desirable to return a correctly typed parent. The return type should be as specific as possible. This approach is taken by TPLc for the generated `getParent` and `setParent` methods.

The union of all contexts in which a node can be used is taken and the highest common denominator is taken as the type of the parent. Literals and variable declarations in the calculation language clearly illustrate this technique. Listing 4.13 gives their tree definitions.

A `Declaration` can only occur as part of a `Program` node. The type of `Declaration`'s parent will therefore be `Program`. `Identifier` on the other hand can be used inside a `StatementBlock` (as statement) or inside a `Declaration`. The highest common denominator of these two classes is `Node`, which becomes the type of `Identifier`'s parent. A `Literal` can be either an `Identifier` or a `Number`

[3] `setParent` is not overridden with a more strictly typed argument, because that would introduce an overloaded method. The old method (taking an `AbstractTPLNode`) would still exist and would be called in case of a wrongly typed argument. The overriding method does check if the argument is of the correct type.

Listing 4.13: Parent type illustration

```
Program(Declaration* declarations, StatementBlock statements) -> PROGRAM;
Declaration(Identifier variable) -> DECLARATION;
StatementBlock(Statement* statements) extends Statement -> BLOCK;

abstract Statement;
abstract Expression extends Statement;
abstract Literal<T>(T value) extends Expression;

Number(value as Integer) extends Literal<Integer> -> NUMBER;
Identifier(value as String) extends Literal<String> -> IDENTIFIER;
```

Construct	Destination
Package declaration	Package
Node definition	Tree node class
Node parameter	Field in the tree node class

Table 4.1: Comments in the tree definition and their destination

and thus can be used in the contexts of both types. The type of the parent of `Number` is `Statement`. The type of `Literal`'s parent will be the common superclass of `Statement` and `Node`: `Node`.

4.4.6 Comments and Headers

Comments can be added to most constructs of the tree definition. The constructs where comments are supported can be found in table 4.1. This table also shows the destination of the comments. The contents of a comment will be copied in verbatim to the output file.

It is not possible to add comments to the various accessor methods and constructors. For these methods comments are generated that refer to the comments for the parameters.

TPL also supports comments that are not copied to the generated files. These comments start with '//' and stop at the end of the line.

It is also possible to add headers to either a single node, or to all nodes in the tree definition. These headers are placed at the beginning of the generated files.

4.4.7 AST Construction

ASTs can be constructed through manual calls of the constructors of the nodes, but a more common approach is to let a parser construct the AST. TPL provides an interface with ANTLR, which allows ANTLR to construct an instance of an AST generated by TPLc.

ANTLR v3 uses the `TreeAdaptor` interface to construct ASTs. TPL provides a custom implementation of this interface to create the nodes. This adaptor uses the types of the nodes to determine the parameter to which new nodes should be added. The information it needs is added to the generated classes in the form of annotations. All AST construction techniques available in ANTLR are supported.

To convert the tokens, read from the input, to the values in the nodes, TPL uses the `ConversionSelector`. The default implementation of this selector is able to convert a string to all formats that can be represented by the primitive types

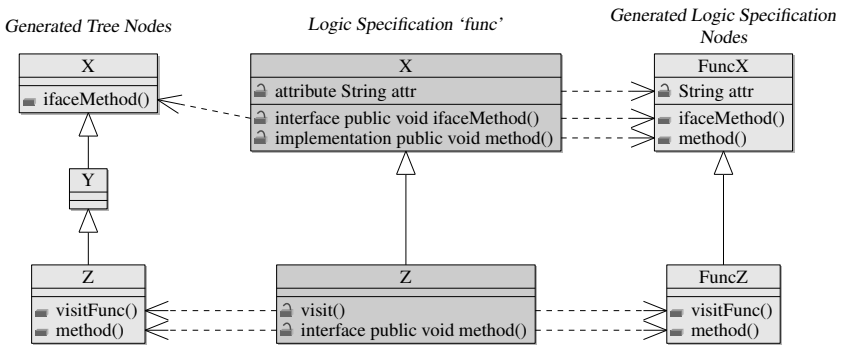

Figure 4.2: Binding of a logic specification

the target language supports. The Java implementation supports `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`, and `Boolean`. It is also possible to specify different conversions through the `addConverter` method. When the conversion fails (either because the input does not have the right format, or because the converter is not available) a `TypeConversionException` is thrown.

Nodes are added one by one to their parent in a sequential order. At any moment it should be clear to which parameter a new sub-node should be added. For example, the following node definition is invalid: `N1(A a, B? b1, C* c, B b2)`, whereas this definition is valid: `N2(A a, B b2, C* c, B? b1)`. In the first example, after an `A` has been inserted, it is unclear whether a `B` node should be inserted as parameter `b1` or as `b2`. It might be possible that both `b1` and `c` are not present in the input.

These ambiguities are detected when the tree definition is processed and reported as errors. In most cases ambiguities such as above can be resolved using ANTLR's tree rewrite rules.

4.5 Logic Specification

Logic specifications are used to add application functionality to AST nodes. This includes attributes (instance fields), methods and interfaces. These are added to the AST nodes, defined in the tree definition. A logic specification can depend on other logic specifications, which will also be part of the AST.

A logic specification defines a separate class inheritance tree. Instances of these classes are put in the tree node[4] classes and accessible through composition. This is illustrated by figure 4.2. In this figure the binding of logic specification node classes `X` and `Z` to the tree node classes `X` and `Z` is shown. The arrows indicate the destination of the declared constructs. The class inheritance tree is backed by a similar interface inheritance tree (not shown in the figure) that contains all interface methods. These interfaces are implemented by the tree node classes and the logic node classes, making the functionality accessible from the outside. The methods in the tree node classes call their equivalent in the corresponding logic node class.

This section defines the formal syntax and informal semantics of the logic

[4]The term 'node' is somewhat ambiguous. It can refer to the concept of a node in an AST, to the class implementing that concept, or to a part of a logic specification that adds functionality to that class. When the actual meaning is not clear in the context the term is used in, *tree node class* is used to refer to the class implementing a node in the tree and *logic node class* is used to refer to the corresponding part of the logic specification.

specification. First, the top level structure of the logic specification is given in §4.5.1. Followed by the specification of a logic node class in §4.5.2. Next, the elements that form the body of a node, attributes and methods, are explained in §4.5.3 to §4.5.6. Inheritance between nodes is discussed in §4.5.7. In method bodies, several macros can be used. These macros are discussed in §4.5.8. §4.5.9 discusses how the logic specification is compiled into a representation in the target language. Finally, §4.5.10 explains the usage of comments and headers in a logic specification.

4.5.1 Top Level Structure

The structure of a logic specification is very similar to that of the tree definition. The syntax for a logic specification is defined in grammar fragment 4.8.

Grammar Fragment 4.8 Logic specification top level

⟨*specification*⟩	::=	⟨*comment*⟩$^{\blacktriangle 4.4}$? ⟨*package-declaration*⟩ ⟨*header*⟩$^{\blacktriangle 4.4}$? ⟨*logic-deps*⟩? ⟨*node-declaration*⟩$^{\blacktriangledown 4.9}$*
⟨*package-declaration*⟩	::=	`'package'` ⟨*package-name*⟩$^{\blacktriangle 4.2}$ `';'`
⟨*logic-deps*⟩	::=	`'depends'` ⟨*identifier*⟩$^{\blacktriangle 4.2}$ (`','` ⟨*identifier*⟩$^{\blacktriangle 4.2}$)* `';'`

The ⟨*package-declaration*⟩ specifies the target package for the generated classes and interfaces. The `header` section is included in every logic node class, its accompanying interface and in tree node classes if needed (when a logic node is bound to a tree node). With the `depends` keyword, other logic specifications that the specification depends on, can be given. A dependency may only be listed once.

4.5.2 Node Declaration

Functionality in a logic specification is provided by means of logic node classes. The tree node classes acquire this functionality by including the logic node classes through composition. The syntax for the logic node classes is given in grammar fragment 4.9.

The syntax of a logic node specification is similar to that of a tree node definition. At least the name of the node class needs to be given. Using the ⟨*extends-clause*⟩, it is possible to let this node class extend another class. When no superclass is specified, the node class will extend `AbstractLogic` (see §4.5.7). The optional ⟨*node-name*⟩, followed by a colon, allows the binding of a logic node class to a tree node class. This will add the functionality provided by the logic node class to the tree node class. The tree node class should be defined in the tree definition (§4.4.4). When the ⟨*node-name*⟩ is omitted, the name of the logic node class is used. When this name does not match the name of a tree node, the logic node is not bound to a tree node. It is not allowed to bind more than one logic node class to a single tree node class in the same logic specification. Finally, the logic node class can have a body, that contains attributes (§4.5.3), interface methods (§4.5.4), implementation methods (§4.5.5) and visiting methods (§4.5.6). The elements ⟨*node-header*⟩ and ⟨*comment*⟩ are discussed in §4.5.10.

The example in listing 4.14 declares the `Program` and `AssignExpression` nodes for the interpreter of the calculation language.

Grammar Fragment 4.9 Logic node specification

⟨*node-declaration*⟩	::=	⟨*node-header*⟩▲4.4?
		⟨*comment*⟩▲4.4?
		(⟨*node-name*⟩ ':')?
		'abstract'? ⟨*target-class*⟩▲4.2
		⟨*extends-clause*⟩?
		('{' ⟨*body*⟩ '}' | ';')
⟨*node-name*⟩	::=	⟨*target-class*⟩▲4.2
⟨*extends-clause*⟩	::=	'extends' ⟨*target-class*⟩▲4.2
⟨*body*⟩	::=	(⟨*comment*⟩▲4.4? ⟨*body-element*⟩)*
⟨*body-element*⟩	::=	⟨*attribute*⟩▼4.10
	|	⟨*interface-method*⟩▼4.11
	|	⟨*implement-method*⟩▼4.12
	|	⟨*visit-method*⟩▼4.13

Listing 4.14: Node declarations

```
Program {
  interface public void run() {
    #this.visit();
  }

  visit() {
    Map<String, Integer> store = new HashMap<String, Integer>();
    #foreach(#decl <- #declarations) {
      #decl.visit(store);
    }
    #statements.visit(store);
  }
}

abstract Expression extends Statement {
  visit(Map<String, Integer> store) {
    #this.visitExpr(store);
  }

  int visitExpr (Map<String, Integer> store);
}

AssignExpression extends Expression {
  int visitExpr(Map<String, Integer> store) {
    int value = #right.visitExpr(store);
    store.put(#var.visitWrite(), value);
    return value;
  }
}
```

`Program` declares an interface method `run` which calls the visiting method in `Program`. This visiting method creates a new store, adds all `declarations` to the store and visits all `statements` (see §4.5.6).

The `AssignExpression` node extends `Expression` (left out) and implements a visiting method that stores the value of the `right` expression under the name of the `var` identifier in the store. The value is also returned from the visiting method.

When a logic node is bound to a non-abstract tree node class, all its interface methods should have a corresponding implementation. The tree node itself will be available to the functionality through the macro `#this` (see §4.5.8). When a logic node class is declared `abstract` and it is bound to a tree node class, then the tree node class will only inherit its interface methods. Because no implementation and composition is added, it is required that the tree node class is also abstract. A subclass of the logic node should provide all abstract methods and should be bound to a subclass of the tree node (see §4.5.7).

4.5.3 Attributes

Attributes are member fields that can be used to store information in a node. For example, a variable node might get a reference to its symbol table entry. An attribute can also have an initial value. The grammar used to add an attribute to a logic class is given in grammar fragment 4.10.

Grammar Fragment 4.10 Attribute specification

⟨*attribute*⟩	::= 'attribute' ⟨*target-type*⟩▲4.2 ⟨*identifier*⟩▲4.2 ('=' ⟨*expression-action*⟩▲4.3)? ';'

⟨*target-type*⟩ is the type of the attribute declared in the target language. The expression ⟨*expression-action*⟩ is also specified in the target language and can be used to give the attribute an initial value. Listing 4.15 shows two correct attribute definitions when Java is the target language[5].

Listing 4.15: Attribute declaration

```
attribute SymbolEntry symbolReference;
attribute Type variableType = Type.UNKNOWN;
```

4.5.4 Interface Methods

To access behaviour of a node through its public interface, an interface method should be added. An interface method exposes behaviour of a node through a method in the tree node class. The method call on the tree node class is forwarded to the logic node class. Examples of behaviours that can be exposed are access to attributes, type information and specialised string serialisation logic. Visiting methods are always part of the interface of a node (see §4.5.6). The syntax, used for interface methods, is given in grammar fragment 4.11.

As with attributes, both ⟨*method-sig-action*⟩ and ⟨*method-body-action*⟩ are actions in the target language, declaring the interface method. Listing 4.16 gives two examples, from the calculation language, that declare interface methods for the Java language.

[5]These examples are not part of the calculation language.

Grammar Fragment 4.11 Interface method specification

⟨*interface-method*⟩ ::= 'interface' ⟨*method-sig-action*⟩$^{\blacktriangle 4.3}$
(';' | '{' ⟨*method-body-action*⟩$^{\blacktriangle 4.3}$ '}')

Listing 4.16: Interface method declaration

```
interface public String toString() {
  return #this.visit(0);
}

interface public void run() {
  #this.visit();
}
```

An interface method without a body is declared abstract. The method will be part of the interface, but its implementation should be provided in all subclasses. In these subclasses, the method should also be provided as an `interface` method. It is not possible to remove a method from the public interface in a subclass. When a method is declared with `implementation` (see §4.5.5), it is possible to override it with `interface`. This will add the method to the public interface in the subclass, where it was previously not exposed.

Inside ⟨*method-body-action*⟩ it is possible to use several macros for accessing child nodes and visiting methods (see §4.5.8).

