Diploma Thesis

Specification-Driven Generation of an Extensible C Compiler

submitted by

Christoff Bürger
born 04.06.1983 in Weimar

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisors: Associate Professor Görel Hedin,
Dipl.-Inf. Andreas Bartho
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted 25th September 2008

Contents

1 Introduction **1**
1.1 Thesis Objectives 2
1.2 Statement of Requirements 2
1.3 A few Words about *C* 3
1.4 Thesis Outline 5

2 Spec.-Driven Compiler Development **7**
2.1 Implementation Objectives and Requirements 7
2.2 Theoretical Concepts 9
2.3 Applied Tools 12
2.3.1 *JastAdd* 13
2.3.2 *Beaver* 14
2.3.3 *JFlex* 16
2.4 Integrating the Tools 17
2.5 Example Compiler Project 20

3 Compiler Modularization **31**
3.1 Modularization Across *Compilation-Phases* 32
3.2 Modularization Across Concerns 33
3.2.1 Basic Decomposition Concept 34
3.2.2 *Compiler-Core* and Extensions 36
3.3 Unresolved Problems 39

4 Lexical and Syntactical Components **41**
4.1 Lexical Components 41
4.1.1 Token Recognition and Construction 42
4.1.2 *JastAddC's Lexer-Stack* 47
4.2 Syntactical Components 51
4.3 Unresolved Problems - The *Typedef Problem* 54
4.3.1 *Typedef Problem*: Overview 54
4.3.2 *Typedef Problem*: Examples 55

5 Semantical Components **59**
5.1 *Core-Concerns*: Tasks and Interactions 59
5.1.1 Compiler Tasks and their Interactions 60
5.1.2 A Generic Error Handling Approach 61
5.1.3 *JastAddC's Core-Concerns* 64
5.2 *Core-Concerns*: Implementations 66

5.2.1 Error Handling Module 67
5.2.2 Access Support Module 72
5.2.3 Constraint Verification Module 76
5.3 Generic Mapping Technique 77
5.3.1 Generic Mapping Technique: Motivation 78
5.3.2 Generic Mapping Technique: Realization 85
5.4 Unresolved Problems 94

6 *JastAddC's* Development Status 95
6.1 Implemented *C* Parts 95
6.2 Missing *C* Parts 96

7 Further Development 99
7.1 Full Modularization Support 99
7.1.1 Overview: Modularization Problems 99
7.1.2 Solution Concept: SGLR Parsing 100
7.2 Disambiguation Techniques 105
7.2.1 Disambiguation Concept for *JastAdd* 106
7.2.2 Integrating Disambiguation and Mapping Techniques . . . 110
7.3 Code Generation 113
7.4 Further Work Schedule 115

8 Conclusions 119

A ISO/IEC 9899:1999 Grammar 123

B *JastAddC's Compiler-Core* 141
B.1 *Core-AST* 141
B.2 *Core-Concerns* 151

Bibliography 162

List of Figures 164

List of Tables 165

List of Listings 168

Chapter 1

Introduction

Compilers are one of the most important tools in computer science. Their classical domain is to transform human readable source code of a programming language to executable programs or programs for an abstract machine. To accomplish this task knowledge about theoretical concepts is needed. Examples include deterministic language recognition and attribute grammars, computer architecture and design, special hardware design and instruction sets, program optimization and mathematical theories. But compilers not only unify knowledge of different fields in computer science. The parts of their implementation realising the requirements of one field also highly depend on and interact with other parts. Thus compilers form very complex and hard to modularize software applications resulting in an implementation that is clumsy and complicated to maintain. But as compilers are such important tools, they need to be up to date. Requirement changes in the hardware design or software development process result in requirement changes of compilers. There is a need for easy compiler maintenance and extensibility, as well as development.

One way to tackle these problems is to use the basic concept of compilers to develop compilers, thus to use specialised compilers taking specifications describing aspects of a compiler and generating the specified compiler parts. The idea is, that the development, maintenance and extension of such specifications is much more easy than changing the source code. Unfortunately, compiler-compilers are often only concerned about a very small part of a complete compiler, like the generation of a parser. Most compilers are also not generated.

Recently there is a lot of work on improving the modularization of compilers, based on attribute grammars, e.g. the *JastAdd* project. Improvements concern extending attribute grammars with new concepts such as support for collection attributes, circular attributes, graphs and for rewriting the abstract syntax tree (AST).

1.1 Thesis Objectives

The goal of this thesis is to show an example for a complete specification driven generated compiler front end, the *JastAddC* compiler. The example front is implemented in *Java* [13, 35, 19, 20] and it realises the reading of the source code file up to a statically evaluated AST for the *C* programming language. The thesis will show how the specifications generating the compiler can be separated in different modules, each representing one aspect of the compiler. It will be explained how the compiler can be extended by adding new specifications. The advantages for the maintainability of the generated compiler will be pointed out. It will become clear that the specifications are much easier to understand and read, thus to develop, than a classical source code implementation approach. The *JastAdd* rewritable, circular, reference attribute grammar (ReCRAG) evaluator-generator is the basic compiler-compiler used, supported by the *Beaver* parser-generator and the *JFlex* lexer-generator. Of key importance to reach the goals mentioned are the attribute grammar (AG) extensions supported by *JastAdd*. It will be shown, how semantical extensibility and modularization can be archived using *JastAdd*'s features.

1.2 Statement of Requirements

This thesis is no introduction to compiler construction or theoretical concepts about language recognition and semantics. Rather, the reader is expected to know already about these concepts. It is highly recommended to be familiar with:

1. Lexical analysis concepts like: deterministic finite automata (DFA), the *principle of the longest match*

2. Syntactical analysis concepts: context-free grammar (CFG), Backus-Naur Form (BNF), extended Backus-Naur Form (EBNF), left to right, k signs look ahead, rightmost derivation (LR(k)) grammars, look-ahead optimized LR(k) (LALR(k)) grammars, push down automata (PDA), non-deterministic PDA (NPDA), deterministic PDA (DPDA), LR(k) parser, LALR(k) parser, generalized LR(k) (GLR) parser, scannerless GLR (SGLR) parser, concrete syntax tree (CST)

3. Semantical analysis concepts: AST, AG

If questions about these concepts arise while reading the thesis or conclusions can't be understand, because of a lack of knowledge in lexical, syntactical or semantical theories, the literature presented in the following can be consulted.

[1] is often considered to be the bible of compiler construction, presenting most common techniques. The book introduces lexical and syntactical analysis methods, type checking, code generation and optimization and the run-time environment. AGs are also investigated under the topic *Syntax directed translation*. Even the second edition handles newer concepts like just in time compiling, parallel machines, data flow analysis and garbage collection the book's concepts are quite old basic concepts, which are nevertheless useful, but sometimes one can feel the age of the book. Extensible compiler construction isn't even considered. Especial newer attribute grammar concepts like ReCRAG are

not handled. Though, the book is still one of the best choices to get an overview about compiler construction.

[42] is a good introduction to AGs. Different attribute evaluation strategies and AG classes are discussed. Also the lexical analysis as well as left to right, k signs look ahead, rightmost derivation (LR(k)), look-ahead optimized LR(k) (LALR(k)) and LL(k) parsing are covered. The focus is on the development of syntax based tools. Throughout the book AGs and their definition and interaction for the development of programming languages based on a context-free grammar (CFG) are investigated.

[2] gives a good introduction to the basics of theoretical computer science (grammar classes, CFG, decision problems, complexity, push down automata (PDA), non-deterministic finite automata (NFA), deterministic finite automata (DFA) etc.). The concepts are explained as well as formally introduced. Additionally the most important proofs are provided.

[43] and [44] present and prove the most important parsing theories in detail. Especially for deterministic languages nearly every aspect is covered and proved.

[37] gives a good introduction into generalized LR(k) (GLR) parsing and [47] provides a detailed overview about scannerless GLR (SGLR) parsing.

[31] is considered to be the first formal presentation of AGs. It is important to mention the correction [32] the author released 3 years later.

[11] and [36] are two phd-thesis, each consisting of several papers, which present advanced AG techniques as well as their application. Both thesis cover ReCRAG and describe their properties as well as the implementation for the additional AG concepts. Additionally [36] introduces the concept of *collection attributes* into ReCRAGs.

Though, it is a long way for a reader starting from scratch until getting into the more specialized problems[1] handled by this thesis.

1.3 A few Words about *C*

The *JastAddC* compiler front-end presented in the thesis uses *C* as example language to demonstrate extensible, declarative, specification driven compiler construction. It is important to know, that **no** complete *C* compiler front-end is implemented, but instead the most complicated and interesting concepts of *C*. There exist many *C* versions and standards. [21] distinguishes four *C* generations:

1. Traditional *C*: Everything before the first *C* standard (1989)

2. Standard *C* (1989): The first *C* standard, also named *American National Standards Institute (ANSI) C* or *C89*.

3. Standard *C* (1995): *American National Standards Institute (ANSI) C* together with it's two Technical Corrigenda and an Amendment (extension).

4. Standard *C* (1999): *ISO/IEC 9899:1999*, also named *C99*. *C99* is a replacement of all previous *C* standards. It extends *C95* with many new concepts. Of course, the extensions do not break the "Spirit of *C*". There exist two Technical Corrigenda for *C99*.

[1] Mainly the specification-driven development of extensible compilers.

All parts implemented in *JastAddC* conform to the *C99* standard, as far as there exists any implementation for a semantical concept in *JastAddC* at all. E.g. the generated parser is based on the official *C99* grammar and the name analysis conforms to the standard, except for external translation unit name resolution, because external semantics are not implemented yet (*JastAddC* has no preprocessor, so the `include` directive is unknown). In further chapters with *C* always *C99* is meant and *C99* is defined only by the *ISO/IEC 9899:1999* [23].

But the reader should keep in mind: *C* is not completely specified. There exist many constructs and semantics, which are implementation defined. Also, as most *C* developers know, *C* compilers tend to be vendor specific. Most *C* compilers introduce their own language extensions and special functionalities. Everyone, who ever tried to switch the compiler used to process some major *C* project knows: this is an awful task. Of course, if the program would only be written using standardized features, switching the compiler should be no problem. But most times big *C* projects don't use only standardized features. Additionally, machine dependencies may be encountered. There exists a lot of legacy code only waiting to give someone some headache. The *JastAddC* implementation is not concerned about vendor specific extensions and machine dependencies. But *JastAddC's* easy extensibility offers the basis to change the compiler in ways needed to be able to handle *C* programs, which depend on specific compiler behavior for unspecified language constructs.

[26] says *C* is no big programming language. Even if it is true, the complicated part is not the size, but the possibility to combine nearly all the constructs. Famous specialities of *C* like the comma operator, the fact that assignments are usual expressions or the array access by index, which can for an integer value `i` and a pointer `p` to an value object be expressed as `p[i]`, as well as `i[p]`, do not only look strange, they introduce a lot more implementation effort for a compiler developer. [21] and [33] give a good overview about complicated parts of *C* as well as it's complexity and scale. The most complicated part in the *C* language are declarations. In *C99*, declarations even become more complicated. *JastAddC* handles full *C99* declarations and a complete section in this thesis will be spent to describe a technique to map *C* declarations to very easy to read and evaluate ASTs. The downside is, that the reader should be familiar with *C*. Good knowledge in programming is a requirement to understand many of the complicated constructs.

In general, *C* is not simply a programming language, it is the incarnation for a complete computer design, the *von Neumann architecture*. One may love *C* or hate it, but as long as computers are build like nowadays (addressable memory, fixed instruction set working on fixed data types) *C* will not die. And good knowledge in *C* requires good knowledge in the way computers work. To learn and understand *C* is a good idea. Many *C* constructs look complicated and illogical at a first sight, but *C* turns out to be a consistent, logical language: A language, where hardware design, operation and programming work hand in hand.

Hopefully, this thesis helps to show how a consistent and logical programming language can be implemented in a consistent and logical way using declar-

ative, theoretically backed concepts. And maybe a few *C* constructs become clear for the uncomfortable *C* programmer.

1.4 Thesis Outline

The next chapter will introduce the *specification-driven* compiler development used to implement the *JastAddC C* compiler front-end. The requirements and objectives of such an implementation approach are presented as well as the main concepts and techniques used to achieve them. Afterwards, the used tools are investigated and their integration to generate a complete compiler is shown. A small example compiler project concludes the chapter. The basic techniques presented in chapter 2 will be applied in chapter 3, 4 and 5 to implement *JastAddC*.

Chapter 3 presents the architecture and modularization concept of *JastAddC*. Beside defining compiler parts, interfaces and a composition concept, the way extensions are handled will be shown. Unfortunately, our high modularization demands cannot be met by the lexical and syntactical components. The conceptual problems will be explained.

Afterwards, chapter 4 explains the lexical and syntactical implementation and their problems, especially the *typedef problem*.

Chapter 5 concludes the description of *JastAddC's* implementation. It will be explained, how to implement a compiler's semantics by introducing a generic error handling concept, a module supporting AST information access methods (especially context information) and a module handling the actual semantic checks. Also, a technique supporting the construction of nice ASTs from concrete syntax tree (CST)s will be introduced. The chapter is completed by an analysis of remaining semantical problems and minor improvement suggestions for *JastAdd*.

Chapter 6 gives a short overview about the development status of *JastAddC*. The *C* parts not implemented will be listed.

Chapter 7 comes back to the lexical, syntactical and semantical problems mentioned and suggests solutions, which are not yet integrated in *JastAddC* or not supported by the used tools. A brief work schedule for the further *JastAddC* development will be proposed.

Finally, chapter 8 summarizes the main concepts introduced, lists the contributions of this thesis and investigates related work.

Throughout the thesis links to the standard [23] are given, so the interested reader can look up associated language specifications. Links are embedded in squared brackets, in the format [*chapter*:*paragraphs*;p. *pages*]. Often comments in *JastAddC*'s specifications also contain such references, which represent a mapping between a language constraint and its implementation.

Chapter 2

Specification-Driven Compiler Development

This chapter introduces the *specification-driven* compiler development step by step, starting on an abstract level down to an example implementation. First, the implementation objectives and requirements for the *specification-driven* development of extensible compilers are investigated. Of course, the *JastAddC C* compiler-frontend presented in the thesis shares these requirements and objectives. Afterwards, the theoretical concepts of the tools used to implement *JastAddC* as well as the concepts of applied techniques are investigated. An introduction to the tools used follows. The next section shows how a complete compiler can be composed using the tools just presented. Finally a little example compiler project is developed, so the concepts and tools presented so far are illustrated. The *JastAddC* compiler is implemented in a similar way, just its scale is much greater.

2.1 Implementation Objectives and Requirements

The main objective of the thesis is to show how complete compilers can be generated and extended only using specifications.

The specifications must be based on mathematical formalisms and have to be easy to implement and understand, such that compiler tasks can be implemented in a "natural way". In a "natural way" means on the one hand, that each specification is similar to an equal formal specification, as one would write it down on paper[1], and on the other hand, that the specifications are comprehensible and describe a problem's properties as close to an informal problem description as possible, i.e. it is easy to understand by humans. The tool processing a specification generates a solution: A compiler component.

Extensibility means specifications for additional language constructs or semantic analysis can be added to an already existing compiler, such that the complete compiler generated by executing all the specifications also supports the additional language constructs. Thus, the compiler must support a modularization concept on the specification level.

[1] Not related to any specific tool. Hence, the specification does not bother about technical details.

The tools for which specifications are written must be theoretically well-founded. This means specifications must depend on a theoretical formalism. Additionally, if a tool accepts a specification, the generated compiler part must be complete and correct regarding the theoretical formalism's semantics. Thus, specifications must be verifiable. The used specification languages should be on the one hand as powerful as possible and on the other hand comprehensible.

The interaction between specifications should be theoretically backed. Two interaction scenarios between specifications exist: The interaction between specifications of the same specification language and the interaction between specifications of different specification languages. The interaction between specifications of the same specification language should be covered by the specification language's theoretical formalism. The interaction between different specification languages should be covered by the theory of the formal language classes (*Chomsky hierarchy*) or the theoretical formalism of one specification language depends explicitly on the theoretical formalism of another specification language. An example for the interaction between formal language classes is the interaction between regular languages and context-free languages, where the lexical analysis's results are the input for the syntactical analysis. An example for a formalism, which is based explicitly on another formalism are AGs, which are based on CFGs. Hence the syntax tree constructed by the syntactical analysis is the input for the semantical evaluator.

The requirements mentioned so far are necessary to support the thesis's main objective: The construction of maintainable, extensible compilers. If a component model with composition operators for each specification language exists, the specification driven approach enables the composition of compiler modules consisting of specifications. For specifications based on theoretically well-founded formalisms, such component models are already investigated. Formal language closures are known and proved [43, 44, 2]. Thus, it is known if a formal language class is closed under *unification*, *intersection*, *complement*, *difference*, *Klenee star* and *concatenation*. Automatically construction processes, as well as their complexity, for any of the operations are known. Thus, it is possible to conclude based on the formalism a tool depends on, if automatic specification extensibility can be achieved or not. Of course, the tool still has to implement the formalism's composition operators. If all specification languages are closed under *unification* the complete compiler can be specified in several modules. Each module contains an arbitrary number of specifications. To generate the compiler, each tool can simply weave all its specifications together. After doing so, the tool can generate its compiler part based on the resulting unified specification. Alternatively, the tool could also weave the results generated for its specifications together.