4.5.5 Implementation Methods

An implementation method adds functionality to the node, that is not accessible through the public interface of the tree node classes. In contrast to interface methods, no forwarding call from the tree node class to the logic node class is added for implementation methods. The grammar used to add implementation methods is given in grammar fragment 4.12.

Grammar Fragment 4.12 Implementation method specification

⟨*implement-method*⟩ ::= 'implementation' ⟨*method-sig-action*⟩$^{\blacktriangle 4.3}$
(';' | '{' ⟨*method-body-action*⟩$^{\blacktriangle 4.3}$ '}')

Listing 4.17 shows some implementation method specifications, from the calculation language's pretty printer, for the Java target.

4.5.6 Tree Traversal

Both the interface and implementation methods allow functionality to be attached to a single node type. However, algorithms often operate on the entire tree, or a part of it. To define functionality that is part of a tree traversal, a visiting method can be used. A visiting method is always part of the public interface of a node. It is declared with the syntax given in grammar fragment 4.13.

A visiting method can have one or more return values. When multiple return values are specified these should be returned and read with the designated macros (see §4.5.8).

Listing 4.17: Implementation method declaration

```
implementation protected String writeIndentation(int indentation) {
  String ret = "";
  for (int count=0; count<indentation; count++) {
    ret += "  ";
  }
  return ret;
}

implementation abstract protected String getOperator();

implementation protected String getOperator() {
  return ":=";
}
```

Grammar Fragment 4.13 Visiting method specification

⟨*visit-method*⟩	::=	⟨*visit-signature*⟩ (';' | '{' ⟨*method-body-action*⟩▲4.3 '}')
⟨*visit-signature*⟩	::=	⟨*return-type*⟩? ⟨*identifier*⟩▲4.2 '(' ⟨*argument-list*⟩? ')' ⟨*throws-clause*⟩?
⟨*throws-clause*⟩	::=	'throws' ⟨*target-type*⟩▲4.2 (',' ⟨*target-type*⟩▲4.2)*
⟨*return-type*⟩	::=	⟨*target-type*⟩▲4.2 | '(' ⟨*named-type*⟩ (',' ⟨*named-type*⟩)* ')'
⟨*named-type*⟩	::=	⟨*target-type*⟩▲4.2 ⟨*identifier*⟩▲4.2
⟨*argument-list*⟩	::=	⟨*named-type*⟩ (',' ⟨*named-type*⟩)*

It is possible to override the return type of a visiting method in a subclass. When the visiting method in the superclass declares only a ⟨*target-type*⟩, the subclass can also only declare a ⟨*target-type*⟩. This type should be a subclass of the return type of the method in the superclass. When the visiting method in the superclass declares multiple return values, or a single value with a name, the subclass is allowed to add new values and to change the type of existing values. Again a new type should be a subclass of the old type.

Adding new method arguments or changing the type of an argument in a subclass is not allowed. An external class might use the subclass with the type of the superclass. This would cause the method to be called with insufficient arguments. For example, a program might iterate over all expressions. At that point, the program only knows about the signature of the method, as it is defined on `Expression`. When one of these expressions (such as `Identifier`) required an additional method argument the program would fail to provide it, because it accesses the identifier through the signature provided in `Expression`.

⟨*method-body-action*⟩ specifies the behaviour of the visiting method. It is specified in the target language, enriched with the macros defined in §4.5.8. It is possible to omit the method body, in which case an abstract visiting method is declared, which needs to be implemented in all subclasses.

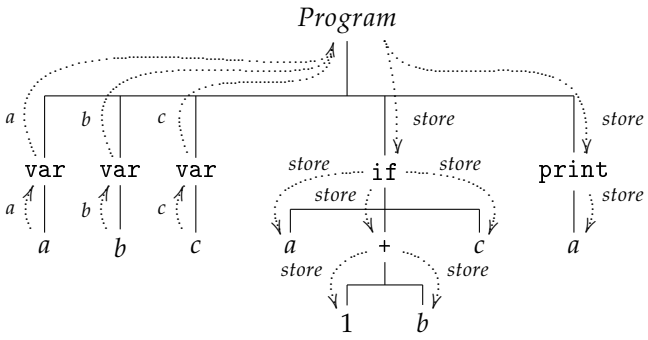

Figure 4.3: Inherited and synthesised attributes

Inside the visiting logic it is possible to pass information to nodes deeper in the tree, or up towards the root. This is related to attribute grammars, where information is passed through attributes. An attribute passed down the tree is called an inherited attribute and an attribute passed upward is called a synthesised attribute. With visiting methods, inherited attributes are passed as method arguments and synthesised attributes as return values. Both attributes are illustrated in figure 4.3. Arrows from the leafs toward the root are synthesised attributes. In this example they pass the name of a declared variable upward. The arrows from the root toward the leafs are inherited attributes. They pass the store with the values of variables to all parts of the program. Expression results are omitted from the figure. A comparison between visiting methods and attribute grammars is presented in appendix D.

4.5.7 Inheritance

A logic specification can define an inheritance tree that is independent of the tree defined by either the tree definition, or any other logic specification. As mentioned before, this is achieved through a separate inheritance tree, an accompanying interface inheritance tree and composition. Inheritance among logic specification classes follows the semantics of the target language.

There are some restrictions to the independence of the inheritance tree defined in a logic specification. When an interface is bound to a tree node class at a certain point, all tree node classes extending—directly or indirectly—this node, will inherit this interface. When another interface is bound to one of these subclasses, this interface should be a subinterface of the first.

This is illustrated by `Expression` in the interpreter of the example language. The corresponding code fragments are given in listing 4.18.

The `Statement` specifies a visiting method that takes a store and returns nothing. Expressions can also be used as statements in this language. Therefore `Expression` also needs to specify the visiting method declared by `Statement`. In addition it also declares a new visiting method that returns the result of the expression.

4.5.8 Macros in Logic Actions

Both the implementation and visit method bodies are specified in the target language. To aid the developer in accessing various properties of the AST, such as nodes and other logic specifications, various macros can be used. These macros are expanded to code in the target language. When possible, error checking is

Listing 4.18: Inheritance within a logic specification

```
abstract Statement {
  visit(Map<String, Integer> store);
}

abstract Expression extends Statement {
  visit(Map<String, Integer> store) {
    #this.visitExpr(store);
  }

  int visitExpr (Map<String, Integer> store);
}
```

performed, but these checks are limited, because not much is known about the context in which the macros are used.

4.5.8.1 Iterating over Nodes

The for-each statement can be used to loop over multiple nodes. The syntax for the for-each statement is given in grammar fragment 4.14.

Grammar Fragment 4.14 For-each statement

⟨*for-each*⟩	::=	'#' 'foreach' '(' '#' ⟨*iterator*⟩ '<-' ⟨*node-selection*⟩ ▼4.15 ')'
⟨*iterator*⟩	::=	⟨*identifier*⟩ ▲4.2

The nodes are made available one at a time as the first identifier. `#foreach` behaves as a normal `for` in the target language and, as such, can be followed by a statement block.

Listing 4.19 gives some exemplary for-each statements, as they can be used in an application. The first iterates over all `Statements` and calls the `visitStmt` method on every `Statement`. The second iterates over all `Identifiers` under `statements` and calls the `visit` method on every `Identifier`.

Listing 4.19: For-each statement

```
#foreach(#stmt <- #statements) {
  ret += #stmt.visit(indentation);
}

#foreach(#curId <- #statements//*[Identifier]) {
  errors += #curId.visitUse(symbols);
}
```

4.5.8.2 Selecting Nodes and Values

It is often necessary to access nodes when implementing certain behaviour. These nodes can range from the current node to any node anywhere in the tree. TPL

provides a query language through which nodes can be accessed. Node selection queries always start with the # symbol. The syntax of this query language is given in grammar fragment 4.15.

Grammar Fragment 4.15 Node selection

⟨*node-selection*⟩	::= '#' ⟨*selection-step*⟩ (⟨*step-operator*⟩ ⟨*selection-step*⟩)*
⟨*step-operator*⟩	::= '/' | '//'
⟨*selection-step*⟩	::= ⟨*selector*⟩ ⟨*filter*⟩?
⟨*selector*⟩	::= ⟨*identifier*⟩▲4.2 | 'this' | 'value' | 'parent' | '*'
⟨*filter*⟩	::= '[' ⟨*identifier*⟩▲4.2 (',' ⟨*identifier*⟩▲4.2)* ']'

The most commonly used queries consist of a single ⟨*selection-step*⟩. They are used to select direct children of a node (such as `#left`), the current node (with `#this`), the value of the current node (with `#value`) or its parent (with `#parent`). The wildcard ⟨*selector*⟩ '*' selects all children.

Subsequent ⟨*selection-step*⟩s can be concatenated with a ⟨*step-operator*⟩. Two ⟨*step-operator*⟩s are available. The first, '/', continues a query at the node that is currently selected, whereas the second, '//', continues at the current node, or any of its direct or indirect children.

A ⟨*filter*⟩ can be applied to a ⟨*selection-step*⟩, to only allow nodes of a given set of types. These types should be node classes, defined in the tree definition. If no ⟨*filter*⟩ is specified, all nodes are selected.

Normally, a node selection starts at the current node. When an iterator from a for-each statement (see §4.5.8.1) is used as the first ⟨*selection-step*⟩, the node selection will start at the iterator, instead of the current node. An iterator can only be specified as the first ⟨*selection-step*⟩.

The node selection can also be used to access the value of a node, with the 'value' ⟨*selector*⟩. The selection of the 'value' of a node is only allowed as the last ⟨*selection-step*⟩. Also, the node selection should always match exactly one value.

Listing 4.20: Node selection

```
#parent
#declarations/variable
#statements//*[Identifier]
#this//*[AssignExpression]/var
```

Listing 4.20 gives some node selection examples. The first selects the parent of the current node. The second selects the variable under every declaration. The third searches the entire tree under `statements` for nodes of type `Identifier`. The last query matches all `var` nodes under a `AssignExpression`.

Semantics The semantics of a node selection are defined in $1..m$ steps. A step is defined as $\sigma_i =< s_i, T_i, o_i, \sigma_{i+1} >$ and the last step as $\sigma_m = \bot$, where s_i is the

selection, T_i the filters applied and o_i the operator that seperates this step from the next.

The function $A(n, s)$ defines the advancement for a node and a selection:

$$A(n,s) = \begin{cases} \{n\} & \text{if } s = \text{`this'} \\ P(n) & \text{if } s = \text{`parent'} \\ V(n) & \text{if } s = \text{`value'} \\ C(n) & \text{if } s = \text{`*'} \\ \{x \in C(n) \mid N(x) = s\} & \text{if } \tau(s) = \langle \textit{identifier} \rangle \end{cases}$$

This advance function uses several utility functions. These functions are defined as:

$$P(n) = \begin{cases} \varnothing & \text{if } n \text{ does not have a parent} \\ \{\text{the parent of } n\} \end{cases}$$

$$V(n) = \begin{cases} \varnothing & \text{if } n \text{ does not have a value} \\ \{\text{the value of } n\} \end{cases}$$

$$C(n) = \text{A set of all children of } n\text{, excluding its value}$$

$$N(n) = \text{The name under which a child is stored in its parent}$$

Every node selection starts with a selection set S containing just the current node or an iterator (see §4.5.8.1). The function A is applied to every node in the current selection set and the results are merged into a single set. The filters are applied. The following function executes a single step σ_i, using the current selection set S, the selection s_i and filters T_i:

$$S'(S, s_i, T_i) = \left\{ m \mid m \in \bigcup_{n \in S} A(n, s_i) \wedge \tau(m) \in T_i \right\}$$

Between the steps, the $\langle \textit{step-operator} \rangle$ determines with which nodes the algorithm will continue. The semantics of the $\langle \textit{step-operator} \rangle$ is defined by the following function on the current selection set S and the operator o_i:

$$O(S, o_i) = \begin{cases} S & \text{if } o_i = \text{`/'} \\ S \cup \bigcup_{n \in S} D(n) & \text{if } o_i = \text{`//'} \end{cases}$$

$$D(n) = C(n) \cup \bigcup_{n' \in C(n)} D(n')$$

The execution $\varepsilon(S, \sigma_i)$ of a entire node selection becomes:

$$\varepsilon(S, \bot) = S$$

$$\varepsilon(S, < s_i, T_i, o_i, \sigma_{i+1} >) = \varepsilon(O(S'(S, s_i, T_i), o_i), \sigma_{i+1})$$

4.5.8.3 Visiting Nodes

Calling a visiting method is similar to calling a normal method. It should always follow a node selection as defined above. All attributes should always be provided in the specified order. The complete syntax for visiting method invocation is given in grammar fragment 4.16. Listing 4.19 gives some visit invocation examples.

Visiting methods can return multiple values, as specified in §4.5.6. Special macros are available for constructing and reading these values. As with the visit method invocation, these attributes should be given in the specified order. The syntax for these macros is given in grammar fragment 4.17.

Grammar Fragment 4.16 Visit invocation

⟨*visit-invocation*⟩	::=	⟨*node-selection*⟩▲4.15 '.' ⟨*identifier*⟩▲4.2 '(' ⟨*arguments*⟩ ')'
⟨*arguments*⟩	::=	⟨*expression-action*⟩▲4.3 (',' ⟨*expression-action*⟩▲4.3)*

Grammar Fragment 4.17 Return values

⟨*multi-val-create*⟩	::=	'$' '(' ⟨*arguments*⟩▲4.16 ')'
⟨*multi-val-store*⟩	::=	'$' ⟨*identifier*⟩▲4.2 '=' ⟨*visit-invocation*⟩▲4.16
⟨*multi-val-read*⟩	::=	'$' ⟨*identifier*⟩▲4.2 '.' ⟨*identifier*⟩▲4.2

The ⟨*multi-val-create*⟩ construct is used to create a tuple with the given values. With ⟨*multi-val-store*⟩, the result of a visiting method is stored in a local variable with the given name. Finally, with ⟨*multi-val-read*⟩, the values can be read from this variable.

Listing 4.21: Multiple return values

```
Program {
  interface public void example() {
    #foreach(#decl <- #declarations) {
      $declRet = #decl.visitNameType();
      System.out.println($declRet.name + ": " + $declRet.type);
    }
  }
}

Declaration {
  (Integer value, String type) visitNameType() {
    return $(#variable/value, "int");
  }
}
```

Listing 4.21 gives some examples[6] of using multiple return values. In `Declaration`, a tuple is constructed, containing the value, and the string `"int"`. `Program` iterates over all declarations, storing the result of `visitNameType` in the local variable `declRet`, and printing its contents.

4.5.9 The Structure of the Generated AST

Every logic specification is compiled into a separate class hierarchy. Instances of these classes have a reference to the tree node to which they belong and are part of these nodes using composition. Figure 4.4 shows a fraction of the class diagram for the interpreter of the example language. This figure shows that `AbstractLogic` contains the reference to a tree node, of type `N`. `InterpreterPro-`

[6] These examples are not from the calculation language. This language is too simplistic to use multiple return values.

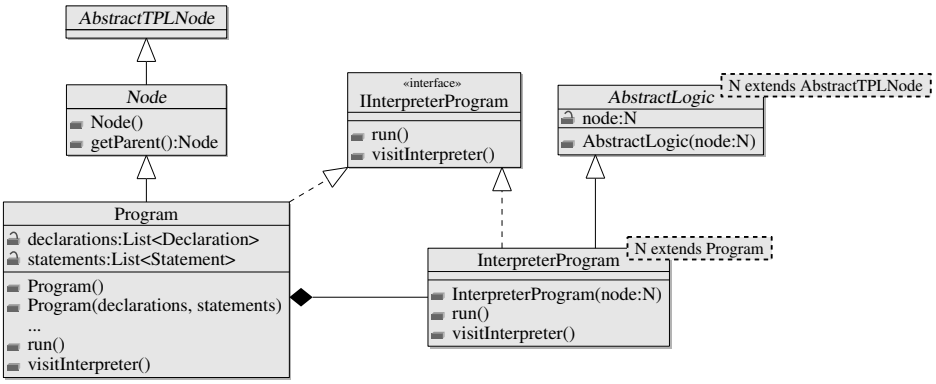

Figure 4.4: Class diagram for logic specification

gram provides `Program` for this type parameter `N`. `Program` accesses `InterpreterProgram` through composition. Instantiation of the composition classes is done automatically when the functionality is first used.

The logic specification enhances the node classes, generated by the tree definition. Three additions are made to the node classes, for every logic specification, that adds functionality to a node: (1) the node class implements the interface (if any) specified by the logic specification, (2) a `protected` field, containing a reference to the logic node class, is added, where needed, when the tree node class is not declared `abstract` and (3) all interface methods are passed to that field (including the visiting methods).

4.5.9.1 Attributes

Attributes are directly added to the logic classes as instance fields. Because every logic specification will have its own class hierarchy, accessor methods in the node are needed when other logic specifications need access to the attributes. These accessor methods will need to be part of the public interface of the logic specification.

4.5.9.2 Interface Methods

Interface methods are the only methods that are accessible from outside the logic specification. This includes all visiting methods. Every interface method is added to the interface for that node class. This interface is implemented by the tree node class and the logic node class. In the tree node class, these methods are implemented by forwarding control to the logic node class. The logic node class contains the actual method. The `run` method, shown in figure 4.4 is an interface method of `Program`. It is added to the interface `IInterpreterProgram`, the tree node class `Program` and the logic node class `InterpreterProgram`. The method in `InterpreterProgram` contains the actual functionality and this method is called from `Program`.

When an interface method is declared without a body (it is abstract), then the method is only added to the interface, and a forwarding implementation is added to the tree node class. The implementation of the abstract method needs to be provided in a subclass of the logic node class.

Construct	Destination
Package declaration	Package
Node declaration	Logic node class and interface
Attribute	Field in logic node class
Interface method	Method in logic node class and interface
Implementation method	Method in logic node class
Visiting method	Method in logic node interface

Table 4.2: Comments in the logic specification and their destination

4.5.9.3 Implementation Methods

All implementation methods are added to the logic node classes. These methods are not added to the interface, or the tree node class. This makes it impossible to access implementation methods from outside the logic specification. It is possible to specify abstract implementation methods. Such a method needs an implementation in a subclass.

4.5.9.4 Visiting Methods

Visiting methods are mostly treated as normal interface methods. The name of the logic specification is appended to the name of the method to avoid naming conflicts between multiple logic specifications. When a visiting method returns multiple values, a tuple class is generated to hold those values. This tuple class has `private` fields for all values, a single `public` constructor that takes all fields as arguments and a `get`-method for every value. Overriding the return types (see §4.5.6) will result in a subclass of the tuple defining the new values.

4.5.10 Comments and Headers

As with the tree definition the logic specification also supports comments on most constructs. These constructs are displayed in table 4.2, together with the destination of those comments. The contents of a comment will be copied in verbatim to the output file. In the logic specification it is also possible to use comments that start with '///' and stop at the end of the line. These comments are ignored.

As with the tree definition, the logic specification allows the addition of headers to a single node, or to all nodes. When an header is added to a node, this header is also added to the accompanying interface and all tree node classes implementing that interface.

Chapter 5

Design of TPLc

In this chapter an overview of the design of TPLc, the compiler for systems written in TPL is presented. It starts with an hierarchical overview of TPLc's components, followed by a discussion of these components. §5.2 describes the driver. The parser is discussed in §5.3, the context checker in §5.4 and the generator in §5.5. The use of the StringTemplate engine is explained in §5.6. Finally the design of the runtime library is discussed in §5.7.

5.1 Hierarchy

TPLc reads its input from a single tree definition and zero or more logic specifications. It generates a collection of classes in the target language. The input files are parsed and fed to the context checker. The context checker verifies the types and attributes used in the constructs in the input. Finally, the input is sent to the code generator, which instantiates the right templates.

An overview of the hierarchy of TPL is given in figure 5.1. In this diagram the following components are displayed:

Driver The `Driver` component controls the execution flow of TPL. It parses the command line options and starts all other components in the right order.

Parser The `Parser` component contains parsers for tree definitions and logic specifications. The parsers generate ASTs.

ContextChecker The `ContextChecker` verifies the correctness of both the tree definition and any logic specification. It also decorates the ASTs with addition information, such as type information, that can be used by the generator.

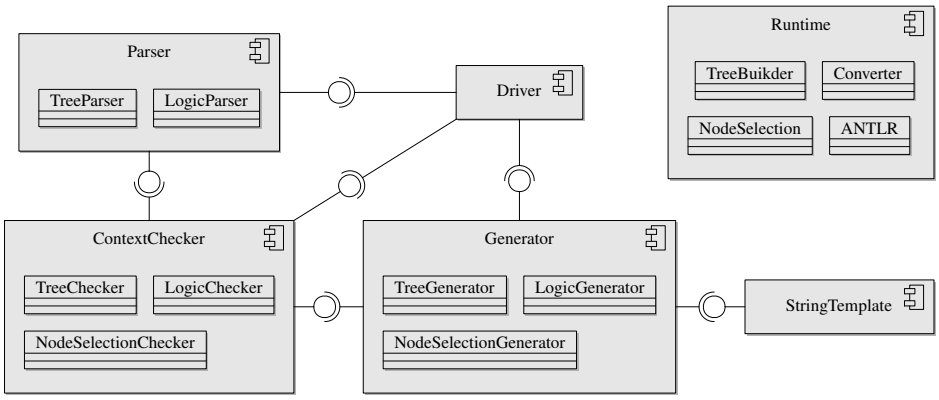

Figure 5.1: Component diagram for TPL

Generator The `Generator` component is responsible for generating the various classes. From the ASTs, produced by the `ContextChecker`, the corresponding `StringTemplates` are instantiated. As with the `ContextChecker`, this component is split into three parts: tree definition, logic specifications and node selection.

StringTemplate From [Parr, 2004]: "`StringTemplate` is a java template engine for generating source code, web pages, emails, or any other formatted text output." It will be used by the `Generator` component to build the classes.

Runtime The `Runtime` library contains classes required when a generated application is run. This consists of base classes for the AST nodes, functionality for constructing the AST, type conversion classes, runtime node selection support and an integration with ANTLR.

5.2 Driver

The driver is a simple component that is the entry point of TPL. It is started from the command line and initiates all other components. The main responsibilities of the driver are parsing the command line options and calling the other components. First, the tree definition is parsed, followed by all logic specifications. Next, the resulting ASTs are passed to the context checker. Finally, the ASTs are passed to the generator, which generates the output files.

5.3 Parser

Multiple parsers are required for TPL. The two most important are the parser for a tree definition and the parser for logic specifications. Both parsers are implemented in ANTLR v3 and have some common tokens and rules. Most of these tokens are target language dependent. In this section the common tokens and rules will be discussed, followed by a discussion of both parsers.

5.3.1 Common Tokens

Most of the common tokens and rules used by both parsers are target language dependent. These tokens are listed in §4.1. Some of these tokens match complex strings, such as method signatures or expressions. Writing complete lexers for these tokens would result in including large portions of the parser of the target language in the parser for TPL. To prevent this, the special tokens will be parsed as free form text with delimiters, which indicate the start and end of the tokens. These delimiters may depend on the context of the token. Below, all common tokens and rules will be discussed.

5.3.1.1 Identifier Token

Identifiers are matched with a single token in most programming languages. The definition for identifiers, from the Java Language Specification [Gosling et al., 2005] is given in grammar fragment 5.1. When the identifier starts with `visit`, a special type of identifier, ⟨*visit-identifier*⟩, will be matched.

5.3.1.2 Target Class Rule

The ⟨*target-class*⟩ rule matches a class name (or type) in the target language. This can be complex structures, such as parameterised types in Java or plain identifiers.

Grammar Fragment 5.1 Identifier in Java

⟨*identifier*⟩	::=	⟨*identifier-chars*⟩
⟨*identifier-chars*⟩	::=	⟨*java-letter*⟩ (⟨*java-letter-or-digit*⟩)*
⟨*java-letter*⟩	::=	Any Unicode character that is a Java letter.
⟨*java-letter-or-digit*⟩	::=	Any Unicode character that is a Java letter-or-digit.

The ⟨*target-class*⟩ rule should not match arrays. Its syntax, for the Java target, is given in grammar fragment 5.2.

Grammar Fragment 5.2 Target class in Java

⟨*target-class*⟩	::=	⟨*identifier*⟩$^{\blacktriangle 5.1}$ ⟨*type-arguments*⟩? ('.' ⟨*identifier*⟩$^{\blacktriangle 5.1}$ ⟨*type-arguments*⟩)*;
⟨*type-arguments*⟩	::=	'<' ⟨*type-argument*⟩ (',' ⟨*type-argument*⟩)* '>'
⟨*type-argument*⟩	::=	⟨*target-class*⟩
	|	'?' (('extends' | 'super') ⟨*target-class*⟩)?

5.3.1.3 Target Type Rule

The ⟨*target-type*⟩ rule matches all types in the target language, including primitives, classes and arrays. Grammar fragment 5.3 gives the syntax for the Java target.

Grammar Fragment 5.3 Target type in Java

⟨*target-type*⟩	::=	(⟨*target-primitive*⟩ | ⟨*target-class*⟩$^{\blacktriangle 5.2}$) ('[' ']')*;
⟨*target-primitive*⟩	::=	'void' | 'boolean' | 'byte' | 'char' | 'short' | 'int' | 'float' | 'double'

5.3.1.4 Package Name Rule

A package name indicates the target package, namespace or module of the generated code. This notion depends heavily on the selected target language. For Java this is a list of identifiers separated by dots, as shown by grammar fragment 5.4.

5.3.1.5 Expression Action Token

⟨*expression-action*⟩ tokens can be used in attribute declarations and in visit method invocations. In the first case they assign a default value to an attribute and in the second case they are used as values for attributes (synthesised and inherited). Expressions can be difficult and large constructs. Providing an exact lexer for expressions would result in including a large part of a parser for the target language in the lexer. Also, it is likely that other parts of the input could also be matched by that lexer, producing ambiguities. Therefore the following strategy is used:

Grammar Fragment 5.4 Package name in Java

⟨*package-name*⟩	::=	⟨*identifier*⟩$^{\blacktriangle 5.1}$ ('.' ⟨*identifier*⟩$^{\blacktriangle 5.1}$)*;

(1) The token ⟨*expression-action*⟩ can only be matched when the boolean flag `isExprFollow()` returns true.

(2) The token ends when a ';', a ',' or a ')' is read that is not part of the expression[1]. These characters are ignored when part of a string, other literal or comment.

(3) Every opening parenthesis or curly bracket should be matched by a closing parenthesis or curly bracket.

5.3.1.6 Method Signature Action Token

The token ⟨*method-sig-action*⟩ faces a similar problem as ⟨*expression-action*⟩. The token matches a complete method signature. This includes modifiers, the return type, the name and all method arguments. The following rules are used to match a ⟨*method-sig-action*⟩ token:

(1) The token can only be matched when the boolean flag `isSigFollow()` returns true.

(2) The token ends when a ';' or '{' token is read that is not part of the method signature, accounting for literals and comments.

(3) The method body is allowed to contain expressions, as discussed above. This allows the usage of annotations in Java[2].