Hence, compared to a classical handcrafted compiler implementation extensibility is supported by the compiler generator tools, based on *unification* operations for each specification formalism.

If all of the mentioned requirements are met, the theoretically well-founded, comprehensible construction of maintainable and extensible compilers becomes possible. Theoretical well-foundeness is achieved because only theoretically well-founded tools are used. Comprehensibility is achieved since the specification languages raise the level of abstraction and are powerful enough to specify even

complicated computations in a "natural way". Maintainability is achieved, because specifications are easier to maintain than a source code implementation. Extensibility is achieved, because a modularization concept on the specification level is provided (see chapter 3) and the requirements described above for the interaction between specifications are met.

It is not by chance, that the objectives and requirements are not specifically associated with *JastAddC*. One objective of the thesis is to introduce general concepts realizing the specification driven generation of extensible compilers. The language *C* has been chosen as a realistic example language, so it can be proved, that the concepts and techniques presented are not only applicable for academic toy languages, but can also handle real languages with all their rough edges. Especially the techniques of the main chapter 5, the error handling concept (chapter 5.2.1), the AST access support (chapter 5.2.2), the constraint checking concept (chapter 5.2.3) and the mapping concept (chapter 5.3) can be applied for many different languages. But also the suggested solutions for the further development, like the disambiguation technique (chapter 7.2), are useable for most real languages. One can even go so far to adapt the *JastAddC* compiler itself to support another imperative language or a specific *C* derivate.

However, *JastAddC*'s implementation and the used tools cannot always meet the objectives and requirements. Especially the modularization/extensibility on the lexical as well as syntactical level are not yet sufficient enough (see chapter 3.3). The interaction between compiler components cannot be achieved completely automatically. *Glue-code*[2] is needed. The *glue-code* is realized by statically hand written *Java* classes, which call generated components or are called by generated components[3].

Most generator tools also support the possibility to embed source code in specifications, which has to be done with caution. The *JastAddC* implementation tries not to overuse this feature. If one of the objectives or requirements mentioned in this chapter cannot be met, it will be noted.

2.2 Theoretical Concepts

The objectives can only be achieved if the requirements for the specification languages and their generators are met. Hence, the concepts of the tools used to generate the *JastAddC* compiler highly determine the implementation's quality. This section discusses the main concepts applied. The used tools realizing these concepts will be presented in section 2.3.

Generating compiler parts is well known. Especially the generation of lexers and parsers is an established technique. Common tools are the lexer generator

[2] *Glue-code* is code needed to connect components, which depend on each other. The *glue-code* for a component *A*, which depends on a component *B*, realizes an adapter, which provides the transformation of information so *A* can communicate with *B*. *Glue-code* never computes anything new or contains any semantics, except for the exchange of information. Thus *glue-code* does also not transform the exchanged information in such a way, that the information's semantics change. *Glue-code* does not add, leave or manipulate an information's value.

[3] Of course, there is no need to change generated code manually after generation.

Lex [34] and the parser generator *Yacc* [24] and their successors in spirit *Flex* [41] and *Bison* [9]. Also generators for the semantic evaluation are well known like the AG generator available in the compiler construction suite *Eli* [14, 25]. Figure 2.1 illustrates the classical organization of compilers along compilation phases and the techniques, as well as their specification formalisms used to generate compiler components for each phase.

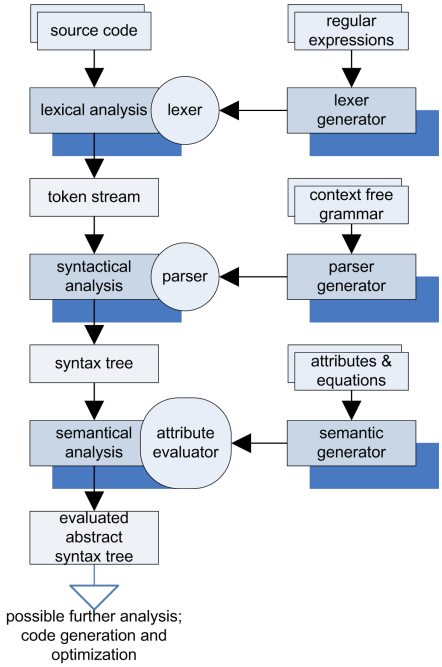

Figure 2.1: Classical organization for the generation of compiler parts for different compilation phases.

The *JastAddC* implementation is based on the classical organization structure in a lexical, syntactical and semantical analysis. The lexical analysis reads the program input and generates a stream of token, which is the input for the syntactical analysis. The syntactical analysis in return constructs an initial AST, which is the input for the semantical analysis. The semantical analysis finally provides functions to request information about the AST. If such an information is requested the AST is evaluated by the semantical evaluator to retrieve the needed information, e.g. to find the declaration of an identifier. Each compilation phase's component is generated using the well known specification techniques presented in figure 2.1. But especially for the specification of the compiler's semantics advanced AG techniques are used. The theoretical concepts used to generate each compilation-phase component are:

- Lexical Analysis: DFA, *principle of the longest match* and *lexer states*
- Syntactical Analysis: LALR(k), with $k = 1$ and *semantic actions* constructing the AST
- Semantical Analysis: ReCRAG, aspect oriented programming (AOP) and the weaving of semantic concerns

The advantage of splitting the compiler in these phases is the refinement of information, so the recognition of some program information can be specified using the most appropriate specification formalism. As described above, the refined information is the input for the next phase, which itself can be specified using a proper formalism. From the literature[4] it is known that the three specification formalisms DFA, LALR(k) and AG are very well suited to specify a compiler in a comprehensible, "natural" way. Also their interaction confirms the requirements stated above. Only the additional objective to support extensibility on the specification level cannot be satisfied for the lexical and syntactical analysis. It is enough to recall the closure proofs known for deterministic CFGs [44]. From the proofs follows, that a LALR(k) parser generator cannot support the needed extension abilities. The problem will be explained in more detail in chapter 3.3, as well as the modularization problems for the current lexer implementation. However, in chapter 7.1 a solution will be suggested.

In contrast to the lexical and syntactical concepts applied in *JastAddC*, ReCRAGs are not well known. ReCRAGs and aspect oriented programming (AOP) together provide the functionality to meet all the objectives and requirements of chapter 2.1, especially the extensibility at the specification level. Chapter 3 will present the modularization in detail. Essentially, AOP is used to weave the semantics specified in different specifications together. The basic structure the semantics are woven in is the AST. The basic contributions woven into the AST are attributes, attribute equations, non-terminal nodes, non-terminal node equations and rewrites. Since an AST can be specified by a CFG and CFGs are closed under unification[5], the AST itself can be extended. Thus two *aspect languages* with a *point-cut language* and *intertype declarations*[6] exist. One to weave semantic specifications, i.e. attributes, equations and rewrites and one to weave grammar rules and node types. The ReCRAG formalism on the other hand is not a requirement to support extensibility. The standard AG formalisms, especially *inherited* attributes are sufficient. But the extensions of ReCRAGs compared to AGs permit much smaller, easier to implement, better understandable and maintainable specifications (see [11] and [36]). ReCRAGs extend AGs by the introduction of

- *Rewrites*: Input ASTs may be rewritten depending on attributes. Rewrites may be conditional or unconditional. A conditional rewrite's condition may depend on attributes itself.

- *Circularity*: Attribute dependencies may be circular. Circular attribute dependencies are evaluated using *fixpoint semantics*. The values of circular attributes are well defined if the values for each attribute on a cycle can be organized in a lattice of finite height[7] and if all the semantic functions involved in computing these attributes are monotonic on the respective lattices [36].

[4]For DFAs see [2, 1]. For LALR(k) see [1, 44]. For AGs see [42, 11].

[5]For a proof see [2].

[6]For details about the concept of AOP and *point-cut languages* as well as *intertype-declarations* see [27].

[7]Thus the values for each attribute in the cycle are elements of a complete partial order with a finite number of elements.

- *Reference Attributes*: Attributes may return remote AST nodes, which have their own attributes. This way attributes of remote nodes can be used to specify the value of a local attribute. To do so a local attribute referencing a remote node can be called to retrieve the remote node, which can then be called for its local attribute.

Additionally, *collection* and *non-terminal attributes* can be introduced. A collection attribute's value is a set of results. Remote nodes can contribute results to a *collection attribute*. The contributions can be guarded by conditions, so a remote node only contributes to a *collection attribute* if the condition of the remote node's contribution is met. A *non-terminal attribute's* equation evaluates to an AST node. Thus the AST is extended based on computations[8] on itself. *Non-terminal* attributes are also known as *higher order attributes* [7, 48]. ReCRAGs extended with *collection* and *non-terminal attributes* can be implemented in a *demand driven* way, so rewrites and attributes are only evaluated if needed.

The concept to refine information, such that at each point during compiler execution the optimal set of information is available and can be handled by an appropriate specification, can even be applied in a compilation phase itself. Rewrites play a critical role to refine the AST. A semantical module may refine the AST by rewriting its structure, such that another module can be implemented based on the refined AST. E.g. the mapping technique presented in chapter 5.3, which transforms the first AST constructed by the parser to a much better AST, which is the basis for all the other semantic implementations, is based on the rewrite functionality of ReCRAGs.

2.3 Applied Tools - Introduction to *JFlex*, *Beaver* and *JastAdd*

In this section the tools used to generate the *JastAddC* compiler will be shortly presented. To emphasize tool interaction the presentation will start with the semantical analysis tool *JastAdd* followed by the syntactical analysis tool *Beaver* and the lexical analysis tool *JFlex*. So it becomes clear, which output is expected to be produced by each tool, because the tool consuming the output has already been presented.

The following tool presentation only gives a short overview. For details the documentation of the appropriate tool has to be consulted. The final output of the *JastAddC* front-end is an evaluated AST[9].

The *JastAdd* system is explained in [10], [11] and [36]. [10] is a good overview and introduction into *JastAdd*. [11] presents the *JastAdd* system and its application to implement an extensible *Java* compiler [13, 35], the *JastAddJ* compiler. Additionally, the implementation and advantages of rewriteable AGs are investigated. [36] presents applications for *referenced*, *circular* and *collection attributes*. The implementation and evaluation of *circular attributes* and *collection attributes* is discussed in-depth. All techniques presented in [36] are investigated with respect to the *JastAdd* system.

[8] Represented by attributes

[9] Hence, the input program's static semantics are computed.

For *Beaver* no printed manual exits, but an online documentation is available [6]. *Beaver* can be considered as a very fast *Yacc* [24]/*Bison* [9] implementation in *Java*.

For details about *JFlex*, which is a *Flex* [41] implementation in *Java*, the user manual [29] as well as the online documentation [28] can be consulted.

2.3.1 *JastAdd*

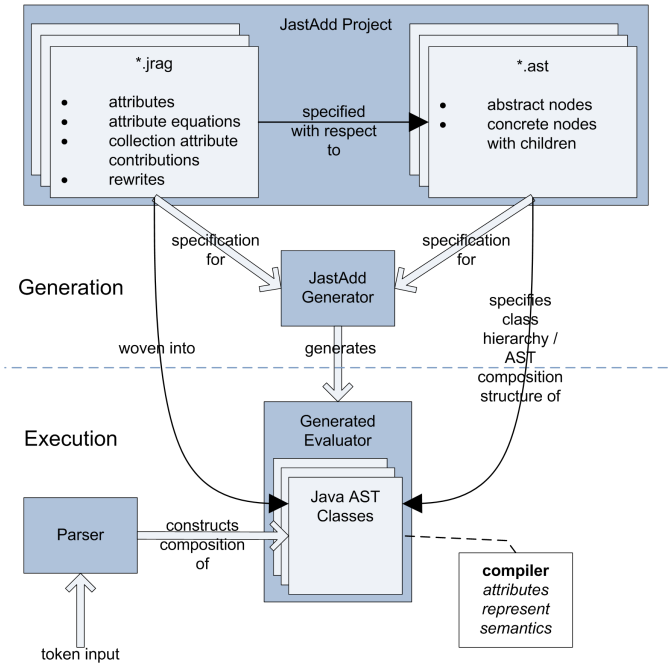

Figure 2.2: Organization of *JastAdd* projects and the interaction between specifications, the *JastAdd* generator, generated semantic evaluator and parser.

The *JastAdd* system is a ReCRAG evaluator generator with support for *non-terminal* and *collection attributes*. Figure 2.2 illustrates the structure of *JastAdd* projects. From the general AG formalism follows, that an AG tool must support two kinds of specifications: One specification language to describe valid ASTs[10] and one specification language to associate attributes with AST nodes and to specify the attribute equations. In *JastAdd* the AST expected as input from the parser is specified by an arbitrary number of `*.ast` specifications. The attributes and their equations are specified by an arbitrary number of `*.jrag` specifications.

First, the *JastAdd* generator reads all the `*.ast` specifications and generates a *Java* class hierarchy, which implements the *composite* design pattern [12]. The classes of the generated hierarchy represent the node types of the AST[11]. Each AST node is typed and knows all its child nodes as well as terminals, it represents

[10] CFGs are only one possible formalism to specify the structure of syntax trees representing words of a context-free language.

[11] Consequently the terms AST class and AST node type can be exchanged mutually.

a context[12]. Consequently, object compositions of these *Java* classes represent ASTs or AST parts, depending whether the object composition's root node is the root node of the abstract syntax or not. It is the parser's task to construct such object compositions, thus to construct the ASTs.

In the second generation step, after generating the AST class hierarchy, the *JastAdd* generator weaves all the semantics into the AST classes. To do so it reads all the `*.jrag` specifications and generates for each attribute `public` *Java* methods which are named like the attribute. These methods return the value of the attribute they represent and are woven into the AST classes the attribute is associated with. Thus, an attribute method implementation realizes the attribute's equation specified for the context represented by the AST class the method is woven into. Hence, all AST classes, which inherit from another AST class also inherit all its semantics[13]. This general approach to represent ASTs and their attributation using an object-oriented class hierarchy, where syntactical contexts are classes and attributes are methods is well known and has been introduced by [17]. Since the *JastAdd* system only accepts valid specifications and in a valid specification the attribute values must be defined for each possible context, it is guaranteed that each attribute access method is defined and returns the correct attribute value. Since each generated attribute access method will only compute its attribute value if called *JastAdd* evaluators are *demand-driven*. Even with the ReCRAG, *non-terminal* and *collection attribute* extensions the *demand driven* approach is satisfied: Only required computations are executed. Also the correct evaluation order is automatically realized, especially rewrites are executed as soon as possible, if the rewritten node is accessed at all.

After weaving the attributes and equations into the generated AST classes the *JastAdd* system is done. The generated evaluator is the attributed AST itself! The application programming interface (API) methods are the attribute access methods as well as the AST structure access methods[14]. Hence, it is no problem to retrieve AST information.

2.3.2 *Beaver*

Beaver is a LALR(k) for $k = 1$ parser generator implementation in *Java* and very similar to *Yacc* [24] and *Bison* [9]. Figure 2.3 illustrates the structure of *Beaver* projects. The *Beaver* parser generator expects exactly one specification and generates one *Java* class implementing the parser. The parser's input are token objects, which must inherit from *Beaver's* abstract `beaver.Symbol` class. A parser generated with *Beaver* expects a lexer. The lexer must inherit from the

[12]Contexts represented by *JastAdd* AST classes are similar to the context term of CFGs. Each rule of a CFG represents a context. In *JastAdd* a CFG rule can be regarded as a node type with the rule's production as child nodes. Hence, an AST node type represents a context. The context term is essential for AGs since attribute equations are specified with respect to a context and for each context in which an attribute can occur an equation for the attribute must be defined.

[13]Of course, the inheritance relationship is transitive. If an equation for an attribute is given in an AST class A, from which the AST class B inherits and there is no equation specified for the attribute in the context B, B simply inherits the equation of A. But if an equation for B is given, the attribute access method is implemented again in B with the given equation, whereby the access method of A is overridden.

[14]The JastAdd system automatically generates methods to travers and access compositions of AST class objects (hence, ASTs).

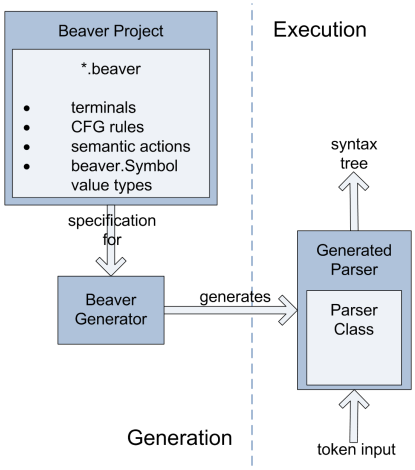

Figure 2.3: Organization of *Beaver* projects and the interaction between specification, *Beaver* generator and generated parser.

abstract class `beaver.Scanner` and implement the `nextToken()` method. Each time the `nextToken()` method is called it returns the next `beaver.Symbol` recognized by the lexer, signals the end of the input program (`EOF` token) or throws an exception[15] if the next input characters cannot represent any valid token. A *Beaver* specification specifies at least a set of tokens, a CFG and *semantic actions* associated with grammar rules. Other supporting constructs can be specified as well, to configure the generated parser and to embed user written *Java* code into the parser.