5.3.1.7 Method Body Action Token

The ⟨*method-body-action*⟩ token matches the entire body of a method, enclosed in braces. This makes the token a little easier to match. The following rules are used to match ⟨*method-body-action*⟩:

(1) The boolean flag `isBodyFollow()` returns true.

(2) The token ends when an unmatched '}' is read. Braces inside strings, other literals and comments are ignored.

5.3.1.8 Implementation

Due to several limitations in ANTLR v3, the lexer is split in 4 different lexers. One performs the default lexing and the other 3 are responsible for the 3 actions tokens described above. The `mTokens()` method in the main lexer is modified to call one of the tree other lexers, when the corresponding conditions are met. Also, only an implementation is provided for the Java target language. Rules specific to Java are spread throughout all lexers and parsers, making it impossible to support different languages without major code duplication. ANTLR's current feature set is not rich enough to allow the construction of an extensible parser.

[1] This causes a problem when Java generics are used in an expression. The lexer mistakenly assumes that, when given the input '`new TreeMap<String, String>()`', the expression ends at the comma. A simple workaround is enclosing the expression in parentheses

[2] Although the lexer allows annotations, TPLc currently is not able to process these annotations.

5.3.2 Tree Definition

The parser for the tree definition only supports the Java target. ANTLR v3 does not (yet) support a mechanism for grammar reuse in multiple parsers. Therefore, all common tokens, required by the tree definition parser, are included in this parser.

As mentioned before, the parser generates an AST. Because TPLc is implemented in TPL, this AST is specified as a tree definition. This definition is given in listing 5.1.

Listing 5.1: Tree definition AST

```
package nl.utwente.ewi.tpl.ast.tree;
Definition(
    Comment? comment,
    PackageName packageName,
    HeaderAction? treeHeader,
    NodeDef* nodes) -> DEFINITION;
NodeDef(
    HeaderAction? nodeHeader,
    Comment? comment,
    Type type,
    ParameterList? parameters,
    AbstractToken? declAbstract,
    Type? superclass,
    Identifier* bindings) -> NODE_DEF;
ParameterList(Parameter* parameters) -> PARAMETER_LIST;
abstract Parameter;
ValueParameter(
    Comment? comment,
    Type type) extends Parameter -> VALUE_PARAMETER;
ValueAsParameter(
    Comment? comment,
    Type newType) extends Parameter -> VALUE_AS_PARAMETER;
ChildParameter(
    Comment? comment,
    Type type,
    Identifier name,
    Multiplicity? multiplicity) extends Parameter -> CHILD_PARAMETER;
ChildAsParameter(
    Comment? comment,
    Identifier name,
    Type newType,
    Identifier? newName) extends Parameter -> CHILD_AS_PARAMETER;
Multiplicity(MultiplicityType value) -> MULTIPLICITY;
PackageName(String value) -> PACKAGE_NAME;
HeaderAction(String value) -> HEADER;
Identifier(String value) -> IDENTIFIER;
Type(String value) -> TYPE;
AbstractToken() -> ABSTRACT;
Comment(String value) -> COMMENT;
```

5.3.3 Logic Specification

Due to the same limitations the tree definition parser faces, the logic specification parser also supports only the Java target. The common tokens are also included

in this parser.

The logic specification parser also generates an AST. The structure of this AST is given in listing 5.2.

5.4 Context Checker

The context checker is responsible for guaranteeing type correctness and correct usage of constructs. While doing so, it also decorates the AST with attributes needed for the code generation. Some of these attributes are also used internally by subsequent context checks. The context checker operates on the AST constructed by the parser (§5.3). The context checker is implemented in one logic specification for tree definitions and one for logic specifications.

This section describes the checks performed for both tree definitions and logic specifications. This is followed by a description of the attributes that are added to a checked tree. Finally, the order, in which the attributes are resolved and the context checks are performed, is determined.

5.4.1 Tree Definition

The main function of the context checker for the tree definition is to check the correctness of the structure of the tree. It also checks the mapping from the tree to the token types and whether the construction of the tree is unambiguous. The following checks are performed:

(1) A single type can only be declared once. It might occur that equal types have different names, for example when Java parameterised types are used. To match these types, a target language dependent comparison is performed. The Java target discards any type parameters.

(2) The superclass of a node should be a valid node type. This can either be a class defined in the tree definition, or the `Node` class. When no superclass is defined, `Node` is used. The type is resolved using the same mechanism as above.

(3) Circular inheritance relations are not allowed.

(4) A `value` parameter should be the first parameter.

(5) The types of the child nodes should be valid node types. As with the superclass, this can either be a class defined in the tree definition, or the class `Node`. For the `value` parameter any type is accepted.

(6) No two child nodes can have the same name.

(7) It is an error to redefine the structure of a node when the structure is already defined in one of the superclasses.

(8) Only existing children may be modified with the `as` keyword.

(9) When a child is assigned a new name, that name should not already have been used in the structure. TPL can not check the type of a redefinition, this will be checked by the compiler of the target language.

(10) It is an error to couple a node, that is declared `abstract` or does not have a structure defined, to token names.

Listing 5.2: Logic specification AST

```
package nl.utwente.ewi.tpl.ast.logic;
Specification(
    Comment? comment,
    PackageName packageName,
    HeaderAction? logicHeader,
    Identifier* logicNames,
    NodeSpec* nodes) -> SPECIFICATION;
NodeSpec(
    HeaderAction? nodeHeader,
    Comment? comment,
    Type type,
    AbstractToken? declAbstract,
    Type superclass,
    BodyElement* bodyElements,
    Type? binding) -> NODE_SPEC;
abstract BodyElement;
Attribute(
    Comment? comment,
    Type type,
    Identifier name,
    Expression? initialValue) extends BodyElement -> ATTRIBUTE;
abstract Method(
    Comment? comment,
    Signature signature,
    MethodBody? body) extends BodyElement;
IfaceMethod(signature as ActionSignature) extends Method -> IFACE_METHOD;
ImplMethod(signature as ActionSignature) extends Method -> IMPL_METHOD;
VisitMethod(signature as VisitSignature) extends Method -> VISIT_METHOD;
abstract Signature;
VisitSignature(
    VisitReturnType? returnType,
    Identifier name,
    TypeNamePair* inherited,
    Type* exceptions) extends Signature -> VISIT_SIGNATURE;
ActionSignature(String value) extends Signature -> METHOD_SIG_ACTION;
abstract VisitReturnType;
SingleVisitReturnType(Type type) extends VisitReturnType ->
     SINGLE_VISIT_RETURN_TYPE;
MultipleVisitReturnType(TypeNamePair+ synthesised) extends VisitReturnType
  -> MULTIPLE_VISIT_RETURN_TYPE;
TypeNamePair(
    Type type,
    Identifier name) -> TYPE_NAME_PAIR;
PackageName(String value) -> PACKAGE_NAME;
Identifier(String value) -> IDENTIFIER, VISIT_IDENTIFIER;
Type(String value) -> TYPE;
Expression(String value) -> EXPRESSION_ACTION;
MethodBody(String value) -> METHOD_BODY_ACTION;
HeaderAction(String value) -> HEADER;
AbstractToken() -> ABSTRACT;
Comment(String value) -> COMMENT;
```

(11) When a node is coupled to one or more token names, these names should be declared in the token file.

(12) Nodes and token names are coupled one-to-many.

(13) The definition of the child nodes should be unambiguous, as described in §4.4.7.

5.4.2 Logic Specification

The logic specification consists of two types of constructs: (1) constructs declaring the structure of the logic, specified in TPL and (2) the actual logic inside the action tokens. Most checks are performed on the structure. The checks performed on the action tokens are limited to the macros. The compiler of the target language will check the syntax and typing of the actions. For the structure the following checks are performed:

(1) A logic specification dependency should refer to an existing logic specification.

(2) A logic specification dependency should only be declared once.

(3) Node names in a logic specification should be unique.

(4) Logic node classes may only extend other logic node classes declared in the same logic specification.

(5) Circular inheritance relations are not allowed.

(6) When a logic node is bound to a tree node using the explicit binding syntax, the tree node should be declared in the tree definition.

(7) A logic node that is declared `abstract` can only be bound to a tree node that is also declared `abstract`.

(8) Logic node classes are bound to tree node classes one-to-one, per logic specification.

(9) When a logic node is bound to a tree node, and a subclass of that logic node is also bound to a tree node, the second tree node should be a subclass of the first.

(10) Attribute names should be unique within a node class.

(11) Visiting method names should be unique within a node class.

(12) The names of return values and method arguments of a visiting method should be unique for that method.

(13) When a visiting method is declared with the same name as a visiting method in a superclass, this new visiting method should at least define the same synthesised attributes. The types of the attributes can be different. New attributes can be added when the superclass declares more than one synthesised attribute, or a single attribute with a name.

(14) When a node is not declared `abstract` it should implement all visiting methods declared by its superclasses.

(15) Interface methods can only be overridden by other interface methods.

5.4.3 Actions

As stated before, the context checking performed on actions is limited. Only the various macros are checked. Code surrounding the macros is regarded as free-form text and no checks are performed whatsoever. The following checks are performed on the action macros:

(1) The node selection on which a visiting method is called should yield a node with the multiplicity of 'exactly one'. Iterators declared in a `#foreach` statement are considered to have a multiplicity of 'exactly one'.

(2) A visiting method, called on a node, should be available on that node.

(3) When a visiting method is called, all its inherited attributes should be provided. It is an error to provide too many attributes.

(4) It is an error to store the results of a visiting method, using the multi-value store syntax, when that method that does not return multiple values.

(5) It is only possible to instantiate a multi-value return tuple in a visiting method with multiple return values.

(6) When instantiating a multi-value return tuple, all synthesised attributes should be provided. It is an error to provide too many attributes.

(7) Only declared attributes of a return value of a visiting method can be accessed.

5.4.4 Node Selection

The node selection context checker checks not only the correctness of a node selection, but also determines the type and multiplicity of the result. It uses a recursive algorithm over the AST of the node selection. The context checker is implemented in the logic specification `contextCheck`, that is part of the `nl.utwente.ewi.tpl.nodeselection` package.

A node selection is considered erroneous when it can never return at least a single node. This can happen when the parent of the root node is selected, when a non-existing child is requested, or when incorrect combinations of filters and nodes are given. When the value of a node is selected, it should always yield exactly one result. It is also an error to continue a selection after selecting `value`.

The type and multiplicity of a node selection are used for the context checks and code generation of visiting method calls and for-each statements. When a visiting method is called, the context checker reports an error when the node selection does not yield exactly one node.

5.4.5 Added Attributes

Table 5.1 shows the attributes that are added to the tree definition AST nodes. The attributes added to the logic specification AST nodes are shown in Table 5.2. The attributes listed in these tables are all added during the first pass over the AST. For the tree definition, this pass is defined in the `annotateTree` logic specification and for the the logic specification it is defined in `annotateLogic`. In addition to setting these attributes, these logic specifications also add some useful methods to the AST and performs context checks related to the attributes set. Some more attributes are added during the generation pass. These additional attributes are mainly needed to work around limitations in `StringTemplate` and will not be discussed.

AST Node	Attribute	Description
ChildAsParameter	original	A link to the ChildParameter that is overridden by the new definition.
	typeNode	A link to the NodeDef that defines the type of the parameter.
ChildParameter	typeNode	A link to the NodeDef that defines the type of the parameter.
Definition	allNodes	A map with all nodes defined in the tree definition.
	rootNode	The artificial NodeDef for the node Node.
	tokens	A map from a token's name to its number created from the tokens file.
NodeDef	logicNodes	All logic specification nodes that are bound to the node.
	matched-Tokens	The token names that are matched by the node. This includes all subclasses.
	parentTypes	A list of NodeDefs that are explicitly declared as possible parents.
	subclasses	All subclasses of the node.
	superclass	A link to the NodeDef that is the superclass of the node.
Parameter	index	The index of the parameter in the node's parameter list.

Table 5.1: Attributes added to tree definition nodes

AST Node	Attribute	Description
MethodBody	usedNodes	A list of NodeDefs used as iterators in #foreach statements.
MultipleVisit-ReturnType	superReturn-Type	A link to the return type that is modified by this type.
	name	The name of the tuple that contains the returns values.
NodeSpec	interface-Needed	A flag indicating if an interface is needed for the node.
	treeBinding	The NodeDef the node is bound to.
	subclasses	All subclasses of the node.
	superclass	A link to the NodeSpec that is the superclass of the node.
Specification	dependencies	A list of specifications the specification depends on.
	logicName	The name of the logic specification (derived from the file name).

Table 5.2: Attributes added to logic specification nodes

Several of the attributes set during the annotation pass can not be resolved until other attributes are evaluated. Sometimes a context check needs to be performed before the attribute can be set or vice versa. Tables 5.3 and 5.4 show the dependencies between the attributes and context checks for tree definitions and logic specifications. The order of the items in the tables represents the order in which the checks are performed and the attributes are set. The dependencies displayed in these tables represents the dependencies as they are seen in the code. To determine the order of the evaluation of the checks and attributes, a transitive closure of the tables is needed.

The tables 5.3 and 5.4, however, lack information about the traversal of the tree. Some attributes can only be set when another attribute is set on all nodes of that type in the entire AST. The same holds for some context checks. For example, the `parentTypes` attribute relies on the `superclass` chain between nodes: if node `A` can have node `P` as parent, then node `S`, superclass of `A`, can also have `P` as parent. If the `superclass` attribute is not set on all nodes, this algorithm will give incorrect results. This issue is solved by first visiting all `NodeDefs`, setting the `superclass` and `subclasses` attributes, and then revisiting the nodes and their `Parameters`, to collect parent type information.

Figures 5.2 and 5.3 illustrate the order in which the nodes for respectively the tree definition and logic specification are visited. The nodes are visited in depth first order, from the left to the right, as they are displayed in the figures. For `Parameters` and `VisitReturnTypes`, the node type that is visited is determined at runtime. This is indicated with a line with a bullet.

5.5 Generator

The generator component is implemented in two logic specifications. `generateTree` generates the output files resulting from the tree definition and is the starting point of the generation. The `generateLogic` specification generates the output files resulting from the logic specifications. Both use `StringTemplates` (see §5.6) to construct the resulting code.

The first logic specification generates a class hierarchy, as displayed in figure 4.1, using the guidelines given in §4.4.5. These classes are the AST node classes. Functionality, specified in the logic specifications, is then added, as displayed in figure 4.4, using the rules given in §4.5.9, by the second logic specification.

5.5.1 Generation Targets

Special care is taken to keep the generator completely independent of the target language. As much information as possible about the target language is stored in the templates. However, not everything can be generated without special support functionality. For example, the code to write a Java file may be different than the code required to write a C++ file, as the first needs to be placed in the right directory, according to the package specification, whereas the second has no such requirement.

This target language dependent functionality should not be mixed with the target language independent code. Therefore, a special interface, `Target`, provides access to this functionality in an implementation independent manner. Every generation target will have its own implementation of the `Target` interface. Examples of pieces of functionality, that are accessible through the `Target` interface, are the name of the `StringTemplateGroup`, file creation and method signature parsing.

Depends on	tokens	rootNode	(5.4.1.1)	allNodes	(5.4.1.2)	superclass	(5.4.1.3)	subclasses	index	(5.4.1.4)	(5.4.1.5)	(5.4.1.6)	(5.4.1.7)	typeNode	(5.4.1.8)	(5.4.1.9)	typeNode (for as)	original	parentTypes	matchedTokens	(5.4.1.10)	(5.4.1.11)	(5.4.1.12)
tokens																							
rootNode																							
(5.4.1.1)		•																					
allNodes		•	•																				
(5.4.1.2)			•																				
superclass			•		•																		
(5.4.1.3)					•	•																	
subclasses						•																	
index																							
(5.4.1.4)								•															
(5.4.1.5)			•																				
(5.4.1.6)																							
(5.4.1.7)						•																	
typeNode								•			•	•	•										
(5.4.1.8)						•																	
(5.4.1.9)						•								•									
typeNode (for as)								•			•			•	•								
original						•					•												
parentTypes													•			•							
matchedTokens								•															
(5.4.1.10)						•																	
(5.4.1.11)	•																						
(5.4.1.12)																							
(5.4.1.13)								•													•	•	•

Table 5.3: Dependencies between tree attributes and context checks

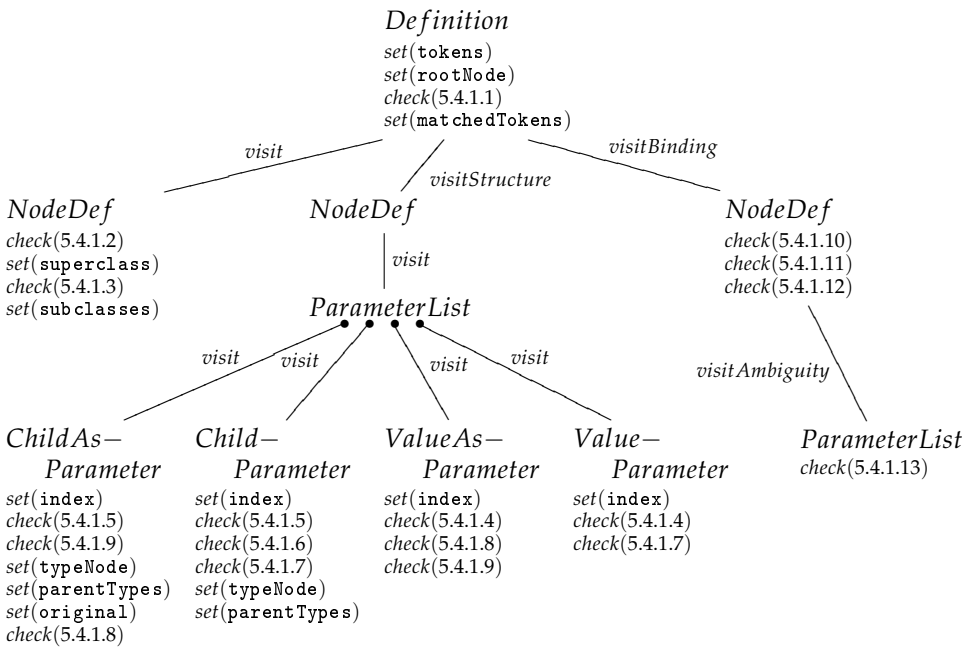

Figure 5.2: Tree traversal of the tree definition

Depends on	logicName	(5.4.2.1)	(5.4.2.2)	dependencies	(5.4.2.3)	(5.4.2.4)	superclass	(5.4.2.5)	subclasses	interfaceNeeded	(5.4.2.6)	treeBinding	(5.4.2.7)	logicNodes	(5.4.2.8)	(5.4.2.9)	(5.4.2.10)	(5.4.2.11)	name	(5.4.2.12)	superReturnType	(5.4.2.13)	usedNodes	(*action*)	(5.4.2.14)
`logicName`																									
(5.4.2.1)	•																								
(5.4.2.2)	•	•																							
`dependencies`		•	•																						
(5.4.2.3)																									
(5.4.2.4)					•																				
`superclass`						•																			
(5.4.2.5)						•	•																		
`subclasses`							•																		
`interfaceNeeded`							•																		
(5.4.2.6)																									
`treeBinding`											•														
(5.4.2.7)											•														
`logicNodes`											•	•													
(5.4.2.8)											•		•												
(5.4.2.9)							•		•		•														
(5.4.2.10)																									
(5.4.2.11)																									
`name`	•																	•							
(5.4.2.12)																									
`superReturnType`																		•		•					
(5.4.2.13)							•														•				
`usedNodes`											•														
(*action*)											•			•											
(5.4.2.14)							•											•							
(5.4.2.15)							•																		

Table 5.4: Dependencies between logic attributes and context checks

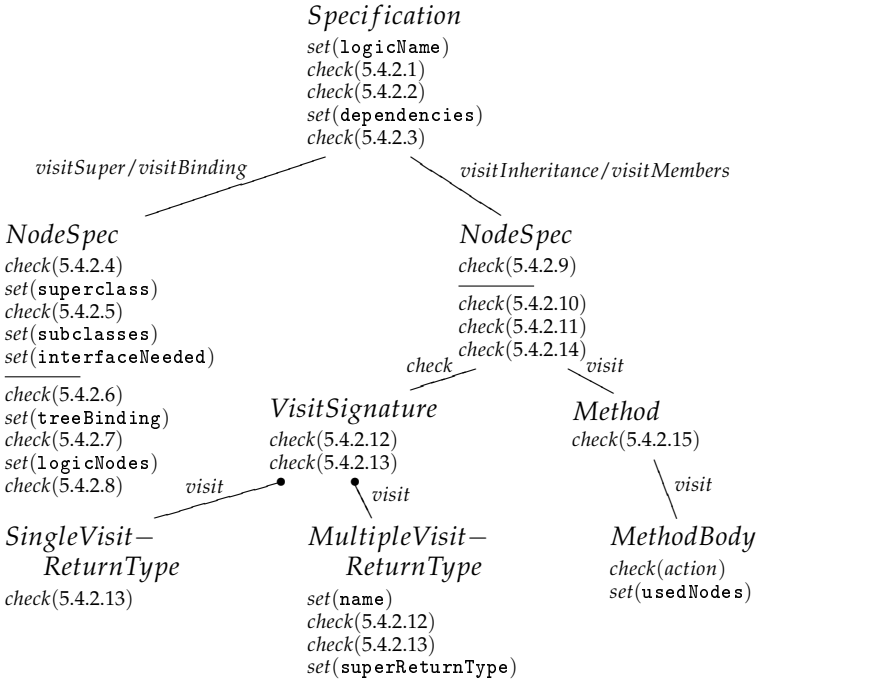

Figure 5.3: Tree traversal of the logic specification

5.6 String Template

The `StringTemplate`[Parr, 2004] library is a template engine that is used to create a mapping between the constructs in TPL and the generated output. `StringTemplate` is also used as the generation engine behind ANTLR v3.

TPL defines a template interface `TPL`, which contains all available templates. To support a new target language a `StringTemplateGroup` has to be created that implements at least these templates. The attributes of the templates are chosen in such a way that they make no assumptions on the syntax and semantics of the target language. If an attribute does contain fragments of functionality or constructs from the target language, the contents of the attribute needs to be generated by another template.

Unfortunately, `StringTemplate` is not strong enough to solve all model-view related problems. For example, it is impossible to convert the first character of a string to upper case. Also, it is impossible to strip type parameters from a type declaration (for example `Literal<T>` to `Literal`). A separate pass over the AST annotates the AST with attributes that only have a syntactic value, such as the `rawType` attribute, that is added to `NodeDef`, `Parameter` and `NodeSpec`.

5.7 Runtime Library

The runtime library is needed by applications written in TPL. It contains support classes, such as the base classes for the AST nodes. All classes of the runtime library are part of the `nl.utwente.ewi.tpl.runtime` package.

5.7.1 Node Base Classes

There are two base classes for nodes: `AbstractTPLNode`, the base class for AST nodes and the base class for logic nodes, called `AbstractLogic`. This last class only contains a reference to an AST node, as can be seen in figure 4.4.

`AbstractTPLNode` contains the type of the node. Optionally, a payload can be added to the node. This is especially useful to store the token that resulted in the node, when the source of the tree is a parser run. It also stores a reference to the parent of the node.

5.7.2 Tree Construction Classes

Trees can be constructed using the generated node constructors. However, sometimes, for example when ANTLR is used to instantiate the AST, it is required to build trees through a 'homogeneous' interface, with methods, such as `addChild`, `createNode` and `getChildCount`. The class `TPLTreeBuilder` provides this interface. It uses the annotations added to the AST nodes (§4.4.7) and reflection to construct the nodes. These annotations, `Structure` and `StructureParameter`, are, together with an enumeration that describes the various multiplicities, also part of the runtime library. It is also possible to inspect these annotations at runtime, using `TPLTreeInspector`.

5.7.3 Type Conversion Classes

TPL supports arbitrary types to be used for the value of a node. During the construction of the AST, tokens are automatically converted to the right type, as indicated by the `Structure` annotation. The classes responsible for the conversion, make up the `nl.utwente.ewi.tpl.runtime.converter` package. The package

contains implementations of the `Converter` interface for all primitive wrapper classes (such as `Integer` and `Boolean`). All converters are accessed through the `ConversionSelector`, which selects the appropriate `Converter`, according to the given type. It is also possible to add custom implementations of the `Converter` interface to the `ConversionSelector`.

5.7.4 Runtime Node Selection

Node selections can be part of the input, in which case they are compiled into code in the target language, which performs the selection. However, it is also possible to perform node selections at runtime. The package `nl.utwente.ewi.tpl.runtime.nodeselection` includes a node selection parser and the logic specification `selection`, which can be used to parse a node selection and apply it to an AST.

5.7.5 ANTLR Integration

It is possible to construct ASTs, generated with TPLc, directly from an ANTLR parser. This is accomplished through a custom implementation of the `TreeAdaptor` interface. This implementation is provided by the `TPLTreeAdaptor` class in the `nl.utwente.ewi.tpl.runtime.antlr` package. This class uses the tree construction classes provided by TPL to build the AST (see §5.7.2).

Chapter 6

Conclusions and Future Work

This chapter looks back at the work presented in this thesis. First, the conclusions and contributions are given in §6.1. This is followed by a discussion of future research in §6.3.

6.1 Summary

Abstract syntax trees, or trees in general, are common data structures in software applications. Many techniques for building ASTs and working with them exist. The most important are outlined in chapter 2. However, many of these techniques are limited in their applicability, require major effort to implement, or introduce maintenance problems in an evolving application.

This thesis introduces TPL and its accompanying compiler TPLc in chapter 3, which solve many of the problems a developer would face, when designing and developing a tree structure. The most important concepts for TPL are type safety and automatic generation of code, from a specification which provides a clear separation of concerns. TPL allows the definition of a strictly typed tree, a *tree definition*, to which behaviour can be added in a modular fashion. Different facets of this behaviour can be developed in separate modules, called *logic specifications*.

TPLc compiles behaviour specifications into the classes composing the tree structure. The inheritance pattern is combined with multiple inheritance to provide a flexible connection between the behaviour and the tree nodes. It allows the specification of different inheritance trees for the tree definition and the logic specifications. To be able to provide the same functionality for languages that do not support multiple inheritance, the inheritance pattern with composition is developed. This pattern emulates multiple inheritance in single inheritance languages, such as Java.

A common applications of tree structures are abstract syntax trees in compilers. Therefore, TPL allows a binding between parser rules and tree node classes. An implementation for the ANTLR parser generator is provided.

A query language is also integrated in TPL. Using these queries, it is possible to search the tree for certain nodes or sub-structures. A construct is provided to iterate over the results of a query.

6.2 Evaluation

The compiler for TPL, TPLc, is implemented in TPL. This implementation demonstrates the advantages of using TPL over a general purpose object-oriented language in compiler construction. The implementation written in TPL is bootstrapped with the prototype compiler, written in Java. Even though this prototype compiler does not support all features, its code is already convoluted. A single file contains functionality for tree construction and modification, string serialisation,

context checking and code generation. Furthermore, a single piece of behaviour, such as the context checker, is split over multiple files. With TPL, a single file is used for every piece of behaviour. Also, the code for constructing and modifying the tree is automatically generated. This results in a compiler that is easier to maintain and that has less lines of code, even though it supports more features.