A *Beaver* symbol has a line and a column of type `int`, an id of type `short` and a value of type `Object`. The symbols do not only represent tokens, but also *non-terminals*. The distinction if a symbol object represents a token or *non-terminal* is done implicitly during the symbol's instantiation. Each symbol representing a token will have an unique id[16]. The token ids are represented as `public static final short` attributes in the automatically generated inner class `Terminals`, which is part of the generated parser class. The token ids available in the generated parser class can be used by a lexer to instantiate appropriate tokens during the lexical analysis. A special `EOF` id is generated as well. The `EOF` token tells *Beaver* that no further input is available.

Semantic actions can be used to compute any value based on information provided by the production of the rule the *semantic action* is associated with. Each *semantic action* must return a *Beaver* symbol. A value computed by a *semantic action* can be stored in the value attribute of the symbol returned. Each time the parser reduces with some grammar rule or shifts a token the symbol representing the reduced rule's left side *non-terminal* or the shifted token is pushed on *Beaver's* stack. Consequently, a successor reduction, which reduces the symbol on the stack, can access the symbol and its value. So, a successor

[15] `java.io.IOException` or `beaver.Scanner.Exception`

[16] If the `public Symbol(Object value)` constructor is called the symbol represents a *non-terminal*. If a constructor with an id as parameter is called the symbol represents the token associated with the id.

semantic action can reuse values computed by preceding *semantic actions*. This behavior is similar to a synthesized attribute, whose value also depends on the value of attributes associated with the child nodes of the node the synthesized attribute is associated with. Hence, it is no problem to construct a syntax tree in a *bottom-up* manner using *semantic actions*.

2.3.3 *JFlex*

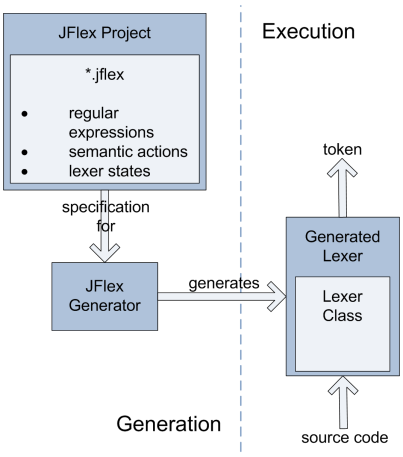

Figure 2.4: Organization of *JFlex* projects and the interaction between specification, *JFlex* generator and generated lexer.

JFlex is a *Flex* [41] implementation in *Java*. Figure 2.4 illustrates the structure of *JFlex* projects. The *JFlex* lexer-generator expects exactly one specification and generates a single *Java* class, which is the specified lexer's implementation. The generated lexer expects input programs represented as *Java* `java.io.InputStream` or `java.io.Reader` objects. Each call of the scanning method[17] returns the next token of the input program if possible. If the end of the input program is reached the special *EOF* token is returned. If the next characters of the input program cannot be matched an exception is thrown.

Tokens are specified by *lexical rules*, which are based on the well known concepts of *regular expressions*, the *principle of the longest match*, token precedence rules, *lexer states* and *semantic actions*.

Each *lexical rule* specifies one token and consists of a *regular expression* and a *semantic action*. A *regular expression* describes a set of input character sequences. Each time the token is recognized, the input character sequence representing the token is consumed. The consumption of input characters, which represent a token is also called to match the token. A *semantic action* describes the instructions to execute if a token is matched. The executed instructions must return a token, i.e. a value of the type representing tokens. Hence, it is the *semantic action's* task to construct the appropriate token after a match.

[17] Named `yylex()` by default

If a character sequence can represent several tokens it represents the token specified first. Thus, an implicit token precedence is defined by the order in which tokens are specified.

Additionally, the *principle of the longest match* is applied. Thus, characters are consumed as long as the current character sequence represents any valid token. As soon as a character c is encountered, whose concatenation to the current character sequence would result in a sequence not representing any token at all, the last token recognized is returned and c is not consumed yet. Hence if w with $|w| > 1$ is the token T_1 and wa with $|a| = 1$ is the token T_2 and wab with $|b| = 1$ is no token the recognized token for the input wab is T_2, which consumes the input wa.

Lexer states can be used to impose a regular automaton (DFA) on top of the lexer's basic DFA[18]. A state of the imposed automaton (thus a *lexer state*) consists of a subset of the lexer's tokens. The lexer is only permitted to recognize tokens contained in the lexer's current state. Transitions of the imposed DFA represent a change of the *lexer state*. Hence, transitions represent a change of the tokens currently recognizable. Transitions are specified with respect to recognized tokens. Hence, *lexer states* are changed in *semantic actions*. A token may be associated with an arbitrary number of *lexer states*, so the same token might be recognized in different *lexer states*. Tokens can be specified without an explicit association to any *lexer state*. To be able to match such tokens, *inclusive* and *exclusive lexer states* are distinguished. *Inclusive lexer states* also include all the tokens not associated with any state, whereas *exclusive lexer states* do not. If the lexer is in an *exclusive lexer state* it can only return a token, which is explicitly associated with the *exclusive lexer state*.

2.4 Integrating the Tools - Complete Compiler Development based on *JFlex*, *Beaver* and *JastAdd* Specifications

Generating a complete compiler based on *JFlex*, *Beaver* and *JastAdd* specifications is straightforward. Figure 2.5 illustrates the compiler development process and the interaction between generated components and specifications. First, the language's structure and semantics have to be designed and specified. Thus, a *Beaver* specification has to be developed, as well as the *JastAdd* `*.ast` specification for the AST structure, which represents the language's semantics best. Whether to start with the specification of the concrete or abstract syntax depends on the concrete compiler project. After specifying the language's structure the tokens should be clear and an appropriate *JFlex* specification can be developed. Next, the *Beaver* specification has to be completed, so *JastAdd* ASTs are constructed. Thus, the *semantic actions* of the *Beaver* specification have to be implemented. Finally, the semantics can be specified by augmenting the AST with attributes. To do so *JastAdd* `*.jrag` specifications are developed. A good separation of concerns has to be found, so the language's semantics can be split in independent aspects (see chapter 3). In-between these steps, the compiler's *main class*, which composes all the compiler components together,

[18]The lexer's basic DFA is the automaton accepting all the tokens specified and discarding any other input.

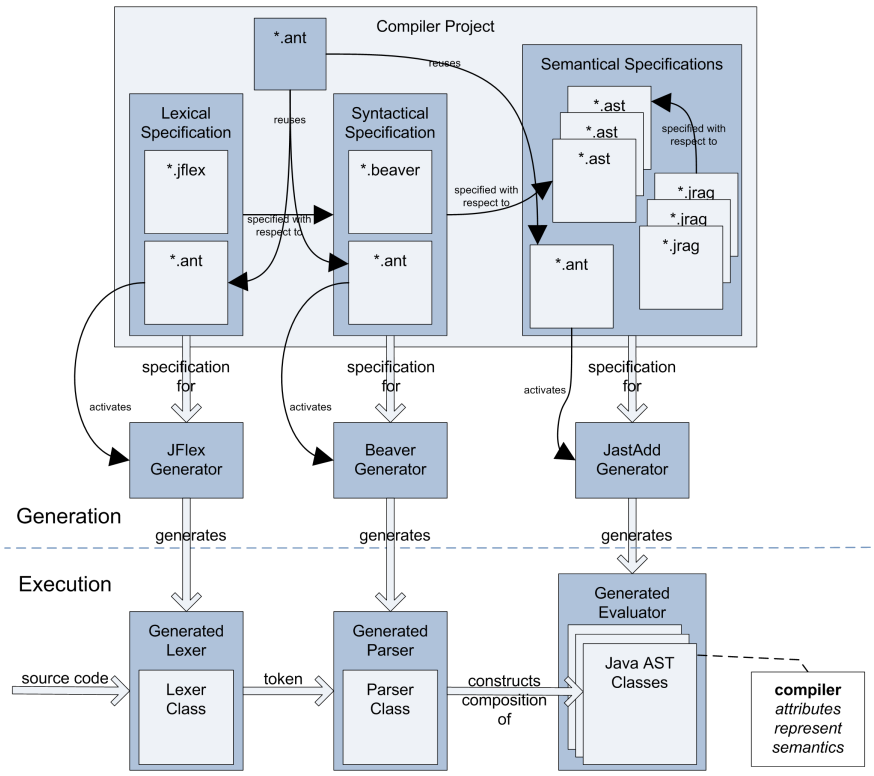

Figure 2.5: Complete compiler project structure for *JastAdd*, *Beaver* and *JFlex* and the interaction between specifications, generators and generated lexer, parser and semantic evaluator. The `*.ant` files are Ant [15] build scripts which are used to organize the automatically generation of compiler components.

i.e starts the compilation and returns the result (the evaluated AST), has to be manually implemented. Listing 2.1 represents a generic *main class* skeleton composing all the generated compiler components.

Listing 2.1: Generic *glue-code main class* skeleton, which composes a final compiler based on a *JFlex* lexer, *Beaver* parser and *JastAdd* semantic evaluator.

```
1  package compiler;
2
3  import java.io.*;
4
5  import compiler.symbols.*;
6  import compiler.syntax.*;
7  import compiler.semantics.*;
8
9  /**
10  * Glue code class composing all generated compiler parts together.
11  */
12 public abstract class Compiler {
13     /**
14      * Performs all the compilation actions needed to construct
```

```
15      * an evaluated AST.
16      */
17     public ASTRootNode constructAST(java.io.Reader sourceCode) throws
            IOException, beaver.Scanner.Exception {
18         GeneratedLexerClass lexer = new GeneratedLexerClass(sourceCode);
19         GeneratedParserClass parser = new GeneratedParserClass();
20         return (ASTRootNode)parser.parse(lexer);
21     }
22
23     /**
24      * Perform the actual compilation, thus access the AST's
25      * attributes to retrieve information and generate the
26      * compiler's output.
27      */
28     public abstract void performAPICompilerActions(ASTRootNode ast);
29
30     /**
31      * Executes the compiler from the command line. The argument
32      * given is the source code file to compile.
33      */
34     public static void main(String[] args) throws IOException, beaver.Scanner.
            Exception {
35         if (args.length < 1)
36             throw new RuntimeException("No source code file specified.");
37         File sourceCode = new File(args[0]);
38
39         ASTRootNode ast = constructAST(new FileReader(sourceCode));
40         performAPICompilerActions(ast);
41     }
42 }
```

The skeleton *main class* is abstract, because the AST's API functionalities[19] used by the compiler differ from compiler to compiler. However, there are common compiler tasks such as a check of the input program for semantic errors or the code generation, which can be realized/triggered by root node attributes. In listing 2.1 the generated lexer class is `compiler.symbols.GeneratedLexerClass`, the generated parser class is `compiler.symbols.GeneratedParserClass` and the AST classes are in package `compiler.semantics`. The AST root node class is `ASTRootNode`.

To organize and automate the generation of the compiler components *Ant* [15] *build scripts* can be used. The specifications can be modularized across the lexical, syntactical and semantical compilation phases. So for each of the generators *JFlex*, *Beaver* and *JastAdd* a *build script* exists, which generates or deletes all the compiler components of the appropriate tool. To construct or delete all the components at once another *build script* can be implemented, which reuses and executes all the tool specific *build scripts*. The destination for the generated compiler components is the compiler implementation, a *Java* source code package hierarchy. Hence the specification's hierarchy and dependencies between specifications are independent of the generated compiler implementation's hier-

[19] The AST's attributation.

archy and architecture. To pack the compiler implementation together into a distribution another *Ant build script* can be used.

Most parts of the compiler implementation are generated. Only *glue code* classes[20], which connect compiler components, and simple wrapper classes, which represent a compilation specific information, have to be handcrafted. Examples for compilation specific information are special tokens or results of semantic evaluations. It is important that the *Ant build scripts* do not manipulate the hand-crafted classes when performing changes on the compiler implementation. If all generated artefacts are deleted the implementation packages should only contain **all** the hand-crafted *glue code* and wrapper classes.

The development of the *JastAddC* compiler is based on the rough concept just described.

2.5 Example Compiler Project

Before the *JastAddC* implementation is presented in detail this section will illustrate the *specification-driven* compiler development. A small example compiler project will be developed based on *JastAdd*, *Beaver* and *JFlex* specifications.

The developed example is a small calculator for natural numbers. The calculator supports increment, decrement, addition, subtraction, multiplication, division and the access to numbers stored in memory by name. Listing 2.2 presents the calculator's syntax, tokens and operators.

Listing 2.2: Calculator compiler: Language syntax.

```
1  calculator = memory expression;
2
3  memory = "#" identifier "=" number memory |
4      ;
5
6  expression =
7      identifier |
8      constant |
9      unary_expression |
10     binary_expression
11     ;
12
13 unary_expression =
14     expression "++" |
15     expression "--" |
16     "(" expression ")"
17     ;
18
19 binary_expression =
20     expression "+" expression |
21     expression "-" expression |
22     expression "*" expression |
23     expression "/" expression
24     ;
```

[20] Like the compiler's *main class*.

The calculator's syntax presented in listing 2.2 is ambiguous, e.g. several derivations for the input programs $1 + a/(7 - b)$ exist. Of course, only the $(1 + (a/(7 - b)))$ interpretation is correct.

The calculator's implementation will be presented as follows: First the AST with the root node `calculator.Calculator` and the calculator's semantics are developed. Afterwards, the parser named `calculator.CalculatorParser`, which constructs the specified AST will be implemented. Finally, the lexer named `calculator.CalculatorLexer` is realized. The output computed by the calculator compiler is the evaluated expression, a natural number printed on the screen.

Listing 2.3 presents the calculator's *main class*. It is based on the *main class* skeleton of listing 2.1.

Listing 2.3: Calculator compiler: *Main class*.

```
package claculator;

import java.io.*;

import calculator.*;

/**
 * The final calculator compiler composing all generated
 * calculator parts together.
 */
public class CalculatorCompiler {
    /**
     * Construct a calculator AST.
     */
    public Calculator constructAST(java.io.Reader sourceCode) throws
        IOException, beaver.Scanner.Exception {
        CalculatorLexer lexer = new CalculatorLexer(sourceCode);
        CalculatorParser parser = new CalculatorParser();
        return (Calculator)parser.parse(lexer);
    }

    /**
     * Perform the actual compilation, thus the computation
     * of the expression's value.
     */
    public void performAPICompilerActions(Calculator ast) {
        System.out.println(ast.Value());
    }

    /**
     * Execute the calculator from the command line. The argument
     * given is the source code file representing an expression to evaluate.
     */
    public static void main(String[] args) throws IOException, beaver.Scanner.
        Exception {
        if (args.length < 1)
            throw new RuntimeException("No source code file specified.");
        File sourceCode = new File(args[0]);

```

```
38          Calculator ast = constructAST(new FileReader(sourceCode));
39          performAPICompilerActions(ast);
40      }
41  }
```

The `Calculator` root node and each `Expression` node have an attribute `Value()`, which computes the input expression's value. Hence, the `performAPICompilerActions` method can just call the `Value()` attribute of the root node and print the result on the screen.

The specification of the AST's structure is given in listing 2.4. The `*.ast` specification is similar to the informal language definition given in listing 2.4. Hence, also the AST is ambiguous. However, since the parser implementation presented later on is unambiguous, always the correct AST will be constructed.