In the comparison between TPL and attribute grammars, it becomes apparent that TPL's feature set supports the evaluation of L-attributed grammars. However, when more complex attribute grammars need to be evaluated, the evaluation order of the attributes needs to be determined by hand.

The visiting methods, supported by TPL, allow the return of multiple values. For this feature, a special syntax is introduced, with which tuples can be instantiated, stored and used. Although useful, these tuples are cumbersome to work with. For example, it is not possible to declare tuple variable without actually storing a value in it. It is questionable if support for tuples falls within the scope of TPL or if the target language should deal with these.

A disadvantage of TPLc is the large number of classes it generates. For example, for the abstract syntax tree for logic specifications has 23 tree node classes, but the generated class structure consists of 113 classes. This is the result of the inheritance pattern with composition. The large number of classes makes it difficult to debug the application and imposes a runtime overhead. This pattern also requires the addition of accessor methods for attributes when functionality from one logic specification requires access to an attribute defined in another logic specification, even when the attribute is added to the same tree node class as the functionality.

6.3 Future Research

During the development of TPLc several issues surfaced, which could not be addressed in the current version. These issues, and possible solutions, are discussed below.

6.3.1 Multiple Bindings for Logic

Currently, TPL allows a node implementing functionality to be bound to at most a single tree node. This makes it impossible to specify the structure displayed in the figures 3.4 and 3.5. In these figures, the functionality classes `PrintLiteral` and `PrintIdentifier` are both bound to two tree nodes. To implement cases like these, additional classes need to be introduced; one class for each tree node that needs the functionality.

It would be better if TPL would allow a single functionality node to be bound to multiple tree nodes. However, this does require a change in the node selection context checking algorithm. The current algorithm depends on a single starting node type. This needs to be extended to multiple node types.

6.3.2 Inclusion of Pattern Matching

In §2.2.1.1 Kimwitu [Van Eijk et al., 1997] and Memphis [Memphis] were introduced. These tools introduce constructs, with which it is possible to match tree structures against patterns. These patterns can be used to construct algorithms over the tree.

It was stated in §3.1.2.2 that support for these patterns would be recommended. Still, TPL does not include such pattern matching constructs. It would be beneficial if the macros in the method bodies were to be extended with pattern matching constructs.

6.3.3 Merge Inheritance Trees

Figure 3.7 illustrates that TPLc generates a complex class structure, even for a small specification. It requires 17 classes and 2 interfaces for a specification that could also have been compiled to 6 classes, as is displayed in figure 3.2. Cases like these, where the behaviour could have been included in the tree node classes, are common. However, there are also cases, like the example illustrated by figure 3.4, where this is not possible.

The large number of classes imposes a performance penalty for some platforms, such as the Java Virtual Machine, which needs to load and verify the classes. Also, the additional indirection, when the inheritance pattern with composition is used, adds an overhead to every method class and the instances of the functionality classes greatly increase the memory consumption of the tree. Therefore, TPLc should merge inheritance trees of logic specifications and the tree definition whenever this is possible. This can greatly reduce the number of classes generated and the number of composition classes required, increasing computational performance and reducing memory consumption.

6.3.4 Attribute Grammars

Attribute grammars can be useful when a formal specification of a language is needed. TPL can be used to evaluate the class of L-attributed grammars, using visiting methods with multiple return values. These grammars can be evaluated in a single top-down pass over the tree. However, other attribute grammars might require multiple passes over the tree. TPL can be used to evaluate these grammars, but the order, in which the attributes are evaluated, needs to be specified manually.

To be able to use TPL with any attribute grammar, a preprocessor needs to be developed, which performs a dependency analysis of the attributes. This preprocessor will take an attribute grammar specification and compile it into tree definition and logic specification, which can be processed by TPLc.

6.3.5 Addition of Children in a Subclass

With the `as` keyword, it is possible to change the name and type of a child of a node in the tree definition. TPL allows only these modifications. A possible improvement would be to allow the addition of new children in a subclass. This can be extended to replace the notion of 'defining the structure a node'. A node definition would start with no children defined and children are added in the subclasses.

6.3.6 Accessing Attributes in other Logic Specifications

The generation of separate inheritance trees for different logic specifications has a counter productive effect on accessing attributes in logic specifications. An attribute defined in one logic specification cannot be accessed directly from another specification. The addition of accessor methods is required, and these methods need to be defined with the `interface` keyword. However, this exposes the attributes through the interface of the tree nodes.

It would be preferable if attributes could be accessed directly from other logic specifications. A possible solution would be to automatically generate the required accessor methods. However, this still exposes the attributes through the interface of the tree nodes. A solution, which might require multiple inheritance,

would be the generation of an 'attribute class' per tree node class, which is extended by the functionality classes for that node. To prevent the usage of multiple inheritance, this 'attribute class' could also be shared through aggregation.

6.3.7 Graphs

TPL is used to describe tree structures. Trees are a particular class of graphs, in which any two vertices are connected by exactly one path. TPL could be extended to support graphs that are not trees. It is questionable whether TPL should be extended to support arbitrary graphs, but the generated code could be made more flexible in this regard. If a node could have more than one parent, it would be possible to share a subtree in different parts of tree.

6.3.8 Visiting Pattern

TPL always generates `visit` methods and a `Visitor` interface for use with the visitor pattern. It would be better if the generation of these methods was optional. A switch could be added which makes it possible to chose between no visitor support, normal visitor support and support for visitor combinators.

Appendix A

Software Requirements Specification

TPL is a language in which tree structures can be described. Specifications in TPL are compiled with TPLc. This software requirements specification (SRS) establishes the requirements and structure of TPL, and its compiler TPLc.

A.1 Environment

TPLc will be a standalone application, implemented in Java. It is executed by the Java Virtual Machine, making it available on all platforms for which a Java Virtual Machine is available.

A.2 Structural Organisation

TPLc will be invoked through a command line interface. A specification in TPL consists of two parts: the tree definition and zero or more logic specifications. The first describes the structure of a tree, and the second contains the behaviour of this tree.

TPLc takes these files, and generates a class structure. The tree definition is used to construct a class inheritance tree, representing the tree nodes (see §A.4.1). The behaviour specified in the logic specifications is added to these tree nodes (see §A.4.2). All generated output files will be of a specified language and are put in the correct packages or modules, depending on what the target language uses. TPL will only support object oriented languages as generation target.

A.3 User Interaction

TPLc is a command line application, with no real interaction. The application is started by the user, and it either generates the output files, or aborts with an error. This section describes the input and output of TPLc.

A.3.1 Input

- Options to TPLc will be specified through command line options.
- At least a single file, the tree definition, needs to be given at the command line. All other files should be logic specifications. The tree definition should be the first file.

A.3.2 Syntax

Although the syntax is an important part of any language, it only serves as an interface to the concepts of that particular language. The syntax for both the tree

definition and logic specification of TPL will be developed after the complete feature set has been designed. The syntax shall facilitate all features of the language and shall be easy to understand for someone with Java experience.

A.3.3 Messages

TPLc can give messages at various points in its execution. The events that trigger these messages and the contents of these messages are described below.

A.3.3.1 Notification Messages

When the `-report` command line option is specified, TPLc will give status information to the user. Information is given about the action currently performed and the file(s) being processed.

A.3.3.2 Error Messages

Error messages are always displayed when processing can not continue due to some abnormal situation. These include syntax errors, IO errors, context errors, etc.

TPL is a research project and not intended to be used in a production environment. Therefore, the error handling does not need to be very robust. TPLc will bail out after the first error and will not try to recover.

A.3.4 Output

The output generated by TPLc consists of several files in the chosen generation language. These files are (1) the generated AST node classes, (2) the classes with the functionality of the tree, (3) a `Visitor` interface and (4) metadata . These files will be placed in the directories specified in the input files.

A.4 Language Features

This section describes the features of the language TPL itself. First the requirements of the tree definition are given, followed by the requirements of the logic specification.

A.4.1 Tree Definition

The tree definition allows the developer to define a type-safe AST structure. It shall conform to the following requirements:

- It shall be possible to define a package or module for the AST node types.
- The tree definition shall allow the definition of names for every node type.
- It shall be possible to specify a many-to-one relation between token types and node types.
- It shall be possible to define the structure of a node type; specifying the children of the node type. The types of these children shall be other node types.
- It shall be possible to define names for children.

- It shall be possible to specify the multiplicity (optional, exactly one, zero or more and one or more) of every child.
- A node type shall also be able to hold a value, which is a special non-tree child.
- The type of the node value shall be specifiable as any class for which a converter from strings is available.
- Every operation on every node shall always be type-safe.
- Type-safe constructors will be generated for every node type. The arguments of these constructors are the child nodes and, when specified, the value.
- For every child node, a type-safe `get-` and `set`-method shall be generated. Or, in the case of a multiplicity of greater than one, a `get-`, `add-` and `remove-`method are generated.
- It shall be possible to define inheritance between node types.
- In a single path in the inheritance tree from the root class to a node type associated with one or more token types, there shall always be exactly one definition of the structure of the node type. Superclasses of the node type defining the structure will not have a structure defined; subclasses will have the structure defined.
- It shall be possible to narrow the type or change the name of children in subclasses of a node type defining the structure.

A.4.2 Logic Specification

The logic specification allows the developer to add attributes and functionality to the AST node classes. It shall conform to the following requirements:

- It shall be possible to define a package or module for the classes and interfaces generated for the logic specification.
- It shall be possible to specify required logic specifications.
- It shall be possible to specify attributes with an optional initial value which are added to AST node types.
- It shall be possible to specify interface methods—methods that are available on the AST node types—which are added to the AST node types.
- It shall be possible to specify implementation methods—methods that are not available on the AST node types—which are added to the AST node types.
- It shall be possible to specify visiting methods—interface methods which can return multiple values—which are added to the AST node types.
- Both the interface and implementation methods shall have an optional body.
- Code in the various method bodies will accept macros for accessing attributes, child nodes and visiting methods.
- It shall be possible to bind a logic type to an AST node type.
- Node types can extend other node types, inheriting attributes and interface, implementation and visiting methods.

A.5 Nonfunctional Requirements

TPL will be developed in separate stages. These stages are:

- Requirements analysis and design.
- Prototype implementation.
- Design refinements and syntax fixation.
- Final product implementation.

There will be three deliverables: (1) This thesis, (2) the product and (3) API documentation.

Appendix B
Testing

The set of unit tests is used for regression testing. All unit tests are implemented with JUnit [JUnit, 2006]. Different testing suites are developed for the four main components: the runtime library, the parsers, the context checkers and the generator. These test cases are discussed below.

In addition to these unit tests, TPLc itself also is a test case. TPLc is implemented in itself, using many of the features present in the language. TPLc should be able to compile itself.

B.1 Runtime Library

The most complex parts of the runtime library are the `TreeAdaptor` and the `ConversionSelector`. Only these classes are tested, and with them the various `Converter` implementations.

`TPLTreeAdaptor` is tested for compliance with the (informal) specification if its API in `TreeAdaptor` on a per-method basis. All cases mentioned in the API are tested one by one.

For the `ConversionSelector` a set of 4 test cases per primitive type is used:

(1) A successful conversion from a string to the selected type.

(2) A malformed string, which results in a `TypeConversionException`.

(3) An overflow of the primitive (when applicable), which should also result in a `TypeConversionException`.

(4) A successful conversion from a primitive value back to a string.

B.2 Parser

The parser unit tests are responsible for testing the relation between the input source and the generated AST. Every test case contains both the input and the output as plain text. The test fails when the generated output does not match the expected output.

Extensively testing a parser against different input fragments is questionable. The grammar is specified in a semi-formal notation, that is close to EBNF. Any sentence that is accepted by the EBNF specification should also be accepted by the generated parser. If this is not the case, it is a bug in the parser generator, not in the parser. The test cases are chosen in such a way that they are likely to hit problems in the grammar specification, and not in the parser generator.

B.3 Context Checker

The context checker test cases can be divided in 4 groups: (1) Helper classes, (2) tree definition checks, (3) logic specification checks and (4) macro checks. These groups use a different test case setup.

For the first group one or more test case classes are written per helper class. These test cases test conformance to the API using black box testing.

The test cases for the other three groups are written in pairs: a case that should trigger a context error and a similar case in which the error should not occur. These test cases all use multiple input files per test case, which are named according to the following naming scheme:

Tree Definition Every test case consists of two input files, the first is named `case<n>Incorrect.tree` and contains the erroneous definition, the second is named `case<n>Correct.tree` and has no errors.

Logic Specification Every test case consists of three input files. One file contains the tree definition for both the correct and incorrect logic specification. It is called `case<n>Logic.tree`. The correct and incorrect logic specifications are called `case<n>Correct.logic` and `case<n>Incorrect.logic`.

Macros The files for the action macros are similar to those of the logic specification. The tree definition is called `case<n>Action.tree`. The correct and incorrect cases are called `case<n>CorrectAction.logic` and `case<n>IncorrectAction.logic`.

B.4 Generator

Extensive testing of the code generator is difficult to achieve, because the output should not only compile, but also be conform the specifications in §4.4.5 and §4.5.9.

These sections however do not specify the exact layout of the code, just its high level structure. This makes it impossible to perform a tests based on pre-written sample outputs using a simple text comparison.

A solution would be to inspect the code after compiling it to a Java class file. However this requires the execution of the Java compiler from within the test case and custom class loading. Also it is still impossible to verify the correctness of the actual code.

Because of these problems it was chosen to see the implementation of TPL in itself as the test case of the generator. The generator is assumed to work correctly if the bootstrapped version of TPL passes all unit tests.

When a problem with the generator is found, a test case that reveals the problem is written as part of the example calculation language. This example is processed before the test cases are executed. The generated code can be inspected through reflection.

Appendix C

Improvements in the Final Version

TPL's development consists of a prototype phase followed by the development of the final product. This final version is not only an architectural improvement, but it also addresses functional problems. This chapter discusses functional improvements made in the final version.

C.1 Structural Changes

Several changes are carried trough with regard to the structure of a node. These change and their rational are explained in this section.

C.1.1 Parent Type

In the prototype the parents of a node were of the type `AbstractTPLNode`. When a parent was accessed it needed to be cast to the required type. In the final version the parents have a type that is as specific as possible. The union of all contexts in which a node can be used is taken and the highest common denominator is taken as the type of the parent.

C.1.2 Setting of the Parent

In the prototype, the parent of a node was set with a separate call to the `setParent` method. The `TPLTreeAdaptor` first added the child to its new parent and then called this method. When using the `TPLTreeAdaptor` this poses no problems, however when constructing a tree by hand, it is easy to forget to call the `setParent` method; resulting in an invalid tree. To guarantee the structural integrity of the AST the call to `setParent` is moved to the accessor methods on the nodes. When a child is added to a node, this `setParent` method is called by the `add-` or `set`-method automatically.

C.1.3 Accessor Methods

In the prototype a pair of `get-` and `set`-methods was generated for every child. When the child had a multiplicity of greater than one, these accessor methods operated on the list of children. This made it possible (and even required) to modify the list outside the node. To give the node more control over these lists, the accessor methods for the multiplicities 'zero-or-more' (`*`) and 'one-or-more' (`+`) have been modified. The `set`-method is removed, making it impossible to overwrite the list. To prevent modification of the list outside the node, the `get`-method returns an unmodifiable wrapper of the list. To add or remove nodes a pair of `add-` and `remove`-methods is added.

Construct	Destination
Tree Definition	
Package declaration	Package
Node definition	Tree node class
Node parameter	Accessor methods and field
Logic Specification	
Package declaration	Package
Node declaration	Logic node class and interface
Attribute	Field in logic node class
Interface method	Method in logic node interface
Implementation method	Method in logic node class
Visiting method	Method in logic node interface

Table C.1: Comments and their destination

C.1.4 Comments and Headers

In the prototype it was not possible to add comments to any part of the node definition or the logic specification. This is addressed in the final version, where it is now possible to add comments to most constructs. The comments will be copied in verbatim to the corresponding constructs in the generated code. For example, comments on a node definition will end up in the tree node class. Table C.1 shows the constructs supporting comments and the destination of those comments.

The `header` section of both the tree definition and logic specification have been improved to allow the addition of headers to a single node.

C.2 For-Each Statement

In the final version, the for-each statement is improved to use a ⟨*node-selection*⟩ to select the nodes to iterate over. The for-each statement can now loop over any node selection. This is especially powerful with the combination of a more advanced node selection (see §C.3).

C.3 Node Selection

The prototype compiler uses a very limited node selection. It is only possible to select a direct child, the value, the parent or the current node. The final version provides an improved node selection, in which it is possible to select children of children and filter nodes on a given type. This concept is discussed in §4.5.8.

Appendix D

Visiting Methods and Attribute Grammars

The visiting methods used by TPL are related to the concept of attribute grammars. An attribute grammar is an extension of a context free grammar in which it is possible to specify the flow of information. This flow is specified by means of synthesised and inherited attributes. A synthesised attribute is used to pass information from a node to its parent and an inherited attribute is used to pass information from a node to its children.

TPL's traversal functionality also allows passing attributes up and down the abstract syntax tree. However, the semantics of an attribute grammar and TPL's traversal functionality are not entirely the same. This is best illustrated with an example. The example was taken from [Alblas and Nymeyer, 1996] and was slightly modified.

Consider a language with the syntax defined in grammar fragment D.1, in which variables can be declared and used. In this language variables need to be declared in the same block as they are being used or in one of the parent blocks. For this grammar, a context checker is specified as attribute grammar. The evaluation rules for this attribute grammar are also displayed in grammar fragment D.1.

The Attributes used in this grammar are specified in table D.1. The *original* and *updated* attributes are used to gather all variable declarations in a block. *original* is passed into a rule, all declarations are gathered and returned with the *updated* attribute. Per block these results are put into the attribute *available*, which contains all valid variables.

In the specification, several utility methods are used. `new_table()` creates an empty symbol table. `append` creates a new table that contains the new name and all declarations of the old table. An error is reported when the new declaration is already present. `concatenate` creates a new symbol table that contains all entries of both tables. Duplicate entries from the first argument will override entries from the second. The final operation `check_declaration` reports an error when the entry is not in the symbol table. The Java implementation of the symbol table is given in listing D.2.

To specify the same context checker in TPL, a tree definition and a single logic specification are needed. The tree definition is given in listing D.1. It defines an AST, which matches the grammar given in fragment D.1.

Two differences between the attribute grammar and tree definition can already

Name	Type	Attribute
original	Symbol Table	**inh of** *statement_part*, *statement*
updated	Symbol Table	**syn of** *statement_part*, *statement*
available	Symbol Table	**inh of** *block*, *statement_part*, *statement*
name	String	**syn of** identifier_token

Table D.1: Attributes used in the context checker

Grammar Fragment D.1 Simple declaration and use attribute grammar

⟨*program*⟩	::=	⟨*block*⟩
⟨*block*⟩	::=	'begin' ⟨*statement-part*⟩ 'end'
⟨*statement-part*⟩	::=	⟨*statement-part*⟩ ⟨*statement*⟩
	|	⟨*statement*⟩
⟨*statement*⟩	::=	'var' ⟨*identifier*⟩
	|	'use' ⟨*identifier*⟩
	|	⟨*block*⟩

program : *block*.
[*available* **of** *block* = new_table();]
block : begin_token *statement_part* end_token.
[*original* **of** *statement_part* = new_table();
available **of** *statement_part* = concatenate(
updated **of** *statement_part*, *available* **of** *block*);]
$statement\_part_0$: $statement\_part_1$ *statement*.
[$original$ **of** $statement\_part_1$ = $original$ **of** $statement\_part_0$;
$original$ **of** *statement* = $updated$ **of** $statement\_part_1$;
$updated$ **of** $statement\_part_0$ = $updated$ **of** *statement*;
$available$ **of** $statement\_part_1$ = $available$ **of** $statement\_part_0$;
$available$ **of** *statement* = $available$ **of** $statement\_part_0$;]
statement_part : *statement*.
[$original$ **of** *statement* = $original$ **of** *statement_part*;
$updated$ **of** *statement_part* = $updated$ **of** *statement*;
$available$ **of** *statement* = $available$ **of** *statement_part*;]
statement : var_token identifier_token.
[$updated$ **of** *statement* =
append(*name* **of** identifier_token, *original* **of** *statement*);]
statement : use_token identifier_token.
[$updated$ **of** *statement* = $original$ **of** *statement*;
check_declaration(*name* **of** identifier_token,
available **of** *statement*);]
statement : *block*.
[$updated$ **of** *statement* = $original$ **of** *statement*;
available **of** *block* = $available$ **of** *statement*;]

Listing D.1: Declare and use tree definition

```
package decluse;

Program(Block block);
Block(StatementPart statementPart);
StatementPart(Statement+ statements);

abstract Statement;
VarStatement(Identifier name) extends Statement;
UseStatement(Identifier name) extends Statement;
BlockStatement(Block block) extends Statement;
Identifier(String value);
```

Listing D.2: Declare and use symbol table

```
package decluse;

import java.util.HashSet;
import java.util.Set;

public class SymbolTable {
  private Set<String> declared;

  private SymbolTable() {
    declared = new HashSet<String>();
  }

  public static SymbolTable newTable() {
    return new SymbolTable();
  }

  public static SymbolTable concatenate(
      SymbolTable newDefs, SymbolTable oldDefs) {
    SymbolTable ret = new SymbolTable();
    ret.declared.addAll(oldDefs.declared);
    ret.declared.addAll(newDefs.declared);
    return ret;
  }

  public static SymbolTable append(
      Identifier name, SymbolTable defs) {
    SymbolTable ret = new SymbolTable();
    ret.declared.addAll(defs.declared);
    if (!ret.declared.add(name.getValue())) {
      System.out.println("Duplicate identifier : "+name.getValue());
    }
    return ret;
  }

  public static void checkDeclaration(
      Identifier name, SymbolTable defs) {
    if (defs.declared.contains(name.getValue())) {
      System.out.println("Undeclared identifier : "+name.getValue());
    }
  }
}
```

be observed: (1) `StatementPart` is declared to contain multiple `Statements` rather than using left recursion and (2) the synthesised attribute *name* of `Identifier` now is a property of the node (and is called `value`).

Behaviour is added to this tree definition with the logic specification given in listing D.3. This listing clearly shows a major difference between attributes grammars and TPL: multiple visiting methods are present on a node. In this example this is caused by the fact that the given grammar requires a 2-pass evaluation. In the first pass the attributes *original* and *updated* are evaluated and the second pass evaluates the attribute *available*. The TPL implementation takes a slightly different approach, by only performing two passes over the statements, instead of the entire tree.

Listing D.3: Declare and use logic specification

```
package decluse.eval;

header {
  import decluse.SymbolTable;
  import static decluse.SymbolTable.*;
}

Program {
  visit() {
    #block.visit(newTable());
  }
}

Block {
  visit(SymbolTable available) {
    SymbolTable updated = #statementPart.visitCollect(newTable());
    #statementPart.visitCheck(concatenate(updated, available));
  }
}

StatementPart {
  SymbolTable visitCollect(SymbolTable original) {
    SymbolTable updated = original;
    #foreach(#stmt <- #statements) {
      updated = #stmt.visitCollect(updated);
    }
    return updated;
  }

  visitCheck(SymbolTable available) {
    #foreach(#stmt <- #statements) {
      #stmt.visitCheck(available);
    }
  }
}

abstract Statement {
  SymbolTable visitCollect(SymbolTable original);
  visitCheck(SymbolTable available);
}

VarStatement extends Statement {
  SymbolTable visitCollect(SymbolTable original) {
    return append(#name, original);
  }

  visitCheck(SymbolTable available) {}
}

UseStatement extends Statement {
  SymbolTable visitCollect(SymbolTable original) {
    return original;
  }

  visitCheck(SymbolTable available) {
    checkDeclaration(#name, available);
  }
```

```
}

BlockStatement extends Statement {
  SymbolTable visitCollect(SymbolTable original) {
    return original;
  }

  visitCheck(SymbolTable available) {
    #block.visit(available);
  }
}
```

In general the evaluation of an attribute grammar can be performed in any order. The evaluation starts with all attributes undefined, except for special attributes such as the value of a terminal. The evaluator picks an attribute for which the input is defined and calculates the output. This operation is repeated until either all attributes are defined, or no attributes remain for which the input is defined.

TPL however does not incorporate a full attribute evaluator. It is the task of the developer to define the order in which the attributes are evaluated. In this case two passes are needed over all statements. The attributes used in the TPL version are still the same. However, they are spread over multiple visiting methods. Some attribute grammars, namely the class of L-attribute grammars (see [Alblas and Nymeyer, 1996, Section 9.2]), can be evaluated in a single pass over the tree. These attribute grammars can be implemented directly in TPL.

Appendix E

Calculation Language Code Listings

E.1 Sample Input

Listing E.1: Calculation language sample input

```
var a;
var b;

a := 3;
b := 5;

if (a - b) {
  print (a + b);
} else {
  print (a - b);
};
```

E.2 Driver

Listing E.2: Calculation language driver

```
package calc;

import nl.utwente.ewi.tpl.runtime.antlr.TPLTreeAdaptor;
import org.antlr.runtime.ANTLRFileStream;
import org.antlr.runtime.CharStream;
import org.antlr.runtime.CommonTokenStream;

public class CalcDriver {
  public static void main(String[] args) throws Exception {
    CharStream calcInput = new ANTLRFileStream(args[0]);
    CalcLexer calcLexer = new CalcLexer(calcInput);
    CommonTokenStream calcTokens = new CommonTokenStream(calcLexer);
    CalcParser calcParser = new CalcParser(calcTokens);

    calcParser.setTreeAdaptor(
        new TPLTreeAdaptor(new calc.PackageDescription()));

    Program program = (Program) calcParser.program().getTree();
    System.out.println("Program is:\n"+program);
    program.check();
    System.out.println("Running...");
    program.run();
  }
}
```

E.3 Parser Specification

Listing E.3: Calculation language parser specification

```
grammar Calc;

options {
  output = AST;
}

tokens {
  PROGRAM;
  DECLARATION;
  IF;
  PRINT;
  BLOCK;
  ADD;
  MINUS;
  ASSIGN;
}

@lexer::header {
  package calc;
}

@parser::header {
  package calc;
}

program
  : declaration* statementBlock
    -> ^(PROGRAM declaration* statementBlock)
  ;

declaration
  : 'var' IDENTIFIER ';'
    -> ^(DECLARATION IDENTIFIER)
  ;

statement
  : expression ';'!
  | ifStatement
  | printStatement
  ;

expression
  : IDENTIFIER ':=' expression
    -> ^(ASSIGN IDENTIFIER expression)
  | addSubExpression
  ;

addSubExpression
  : atomicExpression '+' addSubExpression
    -> ^(ADD atomicExpression addSubExpression)
  | atomicExpression '-' addSubExpression
    -> ^(MINUS atomicExpression addSubExpression)
  | atomicExpression
  ;
```

```
55
atomicExpression
  : IDENTIFIER
  | NUMBER
  ;
60
ifStatement
  : 'if' '(' expression ')' '{' ifThen=statementBlock '}' 'else'
    '{' ifElse=statementBlock '}' ';'
    -> ^(IF expression $ifThen $ifElse)
65  ;

printStatement
  : 'print' '(' expression ')' ';'
    -> ^(PRINT expression)
70  ;

statementBlock
  : statement*
    -> ^(BLOCK statement*)
75  ;

IDENTIFIER
  : ('a'..'z'|'A'..'Z')+
  ;
80
NUMBER
  : ('0'..'9')+
  ;

85 WS
  : ( ' ' | '\t' | '\f' | '\r' | '\n')+ { $channel=HIDDEN; }
  ;
```