Listing 2.4: Calculator compiler: `*.ast` specification.

```
1  Calculator ::= Memory* Expression;
2
3  Memory ::= <Variable:String> <Value:int>;
4
5  abstract Expression;
6
7      Variable:Expression ::= <Identifier:String>;
8      Constant:Expression ::= <Value:int>;
9
10 abstract UnaryExpression:Expression ::= Operand:Expression;
11
12     Increment:UnaryExpression;
13     Decrement:UnaryExpression;
14
15 abstract BinaryExpression:Expression ::= Operand1:Expression Operand2:
       Expression;
16
17     Addition:BinaryExpression;
18     Substraction:BinaryExpression;
19     Multiplication:BinaryExpression;
20     Division:BinaryExpression;
```

The AST class hierarchy generated for the `*.ast` specification is presented in figure 2.6. The AST root node is `Calculator`. Its context is represented by two child nodes: A list of arbitrary many `Memory` nodes and an `Expression` node, which represents the actual expression to evaluate. Each `Memory` node associates a natural number with a variable, whereas the variable is represented by a terminal of type `String` and the natural number by a terminal of type `int`. The `Expression` node is *abstract*, since several expression contexts exist. In fact each of the node types `Variable`, `Constant`, `UnaryExpression` and `BinaryExpression` are `Expression` nodes. The `UnaryExpression` node type extends the `Expression` node type's context with an additional `Expression` child named `Operand`. The `BinaryExpression` extends the `Expression` node type's context with two `Expression` child nodes named `Operand1` and `Operand2`. The `Variable` and `Constant` node types extend the `Expression` node type's context with a terminal child node. For the `Variable` node type the terminal

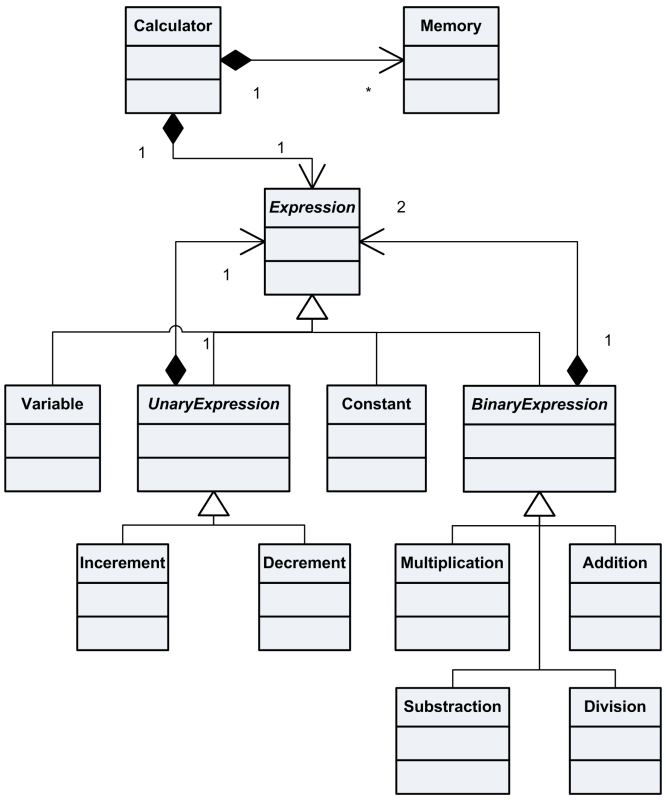

Figure 2.6: Calculator compiler: AST class hierarchy generated for the claculator's `*.ast` specification given in listing 2.4.

child is named `Identifier` and its type is `String`. For the `Constant` node type it is named `Value` and of type `int`. `UnaryExpression` and `BinaryExpression` are *abstract* node types themselves. Their refined contexts are `Increment`, `Decrement` and `Addition`, `Substraction`, `Multiplication` and `Division` respectively. Of course, the inheritance relationship is transitive. Hence, each of the refined `UnaryExpression` and `BinaryExpression` contexts is an `Expression` and inherits all the semantics `Expression` has. But an `UnaryExpression` is no `BinaryExpression` and vice versa. Whenever an `Expression` node is expected, any of the node types specified can occur. If an `UnaryExpression` node is expected an `Increment` or `Decrement` expression can be given. Analogously a `BinaryExpression` node is represented by any of the node types `Addition`, `Substraction`, `Multiplication` and `Division`.

To become familiar with the calculator's AST structure, consider listing 2.5. It presents *Java* code constructing an AST for the calculator expression $1 + + + a/(7 - b)$.

Listing 2.5: Calculator compiler: *Java* code constructing an AST for the expression $1 + + + a/(7 - b)$.

```
1 new Calculator(new List() /* There are no values in memory */,
2         new Addition(
3                 new Increment(
```

```
4                       new Constant(1)),
5                 new Division(
6                       new Variable("a"),
7                       new Substraction(
8                             new Constant(7),
9                             new Variable("b")))))
```

The child nodes of each AST node are given as constructor arguments.

After specifying the AST structure, the semantics have to be defined. As mentioned above, the `Value()` attribute is used to compute an expression's value. Listing 2.6 declares the `Value()` attribute and specifies its equations.

Listing 2.6: Calculator compiler: `*.jrag` specification for the evaluation of expressions.

```
1  aspect ExpressionValues {
2      // The calculator's result simply is the value
3      // of its expression:
4      syn int Calcuator.Value() = getExpression().Value();
5      // Each Expression node has the parameterless
6      // attribute Value of type int:
7      syn int Expression.Value();
8
9      eq Constant.Value() = getValue();
10     eq Variable.Value() = LookUp(getIdentifier());
11     eq Increment.Value() = getOperand().Value() + 1;
12     eq Decrement.Value() = getOperand().Value() − 1;
13     eq Addition.Value() =
14         getOperand1().Value() + getOperand2().Value();
15     eq Substraction.Value() =
16         getOperand1().Value() − getOperand2().Value();
17     eq Multiplication.Value() =
18         getOperand1().Value() ∗ getOperand2().Value();
19     eq Division.Value() =
20         getOperand1().Value() / getOperand2().Value();
21 }
```

It is obvious, that an expression's value only depends on the operation represented by the expression and the value of its child nodes. Hence, `Value()` is a *synthesized attribute*. Its equations are straightforward. The value of an `Increment` expression is the value of the `Operand` child plus one. The value of a `Multiplication` is the value of its first operand multiplied with the value of its second operand. The equations for the other contexts are analogous. Only the equation for `Variable` expressions is more complicated, since the value stored for the variable must be accessed. Since an `Expression`'s subtree does not contain the declarations of its variables[21], the declaration information of variables has to be distributed/broadcasted into `Expression` trees. To do so an *inherited attribute* `LookUp(String name)` is used. It takes a variable name as argument and returns the value stored for it. Listing 2.7 presents the implementation of the calculator's simple name analysis.

[21] Each variable declaration is represented by one `Memory` node.

Listing 2.7: Calculator compiler: `*.jrag` specification for the name analysis.

```
aspect NameAnalysis {
    // Variables can be looked up in memory
    // at each Expression node:
    inh int Expression.LookUp(String name);

    // Broadcast variable declarations into Expression trees:
    eq Calculator.getExpression().LookUp(String name) {
        // Traverse the memory...
        for (int i = 0; i < getNumMemory(); i++)
            // for some variable...
            if (getMemory(i).getVariable().equals(name))
                // and if found return its value...
                return getMemory(i).getValue();
        return 0; // and otherwise zero.
    }
}
```

As known from the AG formalism, the equations of *inherited attributes* are specified with respect to the parent node of the node the *inherited attribute* is associated with. However, since *JastAdd* automatically broadcasts an *inherited attribute's* value into the subtree spaned by the node for which an equation for the attribute is given, it is enough to specify `LookUp` only for the `Calculator` root node. Of course, a broadcasted attribute's value can be refined by contexts with an equation for the attribute, which are further down in the spaned tree. For convenient reasons the value of undeclared variables is specified to be zero [22].

The next development step is to specify a parser, which constructs the calculator's ASTs. Listing 2.8 specifies an appropriate *Beaver* parser.

Listing 2.8: Calculator compiler: *Beaver* specification.

```
%header{:
/**
 * Beaver generated parser for the calculator compiler.
 */
:};

%package "calculator";
%class "CalculatorParser";
%import "calculator.*";

%terminals CONSTANT, IDENTIFIER, INC, DEC, PLUS, MINUS, MUL, DIV,
    BRACKET_OPEN, BRACKET_CLOSE, EQUAL, SHARP;
%typeof IDENTIFIER = "String";
%typeof CONSTANT = "int";
%typeof calculator = "Calculator";
%typeof memory = "List"; // List of Memory nodes
%typeof expression = "Expression";
%typeof multiplicative_expression = "Expression";
%typeof unary_expression = "Expression";
%typeof primary_expression = "Expression";
```

[22] To save the overhead of proper error handling.

```
20
21 %goal calculator;
22
23 calculator = memory expression;
24
25 memory = SHARP IDENTIFIER.id EQUAL CONSTANT.con memory.list
26         {:
27             list .add(new Memory(id, con));
28             /*
29              * Return the symbol containing list as value: The reference
30              * list references a list of Memory nodes. The reference
31              * _symbol_list references the symbol containing list as value.
32              */
33             return _symbol_list;
34         :}
35     |
36         {:
37             return new Symbol(new List());
38         :}
39     ;
40
41 expression = multiplicative_expression.expr
42         {:
43             return _symbol_expr;
44         :}
45     | expression.operand1 PLUS expression.operand2
46         {:
47             return new Symbol(new Addition(operand1, operand2));
48         :}
49     | expression.operand1 MINUS expression.operand2
50         {:
51             return new Symbol(new Substraction(operand1, operand2));
52         :}
53     ;
54
55 multiplicative_expression = unary_expression.expr
56         {:
57             return _symbol_expr;
58         :}
59     | multiplicative_expression.operand1 MUL unary_expression.operand2
60         {:
61             return new Symbol(new Multiplication(operand1, operand2));
62         :}
63     | multiplicative_expression.operand1 DIV unary_expression.operand2
64         {:
65             return new Symbol(new Division(operand1, operand2));
66         :}
67     ;
68
69 unary_expression = primary_expression.expr
70         {:
71             return _symbol_expr;
72         :}
73     | expression.expr INC
```

```
74          {:
75              return new Symbol(new Increment(expr));
76          :}
77      | expression.expr DEC
78          {:
79              return new Symbol(new Decrement(expr));
80          :}
81      ;
82
83  primary_expression = IDENTIFIER.id
84          {:
85              return new Symbol(new Variable(id));
86          :}
87      | CONSTANT.con
88          {:
89              return new Symbol(new Constant(con));
90          :}
91      | BRACKET_OPEN expression.expr BRACKET_CLOSE
92          {:
93              return _symbol_expr;
94          :}
95      ;
```

The `%package` and `%class` keywords are used to specify the fully qualified *Java* name of the generated parser. To reuse the already generated AST classes, they have to be imported by the `%import calculator.*;` directive. The `%terminals` keyword is used to declare the tokens the lexer must match.

For each CFG rule a *semantic action* is embedded between `{:` and `:}`, which constructs the AST for the input reduced by the rule. Hence, if parsing has been successful the symbol returned from the last reduction executed contains the AST.

Often it is necessary to access AST parts constructed in preceding parsing steps to construct the current reduction's AST. Named references can be declared to access a symbol in *semantic actions*, which has been processed in preceding parsing steps and is contained in the current applied rule's production. By default the references point to the *Beaver* symbol on the stack, which represents the child node. Hence, to access the actual value computed (the AST constructed for the symbol) two accesses must be done: First the symbol is accessed and afterwards its value. To ease the access of values, types can be associated with symbols. If a symbol has an associated type, a reference to the symbol accesses the value of the symbol instead the symbol itself. For the calculator compiler all AST nodes are expressions, except the `Calculator` and `Memory` nodes. The `%typeof` directive is used to specify the types of symbols. Sometimes the correct AST for a rule already has been constructed. In such cases simply the symbol already containing the correct AST can be returned instead of constructing another symbol, e.g. in the `expression = multiplicative_expression.expr` rule's *semantic action* the symbol constructed for the `multiplicative_expression` child is returned.

As it can be seen in the listing, the construction of ASTs is straightforward and similar to the static instantiation of ASTs as presented in listing 2.5.

Finally the lexer for the calculator can be specified. Listing 2.9 represents the appropriate *JFlex* specification.

Listing 2.9: Calculator compiler: *JFlex* specification.

```
/**
 * JFlex lexer for the calculator compiler.
 * The lexer constructs tokens for the claculator.ClaculatorParser
 * Beaver parser.
 */

package calculator;

import claculator.CalculatorParser;
import claculator.CalculatorParser.Terminals;

%%

%class CalculatorLexer
%extends beaver.Scanner
%function nextToken
%yylexthrow beaver.Scanner.Exception
%type beaver.Symbol
%eofval{
    return new beaver.Symbol(Terminals.EOF, yyline, yycolumn, yylength(),
        yytext());
%eofval}

%eofclose
%unicode
%line
%column

LineTerminator = \r|\n|\r\n|\u2028|\u2029|\u000B|\u000C|\u0085
WhiteSpace = {LineTerminator} | [ \t\f\u000B]

LatinLetter = [a-z]|[A-Z]
DecimalDigit = [0-9]

%%

<YYINITIAL> {
    // CONSTANT:
    DecimalDigit+ {
        return new beaver.Symbol(
                Terminals.CONSTANT,
                yyline, yycolumn,
                yylength(),
                Integer.parseInt(yytext()));
    }

    // IDENTIFIER:
    ({LatinLetter} | "_")({LatinLetter} | "_" | {DecimalDigit})* {
        return new beaver.Symbol(Terminals.IDENTIFIER, yyline, yycolumn,
            yylength(), yytext());
```

```
    }

    // INC:
    "++" {
        return new beaver.Symbol(Terminals.INC, yyline, yycolumn, yylength(),
            yytext());
    }

    // DEC:
    "--" {
        return new beaver.Symbol(Terminals.DEC, yyline, yycolumn, yylength(),
            yytext());
    }

    // PLUS:
    "+" {
        return new beaver.Symbol(Terminals.PLUS, yyline, yycolumn, yylength(),
            yytext());
    }

    // MINUS:
    "-" {
        return new beaver.Symbol(Terminals.MINUS, yyline, yycolumn, yylength
            (), yytext());
    }

    // MUL:
    "*" {
        return new beaver.Symbol(Terminals.MUL, yyline, yycolumn, yylength(),
            yytext());
    }

    // DIV:
    "/" {
        return new beaver.Symbol(Terminals.DIV, yyline, yycolumn, yylength(),
            yytext());
    }

    // BRACKET_OPEN:
    "(" {
        return new beaver.Symbol(Terminals.BRACKET_OPEN, yyline,
            yycolumn, yylength(), yytext());
    }

    // BRACKET_CLOSE:
    ")" {
        return new beaver.Symbol(Terminals.BRACKET_CLOSE, yyline,
            yycolumn, yylength(), yytext());
    }

    // EQUAL:
    "=" {
        return new beaver.Symbol(Terminals.EQUAL, yyline, yycolumn, yylength
            (), yytext());
```

```
94      }
95
96      // SHARP:
97      "#" {
98          return new beaver.Symbol(Terminals.SHARP, yyline, yycolumn, yylength
                (), yytext());
99      }
100
101     {WhiteSpace} {
102         // Ignore white space.
103     }
104 }
105
106 // If no valid token can be matched throw an exception for Beaver:
107 .|\n {
108     throw new beaver.Scanner.Exception(yyline, yycolumn, yytext());
109 }
```

To adapt the lexer for the *Beaver* parser it must extend *Beaver's* `beaver.Scanner` class, the scanning method must be named `nextToken()`, the scanning method must throw `beaver.Scanner.Exceptions` and the returned token must inherit from `beaver.Symbol`. The directives `%extends beaver.Scanner`, `%function nextToken`, `%yylexthrow beaver.Scanner.Exception` and `%type beaver.Symbol` configure the generated lexer to satisfy these requirements. Additionally, the `%eofval{...}` directive ensures that the `EOF` token specified by *Beaver* is returned if the end of the input is reached.

To ease the *regular expressions* of *lexical rules* four *macro definitions* are defined: `LineTerminator`, `WhiteSpace`, `LatinLetter` and `DecimalDigit`.

The specification of each *lexical rule's semantic action* is simple. Each time a token is matched, the appropriate *Beaver* id for the token is requested from the `Terminals` class generated by *Beaver*. Also the current line, column, matched input text and its length is accessed using the special *JFlex* attributes and methods `line`, `column`, `yytext()` and `yylength()`. These information are used to instantiate a `beaver.Symbol` object, which represents the token matched. The constructed symbol is simply returned.

The calculator's *main class* presented in listing 2.3 and the specifications of listing 2.4, 2.6, 2.7, 2.8 and 2.9 completely implement the calculator. The implementation of *JastAddC* is similar, only the scale is much greater and instead of just one `*.ast` and two `*.jrag` specifications many are given.

Chapter 3

Compiler Modularization

Chapter 2.4 explained how a complete compiler can be generated based on *JastAdd*, *Beaver* and *JFlex* specifications. However, the thesis' objective is the *specification-driven* development of **extensible** compilers. Hence, the compiler consists of independent modules and can be easily extended with additional functionality. A modularization concept is needed, which permits the automatically composition of specifications based on their specification formalisms (see chapter 2.2). For extensible compiler construction this automatically composition is the *specification-driven* development's main advantage compared to classical handcrafted implementations.

Consequently, the modularization concept of *JastAddC* is at the specification level. Sets of specifications form modules. Hence, an extension is a set of specifications. Since the specifications generate the final compiler, the modularization concept is statical. A generated compiler cannot be extended at run-time. In contrast to the generating specifications, its generated implementation has no well defined components, which support a component model and composition operators. Instead, the specifications representing the extension and the specifications for the existing compiler must be processed together to generate the extended compiler. Hence, opposed to [45], where the composed components are binary ones supporting run-time composition, the components for the *specification-driven* compiler development [1] are statically composed specifications. Although extensions result in changed source code[2], it is not convenient to classify *JastAddC's* modularization concept as the statically composition of source code components, as it is realized by the invasive software composition (ISC) paradigm[3] [3]. The specifications are composed and not the generated source code and they represent another abstraction level above the source code level.

Before the modularization concept satisfying the extension requirements is investigated in detail, the next section presents the classical modularization of

[1] As it is described in this thesis.

[2] Because the source code implementation is generated based on all specifcations, including the ones of extensions.

[3] Of course, the tools processing the specifications can use ISC to generate their outputs. However, the exactly tool implementation does not matter for the compiler developer.

compilers in *compilation-phases*, whereas each phase has an arbitrary number of *passes*.