E.4 Tree Definition

Listing E.4: Calculation language tree definition

```
package calc;

Program(Declaration* declarations, StatementBlock statements) -> PROGRAM;
Declaration(Identifier variable) -> DECLARATION;
5
abstract Statement;
IfStatement(Expression condition, StatementBlock thenPart,
    StatementBlock elsePart) extends Statement -> IF;
PrintStatement(Expression message) extends Statement -> PRINT;
10 StatementBlock(Statement* statements) extends Statement -> BLOCK;

abstract Expression extends Statement;
abstract BinaryExpression(Expression left, Expression right) extends Expression;
AddExpression extends BinaryExpression -> ADD;
15 MinusExpression extends BinaryExpression -> MINUS;
AssignExpression(left as Identifier var) extends BinaryExpression -> ASSIGN;

abstract Literal<T>(T value) extends Expression;
```

```
Number(value as Integer) extends Literal<Integer> -> NUMBER;
Identifier(value as String) extends Literal<String> -> IDENTIFIER;
```

E.5 Context Checker

Listing E.5: Calculation language context checker

```
package calc.context;

header {
  import java.util.Set;
  import java.util.HashSet;
}

Program {
  interface public boolean check() {
    int errors = #this.visit();
    if (errors == 0) {
      System.out.println("No errors found.");
      return true;
    } else {
      System.err.println(errors + " errors found.");
      return false;
    }
  }

  int visit() {
    int errors = 0;
    Set<String> symbols = new HashSet<String>();
    #foreach(#curId <- #declarations/variable) {
      errors += #curId.visitDeclare(symbols);
    }
    #foreach(#curId <- #statements//*[Identifier]) {
      errors += #curId.visitUse(symbols);
    }
    return errors;
  }
}

Identifier {
  int visitUse(Set<String> symbols) {
    if (!symbols.contains(#value)) {
      System.err.println("Undeclared variable '"+#value+"'.");
      return 1;
    }
    return 0;
  }

  int visitDeclare(Set<String> symbols) {
    if (symbols.contains(#value)) {
      System.err.println("Variable '"+#value+"' already declared.");
      return 1;
    }
    symbols.add(#value);
    return 0;
  }
```

```
}
```

E.6 Interpreter

Listing E.6: Calculation language interpreter

```
package calc.interpreter;

header {
  import java.util.HashMap;
  import java.util.Map;
}

Program {
  interface public void run() {
    #this.visit();
  }

  visit() {
    Map<String, Integer> store = new HashMap<String, Integer>();
    #foreach(#decl <- #declarations) {
      #decl.visit(store);
    }
    #statements.visit(store);
  }
}

Declaration {
  visit(Map<String, Integer> store) {
    store.put(#variable.visitWrite(), 0);
  }
}

abstract Statement {
  visit(Map<String, Integer> store);
}

IfStatement extends Statement {
  visit(Map<String, Integer> store) {
    if (#condition.visitExpr(store) != 0) {
      #thenPart.visit(store);
    } else {
      #elsePart.visit(store);
    }
  }
}

StatementBlock extends Statement {
  visit(Map<String, Integer> store) {
    #foreach(#stmt <- #statements) {
      #stmt.visit(store);
    }
  }
}

PrintStatement extends Statement {
```

```
  visit(Map<String, Integer> store) {
    System.out.println(#message.visitExpr(store));
  }
}

abstract Expression extends Statement {
  visit(Map<String, Integer> store) {
    #this.visitExpr(store);
  }

  int visitExpr (Map<String, Integer> store);
}

AssignExpression extends Expression {
  int visitExpr(Map<String, Integer> store) {
    int value = #right.visitExpr(store);
    store.put(#var.visitWrite(), value);
    return value;
  }
}

AddExpression extends Expression {
  int visitExpr(Map<String, Integer> store) {
    return #left.visitExpr(store) + #right.visitExpr(store);
  }
}

MinusExpression extends Expression {
  int visitExpr(Map<String, Integer> store) {
    return #left.visitExpr(store) - #right.visitExpr(store);
  }
}

Number extends Expression {
  int visitExpr(Map<String, Integer> store) {
    return #value;
  }
}

Identifier extends Expression {
  int visitExpr(Map<String, Integer> store) {
    return store.get(#value);
  }

  String visitWrite() {
    return #value;
  }
}
```

E.7 Pretty Printer

Listing E.7: Calculation language pretty printer

```
package calc.pretty;

Node: abstract Printable {
```

```
  interface public String toString() {
    return #this.visit(0);
  }

  String visit(int indentation);
}

abstract IndentedConstruct extends Printable {
  implementation protected String writeIndentation(int indentation) {
    String ret = "";
    for (int count=0; count<indentation; count++) {
      ret += "  ";
    }
    return ret;
  }
}

Declaration extends IndentedConstruct {
  String visit(int indentation) {
    return writeIndentation(indentation) + "var " +
        #variable.visitExpr() + ";\n";
  }
}

Program extends IndentedConstruct {
  String visit(int indentation) {
    String ret = "";
    #foreach(#decl <- #declarations) {
      ret += #decl.visit(indentation);
    }
    ret += #statements.visit(indentation);
    return ret;
  }
}

IfStatement extends IndentedConstruct {
  String visit(int indentation) {
    return writeIndentation(indentation) +
      "if (" + #condition.visitExpr() + ") {\n" +
      #thenPart.visit(indentation+1) +
      writeIndentation(indentation) + "} else {\n" +
      #elsePart.visit(indentation+1) +
      writeIndentation(indentation) + "}\n";
  }
}

PrintStatement extends IndentedConstruct {
  String visit(int indentation) {
    return writeIndentation(indentation) +
      "print (" + #message.visitExpr() + ");\n";
  }
}

StatementBlock extends IndentedConstruct {
  String visit(int indentation) {
    String ret = "";
    #foreach(#stmt <- #statements) {
      ret += #stmt.visit(indentation);
```

```
    }
    return ret;
  }
}

abstract Expression extends IndentedConstruct {
  String visit(int indentation) {
    return writeIndentation(indentation) + #this.visitExpr() + ";\n";
  }

  String visitExpr();
}

abstract BinaryExpression extends Expression {
  implementation abstract protected String getOperator();

  String visitExpr() {
    return #left.visitExpr() + " " + getOperator() + " " + #right.visitExpr();
  }
}

AddExpression extends BinaryExpression {
  implementation protected String getOperator() {
    return "+";
  }
}

MinusExpression extends BinaryExpression {
  implementation protected String getOperator() {
    return "-";
  }
}

AssignExpression extends BinaryExpression {
  implementation protected String getOperator() {
    return ":=";
  }
}

Literal extends Expression {
  String visitExpr() {
    return #value.toString();
  }
}
```

Appendix F

Planning

Project Phase	Days	Start Date	End Date
Requirements analysis and design	53	25-04	17-08
Prototype implementation	30	18-08	28-09
Design refinements and syntax fixation	20	29-09	26-10
Final product implementation	35	27-10	14-12
Total	**138**		

F.1 Requirements Analysis and Design

Task	Days	Start Date	End Date
Investigation other parsers	1	25-04	25-04
Concretize concepts	3	26-04	28-04
Preliminary requirements analysis	3	01-05	05-05
Planning	1	08-05	08-05
Formalising language concepts	10	10-05	24-05
Formal syntax specification	5	26-05	02-06
Software design	15	05-06	28-06
Testcase design	15	30-06	17-08
Total	**53**		

F.2 Prototype Implementation

Task	Days	Start Date	End Date
Prototype implementation	20	18-08	14-09
Testcase implementation	10	15-09	28-09
Total	**30**		

F.3 Design Refinements and Syntax Fixation

Task	Days	Start Date	End Date
Problem identification	3	29-09	03-10
Update requirement analysis	2	04-10	05-10
Update formalising language concepts	3	06-10	10-10
Update formal syntax specification	2	11-10	12-10
Update software design	5	13-10	19-10
Update testcase design	5	20-10	26-10
Total	**20**		

F.4 Final Product Implementation

Task	Days	Start Date	End Date
Final product implementation	10	27-10	09-11
Testcase implementation	5	10-11	16-11
Documentation	20	17-11	14-12
Total	**35**		

Bibliography

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 1986. ISBN 0321486811.

Henk Alblas and Albert Nymeyer. *Practice and Principles of Compiler Building with C*. Prentice Hall, 1996. ISBN 0133492672.

ANTLR. ANTLR parser generator, 2006. URL `http://www.antlr.org/`. Last checked at 2006-05-17.

Daniel Bonniot. Structural interfaces for $ML_{\leq}$, April 2000. URL `http://nice.sourceforge.net`.

Mark van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. A generator of efficient strongly typed abstract syntax trees in java. *IEE Proceedings - Software*, 152(2):70–78, 2005. URL `http://link.aip.org/link/?IPS/152/70/1`.

Martin Bravenboer and Eelco Visser. Guiding visitors: Separating navigation from computation. Technical Report UU-CS-2001-42, Institute of Information and Computing Sciences, Utrecht University, 2001.

B. Bryant and B.-S. Lee. Two-level grammar as an object-oriented requirements specification language. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, volume 9, page 280, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1435-9.

James Clark and Steve DeRose. XML path language (XPath). W3C recommendation, World Wide Web Consortium, November 1999. URL `http://www.w3.org/TR/1999/REC-xpath-19991116`.

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 130–145, October 2000.

Arie van Deursen and Joost Visser. Source model analysis using the JJTraveler visitor combinator framework. *Software Practice and Experience*, 34:1345–1379, 2004.

Eclipse. Eclipse.org home, 2007. URL `http://www.eclipse.org`. Last checked at 2007-02-18.

Peter van Eijk, Axel Belinfante, Henk Eertink, and Henk Alblas. The term processor generator Kimwitu. In *TACAS '97: Tools and Algorithms for Construction and Analysis of Systems*, pages 96–111, Enschede, The Netherlands, April 1997. Springer.

Andrea Flexeder, Michael Petter, Scott E. Hudson, C. Scott Ananian, Frank Flannery, Dan Wang, and Andrew W. Appel. CUP LALR parser generator for java, 2006. URL `http://www2.cs.tum.edu/projects/cup/`.

Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 140–154, Washington, DC, USA, August 1998. IEEE Computer Society. URL `http://sablecc.org/index.html`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Boston, Mass., third edition, 2005. ISBN 0-321-24678-0. URL `http://java.sun.com/docs/books/jls`.

Ouafa Hachani and Daniel Bardou. Using aspect-oriented programming for design patterns implementation, 2002.

JavaCC, 2006. URL `https://javacc.dev.java.net/`. Last checked at 2006-05-17.

JJTree, 2006. URL `https://javacc.dev.java.net/JJTree.html`. Last checked at 2006-05-17.

JUnit, 2006. URL `http://www.junit.org`. Last checked at 2006-08-08.

Gerwin Klein. JFlex: The fast lexical analyser generator, 2005. URL `http://jflex.de`.

Gerwin Klein and Alfons Brandl. classgen, 2003. URL `http://classgen.sourceforge.net/index.html`.

Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

T. Kuipers and J.M.W. Visser. Object-oriented tree traversal with JJForester. *Science of Computer Programming*, 47(1):59–87, April 2003. doi: doi:10.1016/S0167-6423(02)00108-9.

Memphis. *Memphis C/C++: A Language for Compiler Writers*. URL `http://memphis.compilertools.net/`. Last checked at 2007-02-28.

Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd IEEE Int. Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.

Terence Parr. ANTLR v3, 2006. URL `http://www.antlr.org/v3`. Last checked at 2006-11-10.

Terence Parr. Enforcing strict model-view separation in template engines. In *WWW '04: The World Wide Web conference*, pages 224–233, New York, NY, USA, May 2004. ACM Press. URL `http://www.stringtemplate.org`.

T.J. Parr and R.W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.

SLADE, 2002. URL `http://fmt.cs.utwente.nl/tools/slade/`. Last checked at 2006-05-17.

S. Doaitse Swierstra, Pablo R. Azero Alcocer, and Joao Sariava. Designing and implementing combinator languages. In *AFP '98: Advanced Functional Programming*, pages 150–206, September 1998.

Kevin Tao and Jens Palsberg. The Java Tree Builder, 2005. URL `http://compilers.cs.ucla.edu/jtb/`. Last checked at 2006-05-17.

J.M.W. Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 270–282, New York, NY, USA, October 2001. ACM Press. ISBN 1-58113-335-9. doi: http://doi.acm.org/10.1145/504282.504302.

John Vlissides. Visitor in frameworks, November 1999. C++ Report.

Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr Abstract Syntax Description Language. In *DSL '97: Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

Rhys Weatherley. Treecc: An aspect-oriented approach to writing compilers. January 2002.

Xiaoqing Wu, Suman Roychoudhury, Barrett R. Bryant, Jeffrey G. Gray, and Marjan Mernik. A two-dimensional separation of concerns for compiler construction. In *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 1365–1369, New York, NY, USA, March 2005. ACM Press. ISBN 1-58113-964-0. doi: http://doi.acm.org/10.1145/1066677.1066985.

Xiaoqing Wu, Barrett R. Bryant, Jeff Gray, Suman Roychoudhury, and Marjan Mernik. Separation of concerns in compiler development using aspect-orientation. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1585–1590, New York, NY, USA, April 2006. ACM Press. ISBN 1-59593-108-2. doi: http://doi.acm.org/10.1145/1141277.1141646.