3.1 Modularization Across *Compilation-Phases*

Figure 3.1 illustrates the general structure for a modularization in *compilation-phases*. The output computed by a *compilation-phase* n is the input for *compilation-*

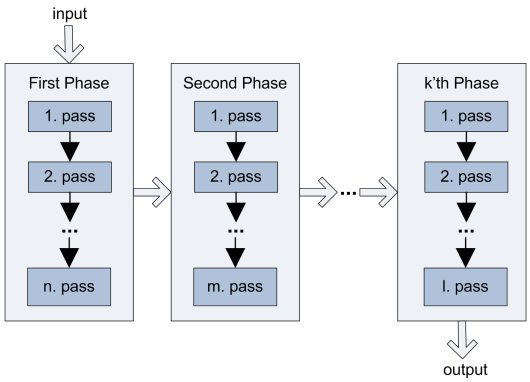

Figure 3.1: Modularization in phases and *passes*.

phase $n + 1$. Each phase may consist of an arbitrary number of *passes*. Each *pass* refines some input information and provides the refined information as input for the next *pass*. Whether the *passes* are nested in an interleaved way[4] or are executed in a sequential order depends on the tasks of the *passes* and performance issues. There exists no formal definition to clarify the difference between a *compilation-phase* and a *pass*. It is a design decision, whether some compilation task becomes its own *compilation-phase* or is simply a *pass* contained in a *compilation-phase*.

If *JastAddC's* specifications are sorted with respect to their specification languages (thus tools) a classical modularization across the compiler's *compilation-phases* results, since each tool generates the compiler components of one compilation phase (compare figures 3.1, 2.1 and 2.5). However, if several specifications for the same tool represent different *passes* or not depends on the tool processing the specifications, the interactions between the specifications and the generated artefacts.

JastAdd is completely *demand-driven* and declarative, thus the resulting evaluator cannot be separated in *passes*. In fact, the automatically organization of the evaluation order is one of *JastAdd's* greatest advantages. The developer has not to bother about the order of specification interactions.

In contrast, each *JFlex* specification generates one lexer. Hence, the design decision to let each *JFlex* specification (thus each generated lexer) represents a *pass* of the lexical *compilation-phase* is self-evident. The lexers generated for

[4] A *pass* P_1 is interleaved with another *pass* P_2, if P_2 pulls P_1 for some information each time P_2 has to compute another of its results. If the *passes* are not interleaved simply the complete *pass* P_1 is executed and afterwards the complete *pass* P_2.

JastAddC are used to refine the analysed input stepwise. Each lexer[5] refines the output of the preceding lexer. The generated lexers can be organized in a *lexer stack* (see chapter 4.1).

For *Beaver* only one specification is given. There exists no construct in *C*, such that a parser stack would be useful.

Since the tools do not know about each other, their composition reflecting the order of the *compilation-phases* must be handcrafted. Fortunately this is quite easy as illustrated in the generic *main-class* skeleton already presented in listing 2.1. In *JastAddC*, the `jastaddc.Compiler` class composes all the *compilation-phases*. Additionally, a similar *glue-code* class, which composes all the *passes* of the lexical *compilation-phase*, is needed, the `jastaddc.symbols.lexicalanalyse.CLexicalAnalyser` class. It is used in the `jastaddc.Compiler` class to represent the complete lexical *compilation-phase*.

It can be said in general: For *compilation-phase* architectures, where each specification represents a *pass*, an extension specification which refines information computed by another specification is often complicated to implement, since a handcrafted embedding of the generated *pass* at the right place may be needed. Hence, the **complete automatically** generation of the extended compiler may not be possible.

It is obvious, that the modularization across *compilation-phases* is not sufficient for the extensible compiler construction, since even it supports the decomposition of a compiler into several *compilation-phases*, the cross-cutting[6] nature of language constructs and extensions cannot be modeled. Even if an extension would only extend one *compilation-phase*, which usually is not the case, its additional behaviour would have to be incorporated into the *compilation-phase's* existing specification. However, to change existing specifications only to extend the compiler in a monotonic way is no appropriate concept for the *specification-driven* extensible compiler development. Another modularization technique is needed.

3.2 Modularization Across Concerns

In chapter 2.2 and the start of this chapter a general extension concept for the *specification-driven* compiler development already has been anticipated[7]: To develop a complete compiler, an arbitrary number of modules is implemented. Each module consists of an arbitrary number of specifications. To generate the compiler implementation, the specifications for each specification language are automatically composed, based on the specification language's formalism. The resulting unified specifications are the input for their respective tool, which generates a compiler component satisfying the unified specification. Since the generated compiler components are known in advantage[8], it is no problem to compose them statically together using always the same handcrafted *glue-code*

[5]Except the first one.

[6]cross-cutting *compilation-phases*

[7]The following presentation illustrates a general concept for the extensible *specification-driven* compiler development. The problems introduced by *JFlex* and *Beaver* are ignored.

[8]The *JastAdd* AST root node is known and the generated lexer and parser class always have the same qualified name, constructors and lexing/parsing method.

class[9], such that the complete compiler results. Hence, extensibility is achieved by the modularization on the specification level. An extension simply is another module consisting of specifications.

It remains to show how the compiler can be appropriately decomposed into such specification modules. What does a module represent, how do modules interact and how is the modularization technique just presented related to the modularization across *compilation-phases*? The next section will present a basic decomposition concept and the next but one section will show, how the dependencies between modules can be incorporated.

3.2.1 Basic Decomposition Concept

To find an appropriate modularization concept for compilers, it is enough to recall, that extensions are also simply modules. Though, what does an extension represent? Extensions introduce another language part into an existing language or contribute additional compiler features. The introduced part/feature may consist of an arbitrary mix of additional lexical constructs, syntactical constructs or semantics. Hence, an extension represents exactly one additional logical language construct/compiler feature. If an extension would introduce $n > 1$ logical language constructs/compiler features, it would be much better to decompose it in n independent extensions, because to specify closely related tasks together and to decouple unrelated tasks[10] results in a better modularization and consequently in more comprehensible specifications. Thus, each module of a compiler project should specify one logical language construct/compiler feature and to do so it may contain specifications for an arbitrary number of specification languages, such that all the language construct's/feature's tasks are specified using the most appropriate available specification formalism. It is obvious, the modularization concept across logical language constructs/compiler features does not replace the modularization across *compilation-phases*, but instead reuses it. Also, the need for the often mentioned automatically specification composition based on the specification's formalism becomes clear. Each specification formalism relates to one *compilation-phase* and a *compilation-phase's* tasks are specified by an arbitrary number of specifications contained in modules contributing logical language constructs/compiler features[11].

In the following, logical language constructs/compiler features are called concerns and their modularization is the modularization across concerns.

In general, the decomposition of a language in concerns is language dependent. The resulting modules should be orthogonal to each other. The cohesion between specifications contained in one module should be high and the coupling

[9] The *main-class*. See listing 2.1.

[10] Often the terms *cohesion* and *coupling* are used to measure the interaction within and between modules respectively and to conclude about the modularization's quality [4].

[11] Hence, if a logical language construct/compiler feature influences a *compilation-phase*, the module representing the logical language construct/compiler feature contains at least one specification for the tool which generates the compiler component realizing the *compilation-phase*. Of course, if a logical language construct/compiler feature influences a *compilation-phase* n, it usually also influences the following phases, since they must be defined for the extended result computed by phase n.

Table 3.1: *JastAddC's compilation-phase – concern matrix*

	Lexical	Syntactical (CFG)	Syntactical (AST)	Semantical
Compiler-Core	•	•	•	•
Access Support				•
AST Mapping		•	•	•
Constraint Verification				•
Error Handling			•	•
Implicit Information Retrieval				•
Name Analysis				•
Type Analysis				•
Extension: Flow Analysis				•
Extension: *Extended* `for`	•	•	•	•

between modules should be low, so the modules are as independent of each other as possible [4].

If a modularization across concerns is done, a *compilation-phase – concern matrix* can be constructed, which contains all the concerns the language is decomposed into and specifies for each concern the *compilation-phases* it influences. Table 3.1 represents the *compilation-phase – concern matrix* for *JastAddC*. The syntactical analysis is divided into the actual parsing based on the *Beaver* parser and the *JastAdd* AST constructed by the parser. Though, the modularization presented in Table 3.1 is idealised. As will be shown in section 3.3, the lexical and syntactical tools used do not support modularization, since only *JastAdd* supports the automatically specification composition based on its specification formalism[12]. Hence, all the lexical and syntactical parts of all concerns must be implemented in one *JFlex* and *Beaver* specification. E.g. consider the extension of *JastAddC* with an *extended* `for`. To specify the extension's lexical and syntactical constructs, the existing *JFlex* and *Beaver* specification must be manipulated. Clearly, such an extension process does not satisfy the *specification-driven* extensible compiler development's objectives (see chapter 2.1). However, if the solution concepts presented in chapter 7.1 are incorporated, all *compilation-phases* of *JastAddC* can be modularized across concerns, such that the modularization can be realized as presented in table 3.1.

Although, concerns should be self-contained they still may depend on each other. They contribute to one language and their interaction realizes the language. Also, extensions have to be developed based on the host language. Thus, a concept is needed, which permits concerns to reuse results computed by other concerns without exposing the implementations. The next section will present an appropriate solution: The *compiler-core* concept.

[12]ReCRAGs

3.2.2 *Compiler-Core* and Extensions

One of the greatest challenges in compiler development is the intended integrity of programming languages. All the language constructs are well balanced on each other. Hence, all the language constructs are somehow connected with each other. These interconnections introduce a very high complexity and result in confusing compiler implementations. To separate the different language constructs, such that the implementation becomes comprehensible again, the modularization across concerns can be used. However, the connections between the concerns are still valid. It is necessary to be able to specify for each concern its functionalities and their results, such that other concerns can reuse and extend them. Thus, the well known concept of the separation between interfaces and their implementation is needed.

In *JastAddC*, such a separation is realized for *C's* semantics, i.e. for the AST and its attribution. The key idea, to separate the semantic implementations of concerns from their interfaces, is to specify all the concerns **the compiler must realize** (the *core-concerns*) and which influence the compiler's semantics as another concern, the *compiler-core*. Thus, a concern's implementation can reuse the semantics of *core-concerns*, since the *compiler-core* claims, that the semantic functionalities of such *core-concerns* are supported. The *compiler-core* module consists only of a set of semantic declarations: The semantic interfaces of the *core-concerns* as well as the AST structure these concerns are based on (the *core-AST*). Each of the declared *core-concerns* still has to be implemented as its own module. Though, its actual semantical implementation may depend on the semantics of other *core-concerns* and additional AST constructs can be introduced based on the *core-AST*. Analogous, compiler extensions can be implemented based on the *compiler-core*. In fact, the distinction if a module is an extension or not is, whether it realizes a *core-concern* or not.

Of course, especially for compiler extensions a full modularization across concerns and not only for the semantics would be very helpful. In such a case the *compiler-core* would not only specify the semantics the compiler must support, but also the syntax and tokens the compiler must process. Thus, the *compiler-core* would contain specifications for all *compilation-phases*, such that an compiler extension's additional lexical and syntactical constructs can be locally specified in the extension's module and are integrated automatically. Chapter 7.1 suggests SGLR parsing as a solution to provide the full modularization across concerns.

Before the next section investigates why *Beaver* and *JFlex* cannot support the modularization across concerns, an example will present how *JastAddC's* semantical modularization across concerns is implemented based on a *compiler-core* specified using *JastAdd*.

Figure 3.2 illustrates *JastAddC's* modularization for the concerns presented in table 3.1[13]. Each *core-concern* is declared as one aspect of the `CompilerAPI-.jrag` specification. It consists of an arbitrary number of attribute declarations, AST extensions and support methods and classes. The declared attributes are associated with the *core-AST*, which is specified in the `C99.ast` specification.

[13]The idealised modularization for the lexical and syntactical *compilation-phases* is not incorporated, so figure 3.2 presents *JastAddC's* current modularization.

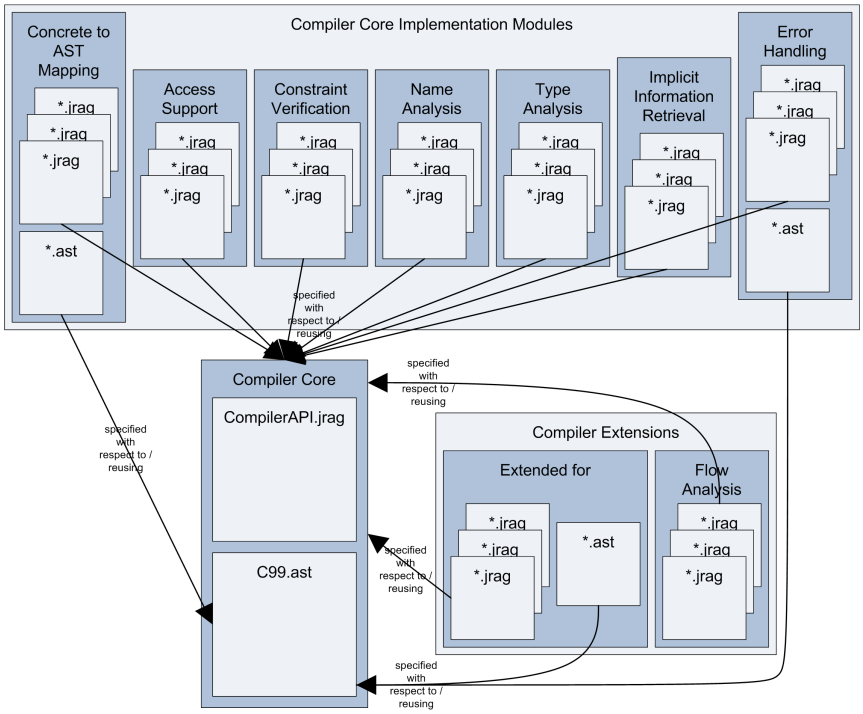

Figure 3.2: *JastAddC's* modularization: The *compiler-core* module, the modules implementing declared *core-concerns* and extension modules.

Listing 3.1 presents a specification fragment of the *core-AST* and listing 3.2 presents a specification fragment of the *core-concerns*.

Listing 3.1: *JastAddC*'s *compiler-core*: Example fragment of the *core-AST* (`C99.ast`) specification.

```
1  ...
2  TranslationUnit ::= Declaration:_IntermediateDeclarationCollection*;
3  ...
4  Declaration:Statement ::= StorageClassSpecifier* FunctionSpecifier* Type:
       _IntermediateType [BindedName:Identifier] [Initializer];
5  ...
6  abstract Initializer ;
7       FunctionDefinitionInitializer : Initializer  ::= Body:CompoundStatement;
8      abstract NormalInitializer: Initializer ;
9           ...
10 ...
11 abstract Type:_IntermediateType ::= TypeQualifier*;
12      ...
13     abstract TagedType:Type ::= [Tag:Identifier];
14          Enumeration:TagedType ::= EnumerationElement*;
15          ...
16     abstract StructuredType:TagedType ::= Element:UnionOrStructureElement*;
17              Structure:StructuredType;
18              Union:StructuredType;
19               ...
```

```
20  ...
```

Listing 3.2: *JastAddC*'s *compiler-core*: Example fragment for the declaration of *core-concerns* (`CompilerAPI.jrag`).

```
1  aspect AccessSupport {
2      ...
3      syn boolean Declaration.IsFunctionPrototyp();
4      ...
5      inh FunctionType Declaration.IsParameterDeclaration();
6      ...
7      coll java.util.List<ReturnJump> FunctionDefinitionInitializer.
           AssociatedReturnJumps()
8          [new java.util.LinkedList<ReturnJump>()] with add;
9      ...
10     inh jastaddc.semantics.NodeAddress ASTNode.NodeAddress();
11     ...
12 }
13
14 aspect ConcreteToASTMapping {...}
15 aspect ConstraintVerification {...}
16 aspect ErrorHandling {...}
17 aspect ImplicitInformationRetrieval {...}
18
19 aspect NameAnalysis {
20     ...
21     syn java.util.List<IOrdinaryNameDeclaration> ASTNode.
           LookUpOrdinaryName(String name);
22     ...
23     syn java.util.List<LabeledStatement> ASTNode.LookUpLabel(String name);
24     ...
25     syn UnionOrStructureElement StructuredType.LookUpMember(String
           memberToFind);
26     ...
27     syn java.util.List<TagedType> ASTNode.LookUpTag(String name);
28     ...
29 }
30
31 aspect TypeAnalysis {...}
```

Listing 3.2 specifies all the seven *core-concerns* of the *compilation-phase – concern matrix* presented in table 3.1. The module implementing the `AccessSupport` concern for example must define the equations of at least four attributes:

1. A *synthesized attribute* `IsFunctionPrototyp()` associated with each `Declaration` node, which returns, whether the `Declaration` node it is associated with declares a function prototype or not.

2. An *inherited attribute* `IsParameterDeclaration()` associated with each `Declaration` node, which returns the function type of which the `Declaration` node represents a parameter declaration or `null`, if the node is no parameter declaration.

3. A *collection attribute* `AssociatedReturnJumps()` associated with each `FunctionDefinitionInitializer` node, which contains all the return statements contained in the function's body.

4. An *inherited attribute* `NodeAddress()` associated with each node, which represents the node's unique *Dewey-notation* [8] address.

The `AccessSupport` concern's implementation can reuse any attributes, methods and classes declared in the *compiler-core* or itself. For convenient reasons the equations for a declared attribute should be specified in a *JastAdd* aspect named like the one containing the attribute's declaration.

JastAddC's complete *compiler-core* can be investigated in appendix B. B.1 specifies the *core-AST* and B.2 the *core-concerns*.

3.3 Unresolved Problems

Throughout the thesis it has been mentioned, that *Beaver* and *JFlex* do not support the automatically composition of their specifications. Hence, they do not support the modularization across concerns. It remains to show why this is the case.

For *Beaver* it already has been mentioned in chapter 2.2, that LR(k) grammars are not closed under unification. Thus, if a language's syntax is specified by LR(k) grammar specifications splited across an arbitrary number of modules (concerns), it cannot be guaranteed, that the unified syntax specification is still LR(k). A compiler extension may introduce additional syntax constructs, such that the LR(k) parser generator cannot generate an appropriate parser. Hence, *Beaver* does not only miss the functionality to support the modularization across concerns, it is inconvenient to do so based on LR(k) grammars.

The modularization problems for *JFlex* specifications are not as obvious as the ones for *Beaver*. From a theoretical view point *regular-expressions* are closed under unification [2]. Even if it is often said in the literature, that lexers are simply DFAs, this is not the complete truth. A DFA accepts or discards its complete input. However, a lexer's input has to be accepted, such that a stream of tokens is produced: The DFA, the lexer is based on has to be executed several times. The lexer has to break his complete input in chunks accepted by its DFA. Additionally, for each chunk accepted, the lexer has to decide in a deterministic way which token the chunk represents.

To break the input in chunks is no problem. The *principle of the longest match*[14] can be used to circumvent the problem, that although the lexer's DFA is deterministic, it still may have more than one accepting state. Hence, the complete input can be split in a deterministic way in chunks accepted by the lexer's DFA, even if several different such accepted splittings for the same input exist. Since the *principle of the longest match* is a global rule, independent of the language's tokens, it also does not introduce any modularization problems. Simply always the same rule can be used to break the input into chunks.

[14] If the DFA is currently in an accepting state and the next input character's consumption results again in an accepting state, the next input character is part of the current input chunk.

However, the decision which token a chunk represents cannot be done based on a global rule independent of the language's tokens. After splitting the input in a deterministic way into chunks, some chunks may represents input character sequences accepted by several *regular-expressions*[15]. Thus, some chunks can represent several different tokens. To decide in a deterministic way, which token such a chunk represents, *precedence-rules* between the involved tokens are necessary. It is obvious, *precedence-rules* cannot be specified in a global way independently of the language's tokens. Special *precedence-rules* must be specified depending on all the compiler's tokens. A compiler extension may introduce additional tokens, such that conflicts between tokens result and additional *precedence-rules* are needed. *JFlex* does not only miss the functionality to support the modularization across concerns, it is inconvenient to do so based on *regular-expressions*, *the principle of the longest match* and *lexer states*.

In chapter 7.1 SGLR parsing will be suggested as an appropriate formalism supporting the modularization across concerns for the lexical and syntactical *compilation-phases*. Though, it must be said, that also SGLR parsing does not solve all the problems. Modularization problems[16] encountered after unifying the SGLR specifications of the compiler's modules will result in the construction of parse forests instead of parse trees. Chapter 7.2 suggests a disambiguation technique for *JastAdd* ASTs representing parse forests. It is obvious, since the combination of the modules may introduce the ambiguity in the compiler, the extension of a compiler may require additional adjustments. A compiler extension may introduce additional SGLR constructs, such that the generated SGLR parser has additional non-determinisms, which must be disambiguated. The advantage of the approach suggested in chapter 7.1 and 7.2 is, that the concepts applied are completely theoretically backed, the disambiguation can be specified in its own module and the disambiguation can be done using the powerful concepts of ReCRAGs.

[15] E.g. the *regular-expressions* $([a-z]|[A-Z]) + ([a-z]|[A-Z]|[0-9])*$ and $''keyword''$ share the character sequence *keyword*.

[16] A modularization problem is given, if the generated compiler or compiler component is ambiguous.

Chapter 4

Lexical and Syntactical Components

In this chapter *JastAddC's* lexical and syntactical components are presented. Their implementation is based on *JFlex* and *Beaver* specifications as illustrated in chapter 2.3 and 2.5. Although, a modularization across concerns is not possible for *JFlex* and *Beaver* (see chapter 3.3), the lexical and syntactical specifications still realize a modularization across *compilation-phases* (see chapter 3.1). Of key importance for the implementation has been to reflect the *C* standard [23] in a comprehensible way, such that specified language constructs can be mapped to the standard and vice versa. The compiler's performance has only been of secondary priority. Hence, the chosen compiler design sometimes results in suboptimal performance.

4.1 Lexical Components

The best way to understand *JastAddC's* lexical implementation is to investigate the lexical constructs and tasks specified in the standard ([6.4;p. 49]) and their execution timing throughout compilation ([5.1.1.2;1;p. 9]). It follows, that two general lexical tasks have to be performed:

1. The recognition and construction of the actual tokens as they are defined and embedded in the language's syntax (see appendix A part "A.1 Lexicalgrammar").

2. Lexical transformations of the input program to compile, e.g. the interpretation of the physical source file characters, the line splicing and the transformation of escape sequences.

Each lexical transformation can be regarded as one *pass* of the lexical *compilation-phase*. The recognition and construction of *C's* tokens is one such lexical transformation. Although, for preprocessing reasons the standard defines, that the tokens, the parser has as input, are recognized based on preprocessor-tokens constructed in another *translation phase*. Since *JastAddC* has no preprocessor, the recognition and construction of the actual tokens can be implemented as one

pass/specification[1]. Of course, the lexer's output has to be a stream of tokens, on which all the standard's lexical transformations are already performed. Hence, the *pass* recognizing and constructing the tokens can be viewed as the lexical analysis' main task, supported by the other lexical transformations/*passes*.

To understand the output constructed by *JastAddC's* lexer, the next section will present the implementation of the *pass*, which recognizes and constructs the tokens. Afterwards, the complete design, structure and interaction of *JastAddC's* lexical *passes* will be investigated.

4.1.1 The lexical Main *Pass*: Token Recognition and Construction

Five different token types can be distinguished in *C*:

Keywords ([6.4.1;p. 50], A.1.2): `int`, `if`, `sizeof`

Punctuators ([6.4.6;p. 63], A.1.7): `+`, `=`, `{`

Indentifiers ([6.4.2;p. 51], A.1.3): `x`, `MyVariable`

Constants ([6.4.4;p. 54], A.1.5): `1u`, `0x.p-A3f`, `L'ü'`

String Literals ([6.4.5;p. 62], A.1.6): `"Hello World"`, `L"Bürger"`

The last three token types are the most interesting ones, since their *regular-expressions* match more than one character sequence. In fact, constant and string literal tokens may be augmented with additional information beside the raw numeric value or string. Integer and floating constants can have a suffix describing the constants type. Character constants and string literals can be augmented with a `L` to declare them to be a wide character constant/string literal. Hence, it is not enough for these tokens, if the lexer only returns their id and the input matched. Instead, the tokens returned for constants and string literals should also model their augmented semantic information, since beside the lexem all the augmented information are needed in further semantic evaluations[2]. Thus, constant and string literal tokens, but also identifiers, have to be represented in the AST and keywords and punctuators have not[3]. Consequently, complex AST node types are introduced for identifiers, constants and string literals. Listing 4.1 shows the *compiler-core's* AST specification for these tokens.

Listing 4.1: Specification fragment of *JastAddC's core-AST*: AST nodes representing identifiers, constants and string literals.

```
1  // Lexem: The original character sequences matched by the lexer.
2  abstract Constant:Expression ::= <Lexem:String>;
3
4      IntegerConstant:Constant ::= <NumericValue:String> Radix:
           NumeralSystemRadix Type:PrimitiveType;
```

[1] As described in chapter 3.1, each *JFlex* specification results in one lexer, which represents a *pass* of the lexical *compilation-phase*.

[2] E.g. during the name and type analysis.

[3] Keywords and punctuators are only needed for the syntactical analysis. However, in the AST such syntactical information can be encoded in its node types and structure.

```
5      FloatingConstant:Constant ::= <WholeNumberPart:String> <FractionalPart:
           String> <Exponent:String> Radix:NumeralSystemRadix Type:
           PrimitiveType;
6      /*
7       * CharacterValue, StringValue: The character constant's/string literal 's
8       *       value after mapping the source character members and escape
9       *       sequences (translation phase 5 [5.1.1.2; p. 9]).
10      */
11     CharacterConstant:Constant ::= <CharacterValue:String> <IsWideCharacter:
           boolean>;
12     StringLiteral:Constant ::= <StringValue:String> <IsWideString:boolean>;
13
14 abstract NumeralSystemRadix;
15
16     NumeralSystemOctal:NumeralSystemRadix;
17     NumeralSystemHexadecimal:NumeralSystemRadix;
18     NumeralSystemDecimal:NumeralSystemRadix;
19
20 Identifier ::= <Value:String>;
```

As can be seen, the specified AST nodes model the semantically relevant tokens as they are given in *C's* syntax definition (see appendix A part "A.1 Lexical-grammar"). Especially, all information which can be augmented with constants is captured (see appendix A part A.1.5). In opposite to the standard, string literals are treated as another constant type.

If the lexer matches a token representing an identifier, constant or string literal, it must return a *Beaver* symbol, whose value is an appropriate AST representing the token. Matched punctuator and keyword tokens are represented by a *Beaver* symbol, whose value is the matched input[4]. Of course, the id of any symbol the lexer returns must be instantiated appropriately using the ids *JastAddC's Beaver* parser supports.

The lexical recognition and construction *pass* is realized by a *JFlex* lexer named `jastaddc.symbols.lexicalanalyse.CLexer`. Listing 4.2 presents the most important fragments of its specification[5]. Only *macro definitions* and *lexical rules* are shown[6]. The `createToken` method is used to construct a new *Beaver* symbol. It expects at least one argument: An id supported by the `Terminals` class of *JastAddC's* parser. Additionally, a value of type `java.lang.Object` can be given. The symbols, the `createToken` method returns, will have correct line, column and length attributes and the id and value given as arguments. If the value argument is left, the constructed symbol's value will be the matched input characters[7].

Listing 4.2: Specification fragment of *JastAddC's* token recognition and construction *pass* (`CLexer`).

```
1 LineTerminator = \r|\n|\r\n|\u2028|\u2029|\u000B|\u000C|\u0085
2 InputCharacter = [^\r\n]
3 WhiteSpace = {LineTerminator} | [ \t\f\u000B] // \u000B is the vertical tabulator
```

[4]Thus, the value of punctuator and keyword tokens is of type `java.lang.String`.
[5]`specifications\lexical\CLexer.jflex`
[6]Specification parts, which adapt the lexer for *JastAddC's Beaver* parser, are left.
[7]In which case the symbol's value is of type `java.lang.String`.

```
TraditionalComment = "/*" ~"*/"
EndOfLineComment = "//" {InputCharacter}*
Comment = {TraditionalComment} | {EndOfLineComment}

LatinLetter = [a-z]|[A-Z]
NonNullDDigit = [1-9]
DDigit = [0-9]
ODigit = [0-7]
HDigit = [0-9]|[A-F]|[a-f]

StringCharacter = [^\r|\n|\r\n|\u2028|\u2029|\u000B|\u000C|\u0085\"]
SingleCharacter = [^\r|\n|\r\n|\u2028|\u2029|\u000B|\u000C|\u0085\']

DecExponent = ("e"|"E")("+"|"-")?{DDigit}+
BinExponent = ("p"|"P")("+"|"-")?{DDigit}+
FConstSuffix = "f"|"F"|"l"|"L"
IConstSuffix = (("u"|"U")("ll"|"LL"))
     |(("ll"|"LL")("u"|"U"))
     |(("u"|"U")("l"|"L"))
     |(("l"|"L")("u"|"U"))
     |("u"|"U")
     |("ll"|"LL")
     |("l"|"L")

 Identifier  = ({LatinLetter} | "_")({LatinLetter} | "_" | {DDigit})*

%%

<YYINITIAL> {
     {Comment} {/* Ignore comments */}

     /* keywords */

     "auto"       {return createToken(Terminals.kAUTO);}
     "break"      {return createToken(Terminals.kBREAK);}
     "case"       {return createToken(Terminals.kCASE);}
      ...
     "volatile"   {return createToken(Terminals.kVOLATILE);}
     "while"      {return createToken(Terminals.kWHILE);}

     /* integer constants */

     "0"("x"|"X"){HDigit}+{IConstSuffix}?
     |"0"{ODigit}*{IConstSuffix}?
     |{NonNullDDigit}{DDigit}*{IConstSuffix}? {
          integerLexer. initialize (yyline, yycolumn, new StringReader(yytext()));
          return createToken(Terminals.CONSTANT,
                    integerLexer.createIntegerConstant());
     }

     /* floating point constants */

     {DDigit}+{DecExponent}{FConstSuffix}?
```

```
    |{DDigit}*"."{DDigit}+{DecExponent}?{FConstSuffix}?
    |{DDigit}+"."{DDigit}*{DecExponent}?{FConstSuffix}?
    |"0"("x"|"X"){HDigit}+{BinExponent}{FConstSuffix}?
    |"0"("x"|"X"){HDigit}+"."{HDigit}*{BinExponent}{FConstSuffix}?
    |"0"("x"|"X"){HDigit}*"."{HDigit}+{BinExponent}{FConstSuffix}? {
        floatLexer. initialize (yyline, yycolumn, new StringReader(yytext()));
        return createToken(Terminals.CONSTANT,
                floatLexer.createFloatingPointConstant());
    }

    /* character-constants and string-literals */

    "L\""({StringCharacter}|"\\\"")*"\"" {
        return createToken(Terminals.STRING_LITERAL,
                new StringLiteral(yytext(),
                        yytext().substring(2, yytext().length() - 1),
                        true)); // Is wide string-literal
    }
    "\""({StringCharacter}|"\\\"")*"\"" {
        return createToken(Terminals.STRING_LITERAL,
                new StringLiteral(yytext(),
                        yytext().substring(1, yytext().length() - 1),
                        false)); // Is no wide string-literal
    }
    ...

    /* identifiers */

    { Identifier } {
            return createToken(Terminals.IDENTIFIER,
                    new Identifier(yytext()));
    }

    /* typedef names */

    "$"{ Identifier } {
        return createToken(Terminals.TYPEDEF_NAME,
                new Identifier(yytext().substring(1)));
    }

    /* punctuators */

    ("["|"<:")   {return createToken(Terminals.pBRACKETOPENSQUARE);}
    ("]"|":>")   {return createToken(Terminals.pBRACKETCLOSESQUARE);}
    "("          {return createToken(Terminals.pBRACKETOPENROUND);}
    ...
    "|="         {return createToken(Terminals.pLINEVERTICAL_EQUALS);}
    ","          {return createToken(Terminals.pCOMMA);}

    {WhiteSpace} {/* Ignore whitespace */}
}

// Error fall back rule:
.|\n {throw new CUnknownTokenException(yyline, yycolumn, yytext());}
```

The *lexical rules* for whitespace, comments, punctuators and keywords are straightforward.

Interesting is the handling of integer- and floating constants. As discussed before and has been presented in listing 4.1, it is important to recognize all their information and not only the lexem.

For a better understanding of *JastAddC's* solution consider what has to be done to recognize the information of such *complicated tokens*[8] and how the resulting specification interacts with the other tokens. To retrieve the semantic information given in the different parts of *complicated tokens*, their recognition must be split across several *lexical rules*. The information given in some token part can then be stored temporarily by the *semantic action* associated with the *lexical rule*, which has been used to match the token part. After processing all parts, all the temporarily stored information is gathered to construct an AST node representing the *complicated token*. To organize the matching of the token parts *lexer states* can be used, such that the different semantic information are retrieved in different *lexer states*. Thus, after recognizing the first part of a *complicated token* the lexer switches into a special *lexer-state*, which keeps on processing the next part and so one. Unfortunately, as can be seen by integer and floating constants, *complicated tokens* can start with similar parts. Hence, the recognition of their parts cannot be developed independently, if the tokens are handled by the same lexer. Even worse, such interactions may be given between *complicated tokens* and ordinary tokens or tokens introduced in further compiler extensions. Consequently, the specification of a lexer recognizing at the same time all tokens as well as the information associated with *complicated tokens* would become very complicated. Additionally, such a lexer would not be extensible[9].

A better approach is to match the complete *complicated token*, as it is done in *JastAddC* and presented in listing 4.2. After matching the complete token, a special lexer processes the matched token again to construct an object representing all the information augmented with the token. For each *complicated token* such a special lexer can be independently implemented. In listing 4.2 this approach is used for integer and floating constants. If the *C* lexer recognizes an integer or floating constant it initializes a special integer/floating constant lexer with the matched input. The special lexer constructs the value[10] of the *Beaver* symbol to return for the constant. Of course, the special integer and floating constant lexers are also implemented using *JFlex*[11]. For character constants and string literals such special lexers are not needed. To decide, whether a character constant/string literal is wide or not, is quite easy.

The specification of identifiers and typedef-names in listing 4.2 presents a speciality of *JastAddC*. To circumvent the *typedef problem*, which will be presented in detail in section 4.3, the *C* language is slightly changed: In *JastAddC*, identifiers starting with a `$` sign are interpreted as typedef-names.

[8] Tokens consisting of several relevant parts, whereas different parts represent different information.

[9] Because of the problems just mentioned.

[10] The AST representing the constant.

[11] `specifications\lexical\CIntegerConstantLexer.jflex` and `specifications\lexical\CFloatingConstantLexer.jflex`

4.1.2 Lexer Design and *Pass* Interaction: *JastAddC's Lexer-Stack*

The just presented token recognition and construction *pass* is embedded between other lexical *passes JastAddC* performs. Each lexical *pass* realizes one of the standard's *translation phases*[12] ([5.1.1.2;1;p. 9]) and is implemented as one *JFlex* specification[13]. The standard defines the following *translation phases*:

1. The mapping of the physical source file multibyte characters to the source character set[14] and the recognition and transformation of *trigraph sequences* ([5.2.1.1;1;p. 18]).
2. The elimination of line splicings[15].
3. The recognition and construction of preprocessing tokens.
4. The execution of preprocessing directives.
5. The mapping of source character set members and escape sequences in character constants and string literals[16] to the execution character set[17].
6. The concatenation of adjacent string literals.
7. The discarding of white-space characters and the transformation of each preprocessing token into a token. The syntactical and semantical analysis of the resulting tokens.
8. The resolution of external object and function references and the linking of library components, such that a program image is constructed, which contains information needed for execution in its execution environment.

As can be seen, only the *phases* one to six and parts of *phase* seven (the construction of tokens and discarding of white-space) handle lexical problems. Since *JastAddC* has no preprocessor, *phase* 4 is not covered and the recognition and construction of preprocessor tokens (*phase* 3) and the transformation of them into tokens (*phase* 7) are handled by the same *pass*: The token recognition and construction *pass* presented in section 4.1.1.

The lexers realizing *JastAddC's* lexical *passes* are the following *Java* classes[18]:

CSourceReader *Translation phase* 1.

[12]The term *translation phase* is defined in the standard and is not related to the term *compilation-phase* introduced before, even if some of *C's translation phases* are implemented as their own *compilation phase* in *JastAddC*.

[13]Only the lexer realizing the concatenation of adjacent string literals (*translation phase* 6) is handcrafted and not implemented using *JFlex*.

[14]The set of characters in which source files are written ([5.2.1;1;p. 17]).

[15]Logical source lines can be split in *C* across several physical source lines using the \ character immediately followed by a new-line character.

[16]E.g. the string literal `L"100.00 \u20AC"` should be the string *100.00* € in the execution character set, if the execution character set does support the unicode sign € at all.

[17]The character set used throughout the compiled programs execution.

[18]All lexer classes are part of the package `jastaddc.symbols.lexicalanalyse`. Their *JFlex* specifications are contained in the folder `specifications\lexical` and named like the lexer classes generated for them, except with the file ending `*.jflex` instead of `*.java`.

CSplicer *Translation phase* 2.

CLexer *Translation phase* 3 and the lexical relevant parts of *phase* 7.

CCharacterMapper *Translation phase* 5.

CStringLiteralConcatenator *Translation phase* 6.

Their interaction is shown in figure 4.1. Each lexer consumes the input produced

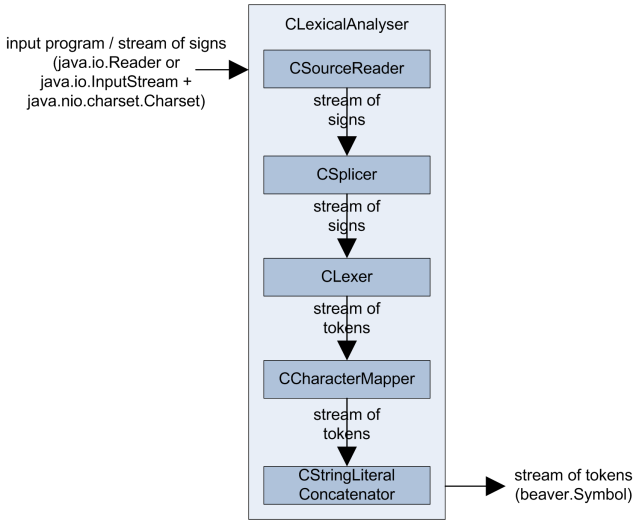

Figure 4.1: *JastAddC's lexer stack.*

by a preceding lexer, refines it and provides its output as input for the next lexer. Hence, *JastAddC's* lexers are organized in a *lexer stack*. To compose the lexers into one logical lexer a *glue-code* class, similar to the *main-class* skeleton presented in listing 2.1, is implemented: The `CLexicalAnalyser`. The `CLexicalAnalyser` lexer realizes the *lexer stack* and implements the interface *JastAddC's Beaver* parser expects. It represents the complete lexer. Listing 4.3 presents the fragment of the `CLexicalAnalyser` class, which composes the *lexer stack*.

Listing 4.3: `CLexicalAnalyser` code fragment: Composition of the *lexer stack*.

```
...
public class CLexicalAnalyser extends beaver.Scanner {
    ...
    private beaver.Scanner lexerStack;

    // Composing JastAddC's lexers using the default character encoding:
    public void initialize (java.io.Reader inputTextStream) {
        lexerStack = new CStringLiteralConcatenator(
                new CCharacterMapper(
                        new CLexer(
                                new CSplicer(
                                        new CSourceReader(inputTextStream)))));
    }
```

```
14
15      // Composing JastAddC's lexers using a user defined character encoding:
16      public void initialize (java.io.InputStream inputTextStream,
17              java.nio.Charset ENCODING) {
18          lexerStack = new CStringLiteralConcatenator(
19                  new CCharacterMapper(
20                          new CLexer(
21                                  new CSplicer(
22                                          new CSourceReader(
23                                                  inputTextStream,
24                                                  ENCODING)))));
25      }
26
27      // Constructors:
28      public CLexicalAnalyser(java.io.Reader inputTextStream) {
29          initialize (inputTextStream);
30      }
31
32      public CLexicalAnalyser(java.io.InputStream inputTextStream,
33              java.nio.Charset ENCODING) {
34          initialize (inputTextStream, ENCODING);
35      }
36       ...
37  }
```

The `initialize` method can be used to reinitialize the lexer with a new input. Two variants exist. The first expects an argument of type `java.io.Reader`, which represents the source code to analyse as a stream of characters. The second variant expects two arguments. One of type `java.io.InputStream`, which represents the input to analyse as a stream of bytes and an additional argument of type `java.nio.Charset`, which defines the input stream's interpretation as a sequence of characters. Hence, the second variant permits the easy adaptation of *JastAddC's* lexer for any user defined character encodings. Thus, the mapping of the physical source file multibyte characters to the source character set, as specified in the standard's *translation phase* 1, is supported for any character encodings.

The lexical *passes' JFlex* implementation is straightforward. Section 4.1.1 already showed the implementation for *translation phases* 3 and 7.

To recognize and transform *trigraph sequences* (as specified in *translation phase* 1), it is enough to specify for each *trigraph sequence* a *lexical rule*, whose *regular expression* is the *trigraph sequence* and its *semantic action* simply returns the character the *trigraph sequence* represents. E.g. for the *trigraph sequence* `??(` the *lexical rule* is `"??(" {return (int)'[';}`. Additionally, another fall back rule is specified, which forwards all the other signs: `.|\n {return yytext().codePointAt(0);}`. Since the first lexer's output is the input for another *JFlex* lexer and *JFlex* lexer's can only process streams of character's, the first *JFlex* lexer must be adapted to be a `java.io.Reader`. This is no problem, since the first lexer only replaces signs of the input program. The adaptation can be done in a similar way as the adaptation of *JastAddC's* token recognition and construction lexer to *Beaver's* `beaver.Scanner` class (see section 4.1.1). Hence, the first lexer's scanning method is renamed to `read()` and returns the

next input character read as `int` value. The mapping of physical source file multibyte characters to the source character set is handled in a similar way as in listing 4.3 (An arbitrary `java.nio.Charset` can be used to interpret the input as a stream of characters.).

The *JFlex* lexer performing *translation phase* 2 (the line splicing) is similar. It also is a `java.io.Reader` with just two *lexical rules*: `"\\"{LineTerminator}{}` and `.|\n {return yytext().codePointAt(0);}`. Thus, always a splicing of the current line is encountered it is removed, such that the next line becomes part of the current line. Any other signs are simply forwarded. Although, this lexer is very simple it eases the specification of the token recognition and construction *pass* a lot. If it would be left, the *regular expressions* of tokens would have to incorporate all possible splicings.

Translation phase 5 (the mapping of source character set members and escape sequences in character constants and string literals) is slightly more complicated to specify using *JFlex*. As can be seen in figure 4.1, starting from the `CLexer`, each lexer's output is a stream of tokens, a stream of *Beaver* symbols. Unfortunately, *JFlex* can only perform DFA analysis for streams of characters. Even though, the recognition of character constant and string literal symbols in the `CLexer`'s output stream is a regular problem and the transformation of escape sequences and character signs is also regular[19], it is not possible to process a stream of *Beaver* symbols using *JFlex*. The solution is to implement a handcrafted *adapter class*[20], which simply filters all character constant and string literal symbols and executes a special *JFlex* lexer. The special lexer has as input the character constant's/string literal's current, not yet mapped `CharacterValue`/`StringValue`[21] and returns the mapped value, which is the character constant's/string literal's new `CharacterValue`/`StringValue`. In *JastAddC*, the special lexer is named `CCharacterMapper_` and its *adapter class* `CCharacterMapper`. Listing 4.4 presents a few *lexical rules* of the `CCharacterMapper_`'s specification. The *lexical rules* presented map escape sequences. In fact, to map any other source character set members in character constants and string literals is not necessary as long as source- and execution character sets are equal.

Listing 4.4: `CCharacterMapper_` specification fragment: *Lexical rules* to map escape sequences in character constants and string literals.

```
...
%{
    /*
     * value: The string contained in value is the mapped CharacterValue/
     *        StringValue constructed based on a CharacterConstant/
     *        StringLiteral AST node's current, not yet mapped CharacterValue/
     *        StringValue terminal node. Hence, the string contained in value
     *        will be the CharacterConstant's/StringLiteral's new
     *        CharacterValue/StringValue.
     */
    private StringBuffer value;
    ...
```

[19] and could be embedded into special *lexer states*

[20] This *adapter class* is a special kind of *glue-code* class.

[21] The `CharacterValue`/`StringValue` is the terminal node of the AST node representing the character constant/string literal (see listing 4.1).

```
13 %}
14 ...
15 %type String
16 %eofval{
17     return value.toString();
18 %eofval}
19 ...
20 HDigit = [0-9]|[A-F]|[a-f]
21
22 %%
23
24 // Map standard escape sequences to the character sign they represent:
25 "\\'"    {value.append('\'');}
26 "\\\""   {value.append('\"');}
27 "\\a"    {value.append('\u0007');} /* unicode bell */
28 ...
29
30 // Map unicode escape sequences to the character sign they represent:
31 "\\u"{HDigit}{HDigit}{HDigit}{HDigit} {
32             try {
33                 value.append(Character.toChars(
34                         Integer.parseInt(yytext().substring(2), 16)));
35             } catch (java.lang.IllegalArgumentException exception) {
36                 throw new CUnknownTokenException(yyline,
37                     yycolumn,
38                     value.toString(),
39                     "No valid unicode escape sequence: "+ yytext());
40             }
41 }
42 ...
```

To specify a *JFlex* lexer for *translation phase* 6 (the concatenation of adjacent string literals) is not possible because of similar reasons as for *translation phase* 5. Once again, the lexer's input is a stream of tokens. However, the task to perform this time is even more easy, so that there is no need for a special lexer. A simple hand-crafted class can perform the concatenation of adjacent string literals: The `CStringLiteralConcatenator`. If *JFlex* would support the processing of arbitrary symbols, not only character signs, it could be used to implement the `CStringLiteralConcatenator`.

4.2 Syntactical Components

The specification of *JastAddC's Beaver* parser is similar to the calculator compiler's one presented in listing 2.8. The CFG rules specified are equivalent to the standard's syntax (see appendix A) and the implementation of each rule's *semantic action* is based on *JastAddC's core-AST* (see chapter 3.2.2 and B.1).

Listing 4.5 presents a typical fragment of *JastAddC's* parser specification.

Listing 4.5: `CParser` specification fragment: CFG rules for `primary_expression` (6.5.1), `argument_expression_list` (6.5.2) and `selection_statement` (6.8.4).

```
1 %import "jastaddc.semantics.ast.*";
```

```
...

%typeof IDENTIFIER = "Identifier";
%typeof CONSTANT = "Constant";
%typeof STRING_LITERAL = "StringLiteral";
%typeof expression = "Expression";
%typeof primary_expression = "Expression";
%typeof argument_expression_list = "List";
%typeof statement = "Statement";
%typeof selection_statement = "Statement";
...

selection_statement = kIF.token pBRACKETOPENROUND expression.expr
      pBRACKETCLOSEROUND statement.stmt
          {:
              IfStatement result = new IfStatement(expr, stmt, new Opt());
              result.line = expr.line;
              result.column = expr.column;
              return new Symbol(result);
          :}
      | kIF.token pBRACKETOPENROUND expression.expr
          pBRACKETCLOSEROUND statement.stmt1 kELSE statement.stmt2
          {:
              IfStatement result = new IfStatement(expr, stmt1, new Opt(stmt2));
              result.line = expr.line;
              result.column = expr.column;
              return new Symbol(result);
          :}
      | kSWITCH.token pBRACKETOPENROUND expression.expr
          pBRACKETCLOSEROUND statement.stmt
          {:
              SwitchStatement result = new SwitchStatement(expr, stmt);
              result.line = expr.line;
              result.column = expr.column;
              return new Symbol(result);
          :}
      ;
...
argument_expression_list = assignment_expression.expr
          {:
              List list = new List();
              list.add(expr);
              list.line = expr.line;
              list.column = expr.column;
              return new Symbol(list);
          :}
      | argument_expression_list.list pCOMMA assignment_expression.expr
          {:
              list.add(expr);
              return _symbol_list;
          :}
      ;
...
primary_expression = IDENTIFIER.id
```

```
53          {:
54              id.line = Symbol.getLine(_symbol_id.getStart());
55              id.column = Symbol.getColumn(_symbol_id.getStart());
56              Variable var = new Variable(id);
57              var.line = id.line;
58              var.column = id.column;
59              return new Symbol(var);
60          :}
61      | CONSTANT.con
62          {:
63              con.line = Symbol.getLine(_symbol_con.getStart());
64              con.column = Symbol.getColumn(_symbol_con.getStart());
65              return _symbol_con;
66          :}
67      | STRING_LITERAL.lit
68          {:
69              lit.line = Symbol.getLine(_symbol_lit.getStart());
70              lit.column = Symbol.getColumn(_symbol_lit.getStart());
71              return _symbol_lit;
72          :}
73      | pBRACKETOPENROUND expression.expr pBRACKETCLOSEROUND
74          {:
75              return _symbol_expr;
76          :}
77      ;
78  ...
```

The `selection_statement`'s specification exemplifies, how trivial the specification of *JastAddC's* parser, based on the standard's syntax and the *compiler-core's* AST, is. The only speciality is the initialization of line and column variables[22] associated with each AST node. If such attributes would not exist, the information which input program part an AST node represents would be lost, since the symbols *JastAddC's* lexer constructs are thrown away after their reduction.

The two rules for the `if` statement represent the CFG's only LALR(k) conflict, also known as the *dangling* `else` *problem*. However, *Beaver* automatically solves the shift-reduce conflict correct, since shifting the `else` token is always prefered to the reduction with rule 6.8.4: `selection-statement` $\longrightarrow$ `if (expression) statement`.

The `argument_expression_list`'s specification illustrates a general pattern for the implementation of recursive *non-terminals*, which describe a sequence of similar language constructs[23]. Such sequences can be represented by *JastAdd* `List` nodes. The *semantic actions* of atomic rules for the recursive *non-terminal* construct a new *JastAdd* `List` node, append the atomic element to the list and return a new *Beaver* symbol with the constructed list as value. The *semantic actions* of recursive rules access the recursive production element's list by reference, append the additional constructs and return the symbol associated with the list reference.

[22] The line and column variables are specified in the *compiler-core* (see appendix B.2).
[23] E.g. `parameter_list`, `block_item_list` and `specifier_qualifier_list`.

The rules for the *non-terminal* `primary_expression` highlight the reuse of the *complicated tokens*[24] *JastAddC's* lexer constructed (see section 4.1.1). Each of the symbol references `id`, `con` and `lit` refers to an AST node of type `Identifier`, `Constant` or `StringLiteral` respectively[25]. Since `Constant` and `StringLiteral` AST nodes inherit from `Expression`, the *semantic actions* of the `primary_expression = CONSTANT.con` and `primary_expression = STRING_LITERAL.lit` rules can simply return the symbol associated with their `con` and `lit` references. However, the `primary_expression = IDENTIFIER` rule represents a variable expression, wich still must be constructed.

4.3 Unresolved Problems - The *Typedef Problem*

Beside *Beaver's* and *JFlex's* modularization problems already presented in chapter 3.3, *JastAddC's* implementation cannot handle the *typedef problem*. *JastAddC* does not decide for identifiers, whether they represent a typedef-name ([6.7.7;p. 123]) or not. Instead, the language is changed in such a way, that identifiers starting with a `$` sign are interpreted as typedef-names. Clearly, such a behavior is not standard conforming. The following section will analyse what the *typedef problem* is and why it is so hard to specify a comprehensible solution based on theoretical concepts which does not look like a "hack". Afterwards, *typedef problem* examples are shown to illustrate the problem's scale and complexity.

4.3.1 *Typedef Problem*: Overview

The *typedef problem* is not explicitly mentioned in the standard. Instead, it is silently induced by *C's* language definition. The standard defines the concept of *type synonyms*, which are introduced by declarations ([6.7;p. 97]) with the *storage-class specifier* `typedef` ([6.7.7;p. 123]). Such *synonyms* are ordinary identifiers, though they do not represent objects, functions, enumeration elements or tags but instead the type they are a synonym for. Hence, on a syntactical viewpoint *type synonyms* can occur everywhere where a type is expected. *Type synonyms* are represented by the *non-terminal* typedef-name. Of course, the standard defines only one rule for typedef-name: $typedef-name \longrightarrow identifier$ (6.7.7). Unfortunately, this rule introduces ambiguities in the standard's CFG. If it would be possible to decide for each identifier as soon as it is encountered, whether it is a typedef-name or not, the ambiguities could be solved. To understand the scale of the *typedef problem* it is also necessary to know, that *type synonyms* are part of the *ordinary namespace* ([6.2.3;p. 31]). Hence, if one reads the input from left to right, token by token and likes to decide for each identifier, whether it is a typedef-name or not as soon as the identifier is encountered, it is necessary to know if a declared *type synonym* with equal name is in scope.

One might wonder, why someone would like to decide as soon as an identifier is encountered, whether it is a typedef-name or not. After all, the short discussion above made clear, that the *typedef problem* is not a context-free problem.

[24] Hence, AST nodes

[25] The `Identifier`, `Constant` and `StringLiteral` AST nodes are defined in *JastAddC's compiler-core* (see listing 4.1).

To push it into the lexer, which can only handle regular problems, does not seem to make sense. However, that's the way most handcrafted *C* compilers work! The lexer has to decide for each identifier, whether it is a typedef-name or not[26]. Of course, the lexer can't do so. It has to get some kind of decision information provided by the parser. However, as discussed above, the parser must know if a declared *type synonym* with equal name is in scope to provide the decision. To do so, the parser maintains a *symbol table* during parsing, which must at least realize the complete semantics of *C's ordinary namespace*, i.e. the ordinary name analysis ([6.2.1;p. 29], [6.2.3;p. 31]). This mixing of lexical, syntactical and semantical analysis, of different language concerns is clearly a horrible hack with respect to compiler modularization, extensibility and comprehensibility. A much better solution would be to handle the context sensitive *typedef problem* as a separate language concern within the semantic analysis, i.e. to do some kind of semantic diambiguation as will be presented in chapter 7.2.

4.3.2 *Typedef Problem*: Examples

In the following example programs will be used to show why the disambiguation between ordinary identifiers and typedef-names by the lexer is extremely error-prone. The idea is to motivate why such an approach has not been chosen for *JastAddC*. It will be shown, that a lot of semantics must be evaluated to provide the lexer with the required disambiguation information. Of key importance is, that these semantics must be evaluated during the AST construction. Thus, the complete AST is not yet available when some of its semantics must be evaluated. This fact and the problem of the theoretically not backed cyclic interactions between the different *compilation-phases* prevent a *specification-driven* extensible solution, where the lexer disambiguates identifiers using a symbol table constructed by the parser, which is based on the tools used to implement *JastAddC*.

Additionally, the section prepares a deeper understanding of the *typedef problem* for chapter 7.2, where a solution based on GLR parsing and ReCRAGs, which satisfies the *specification-driven* extensible compiler development's objectives, will be suggested.

To get used to the *typedef problem* a short example program is presented in listing 4.6.

Listing 4.6: *Typedef problem* introduction example.

```
1  // The lexer must decide, that T is an identifier .
2  // Otherwise a syntax error results :
3  typedef long T;
4
5  void func1(void)
6  {
7      // The lexer must decide, that T is a typedef-name.
8      // Otherwise a syntax error results :
9      T variable;
```

[26]The advantage of the disambiguation of identifiers and typedef-names by the lexer is, that the *non-terminal* typedef-name can be replaced by a special typedef-name token in the standard's grammar, such that the grammar becomes unambiguous. The resulting grammar is LALR(k) with $k = 1$.

```
10      ...
11  }
12
13  void func2(void)
14  {
15      // The lexer must decide, that T is an identifier .
16      // Otherwise a syntax error results :
17      int T;
18      ...
19  }
20
21  void func3(void)
22  {
23      // The lexer must decide, that T is a typedef-name.
24      // Otherwise a semantical error results
25      // (IMPORTANT: On a syntactical view point T can be
26      // both, an identifier or typedef-name):
27      int size = sizeof(T);
28  }
```

Throughout listing 4.6, the identifier `T` is sometimes interpreted as type and sometimes as variable. It is assumed that the parser has a special typedef-name token instead of the standard's CFG rule $typedef - name \longrightarrow identifier$ (6.7.7). Hence, always the lexer matches an identifier it must decide, whether the identifier is a typedef-name or not.

Listing 4.7 presents an interesting problem for the construction of appropriate *symbol tables* during parsing: The possibility to change an identifier's interpretation within the same declaration.

Listing 4.7: *Typedef problem* example: Identifier representing a type and a variable within the same declaration.

```
1  typedef int IntType, FuncType(IntType);
2
3  // The above is equivalent to:
4  typedef int IntType;
5  typedef int FuncType(IntType);
```

The listing makes clear, that the annotation of the *symbol table* in *semantic actions* with a new identifier must be done immediately after the identifier's declarator is completed ([6.7.7;3;p. 123]). It is not sufficient to do so after the complete declaration has been processed. E.g. in line 1 the identifier `IntType` occurs two times within the same declaration. First, `IntType` is declared to be a *type synonym* for `int`. Afterwards the just declared *synonym* is used to specify a parameter's type. Since a declarator can only be followed by a comma (`,`), equal (`=`) or semicolon (`;`) token it is sufficient to require, that: Each newly declared identifier I is annotated in the *symbol table* before a token after the next comma (`,`), equal (`=`) or semicolon (`;`) token, which follows I, is processed by the lexer.

Listing 4.8 illustrates the complexity of the *symbol table's* scoping and shadowing semantics.

Listing 4.8: *Typedef problem* example: Scoping and shadowing of typedef-names.

```
/* Global scope block level 0 */
typedef char* Type;
char c;

Type func(void)
{
    /* Function block level 1 */
    long l1 = sizeof(Type), Type = 3, l2 = sizeof(Type);
    if (l1 != sizeof(char*))
    {
        /* Compiler implementation error */
    }
    if (sizeof(Type) != sizeof(long))
    {
        /* Compiler implementation error */
    }
    if (l2 != sizeof(long))
    {
        /* Compiler implementation error */
    }
    {
        /* Function block level 1.4 */
        long l1 = Type;
        if (l1 != 3 || sizeof(Type) != sizeof(long))
        {
            /* Compiler implementation error */
        }
        typedef double Type;
        if (sizeof(Type) != sizeof(double))
        {
            /* Compiler implementation error */
        }
    }
    {
        /* Function block level 1.5 */
        typedef char Type;
        if (sizeof(Type) != sizeof(char))
        {
            /* Compiler implementation error */
        }
        {
            /* Function block level 1.5.2 */
            Type Type;
            Type = 'c';
            c = Type;
        }
    }
    char* pc = &c;
    return pc;
}
```

Every time the `/* Compiler implementation error */` comment is reached, while executing the program generated for listing 4.8, the compiler has not been

able to associate the identifier `Type` with the correct declaration. Thus, the compiler's *symbol table* has been erroneous. It is obvious, that the parser must evaluate quite complicated name analysis semantics to maintain the symbol table.

Last but not least, listing 4.9 presents a *C* program, which illustrates the ambiguity of *C's* context-free syntax.

Listing 4.9: *Typedef problem* example: Proof for the ambiguity of *C's* syntax.

```
1 void func(void)
2 {
3     ...
4     (id)+1; // Cast or not?
5     ...
6 }
```

Two valid interpretations for the expression `(id)+1` exist on a context-free level. The identifier `id` can represent a type or a variable. If it is interpreted as type, a cast-expression results. If it is interpreted as variable, an addition results. Since there exist an infinite number of valid *C* identifiers and programs it is impossible to encode the correct decision information statically in the CFG. Every valid context-free *C* grammar, equivalent to the one defined in the standard[27], will be ambiguous.

[27] Two CFGs are equivalent, if both specify the same language.

Chapter 5

Semantical Components

In this chapter the implementation of *JastAddC's* semantical components will be presented. Of course, the specifications of *JastAddC's* semantical tasks are modularized across concerns, as presented in chapter 3.2. The rest of this chapter is divided in four sections.

The first section introduces the tasks a compiler front-end has to perform, suggests, how these tasks can be realized, and presents *JastAddC's* resulting *compiler-core* concerns.

Afterwards, the second section presents the implementation of *JastAddC's compiler-core* concerns. The main focus is to give a good overview about the applied techniques and concepts, which can be reused in other compiler projects. Hence, the reader acquires an idea how to implement different semantical tasks in a *specification-driven*, extensible way using *JastAdd* and the ReCRAG formalism. After reading this section, it should be no problem for the reader to further explore independently *JastAddC's* semantical specifications.

In the third section a ReCRAG based technique for AST mappings is presented. In *JastAddC* it is used to realizes the mapping of the AST constructed by the parser to another AST structure much more suited to implement *C's* semantics. The presented technique is generic and *C* is only used as a suitable example for its application.

Finally, the fourth section discusses remaining problems of *JastAddC's* semantical implementation.

5.1 *Core-Concerns*: Tasks and Interactions

The following sections will present the tasks and interactions of *JastAddC's compiler-core* concerns, starting with a general overview about a compiler front-end's tasks (section 5.1.1), followed by a general concept, how to implement the tasks (section 5.1.2), and finally the presentation of *JastAddC's* resulting modularization across concerns (section 5.1.3).

It is necessary to keep in mind, that *JastAddC's* result is the evaluated AST, which contains *C's* semantics represented as API functions: The attribute access methods (see chapter 2.3.1). Hence, each of *C's* semantical computations is represented by one attribute. The terms language semantics, semantical tasks and attributes can be exchanged mutually.

5.1.1 The Tasks of a Compiler Front-End and their Interactions

To comprehend *JastAddC's* modularization across concerns and the resulting *compiler-core* it is necessary to understand the difference between a language's specification and the tasks/semantics a compiler for the language has to perform/compute.

A language's specification is only concerned about the decision, whether a program is valid or not. It specifies semantical constraints/computations between language constructs, which must be satisfied. Thus, the language specification only bothers about correct programs and does not worry what to do if a program is erroneous. The semantics of a programming language are only defined for valid input programs. In contrast, the semantics of a compiler for the language must also be specified for erroneous input programs. Thus, a compiler has not only to reconstruct/compute a language's semantics, but also to recognize, internally represent and report errors. The computations of a language's semantics have to be extended in such a way, that they are still defined even a program is invalid.

E.g. consider language constructs, whose semantics depend on the existence of other language constructs, e.g. in *C* variables must be declared. Such language constructs may be used for the specification of additional semantics, e.g. in *C* variables are expressions and each expression has a type. Since the language's specification does not define the semantics of invalid programs, the semantics of the construct depending on another language construct not given in an erroneous program will not be defined[1]. However, the compiler's semantical computations for the depending semantics still must be defined and return some valid result, e.g. the type computed for a variable which is not declared will be some compiler internal error type.

Because of the additional requirements just discussed, the semantical computations of compilers often become a wild mix of the actual semantics specified in the language definition, the recognition and handling of special error cases, the internal error representation and the reporting of errors for the user. Clearly, such a mix is hard to understand, maintain and extend and the verification of the compiler implementation against the language's specification becomes a serious issue, because of the incorporation of all possible error cases.

To handle these problems in a convenient way, the semantic processing can be separated in three tasks:

1. Attributes representing the semantical computations specified in the language's specification. These attributes should be implemented as close to the language specification as possible, i.e. without the handling of error cases, except error cases explicitly mentioned in the language's specification. Especially, if the semantical computation represented by an attribute depends on another AST node, which might not exist in erroneous input programs, no special error case specifying what to do if the other node does not exist should be required.

[1] Just like the semantics of constructs depending on erroneous program parts are not defined.

2. Attributes representing errors in the AST. Especially nonexistent, but semantically required nodes have to be represented.

3. Attributes for constraint checking and error message construction.

The first task represents the implementation of the language's semantics, whereas the second and third tasks represent additional functionalities a compiler front-end has to support.

In *JastAddC* the first task is separated in different *compiler-core* concerns, each representing one of *C's* semantical concepts ([6.2;p. 29]). Also, the second and third tasks are separate *compiler-core* concerns. Thus, a modularization of *JastAddC's* semantics across concerns is given and the mixing of the semantics specified by the standard [23] with additional compiler tasks can be avoided.

5.1.2 A Generic Error Handling Approach

In this section will be shown, how the semantic computations' requirements[2], presented in the preceding section, can be satisfied and how the interaction between semantical computations, internal error representation and constraint checking and construction of error messages can be organized.

Table 5.1 summarizes the solution concept presented in the following.

As has been mentioned in section 5.1.1, it would be convenient if the semantical computations specified in the standard could be implemented in such a way, that an attribute depending on another node[3] which does not exist can be transparently evaluated, i.e. the attribute's default equation[4] still can be used and no special error case has to be incorporated. In *JastAddC*, such an one to one mapping of the standard's semantics is realized using the concepts of *error nodes*, *non-null reference attributes* and the representation of each semantical computation specified in the standard by one attribute[5].

Consider a semantical computation S_1, which is represented by the attribute A_1 associated with node N_1, depends on the semantical computation S_2, which is represented by the attribute A_2 associated with another non-local node N_2 (i.e., $N_2 \neq N_1$ and N_2 is not necessarly a child of N_1 and vice versa). In such a case the attribute equation for A_1 does not try to access N_2 directly, but uses instead a *helper attribute*, which computes whether N_2 exists or not. If N_2 exists, the *helper attribute* simply returns it, so A_1 can access A_2. If N_2 does not exist, some kind of mock/dummy AST node of N_2's type is returned: An *error node*. Thus, the *helper attribute* never returns `null`, it is a *non-null reference attribute*. Since the *error node* has N_2's type, A_1 can still access A_2 as usual using the error node's A_2 attribute.

[2]Semantical computations are specified as close as possible to the language's specification. Especially, if the semantical computation represented by an attribute depends on another AST node, which might not exist in erroneous input programs, no special error case specifying what to do if the other node does not exist should be required.

[3]Of course, an attribute which depends on an attribute associated with another node, depends on the other node.

[4]The semantical constraint/computation specified in the standard.

[5]In the rest of this section the term semantical computation always means one logical semantical constraint specified in the standard.

Table 5.1: Overview of *JastAddC's* concepts for semantical computations, internal error representation/handling and constraint checking/construction of error messages.

Task	Attributes	Dependencies on other Tasks	Results
Semantical Computations	Each semantical computation is represented by one attribute. Attribute equations reflect their definition in the standard.	Access of remote located semantical computations (attributes) via appropriate *non-null reference attributes*.	As defined in the standard, thus if an attribute value is undefined `null` or a collection with undefined element constellation.
Internal Error Representation/Handling	For each semantical constraint specifying a relationship between remote AST nodes a *non-null reference attribute* is introduced. Semantical computations of *error nodes* return values of such a kind, that semantical computations can use their default equations, even if working with *error nodes*. Each AST node maintains a set of its errors.	Semantical computations reused to specify the equations of *non-null reference attributes*.	Unsatisfied semantical computation/constraint: `null` or collection with undefined element constellation. Unsatisfied dependency between remote AST nodes: *Error node* is result of associated *non-null reference attribute*.
Constraint Checking/Construction of Error Messages	Each AST node has a local constraint checking method and a sub-tree traversing constraint checking method.	Local constraint checking methods reuse semantical computations and *non-null reference attributes*.	If errors are recognized, the constraint checking constructs an *error object*, associates it with the erroneous AST node and returns, after checking all its other constraints, a value signaling an error.

Thus, the task of an *error node's* semantical computations is to return values of such a kind, that other nodes' semantical computations can always use their default equations[6], even in the presence of errors. Which kind of value an *error node's* semantical computation must return to satisfy this behaviour is task dependent. E.g. assume there exists a special error type for the type analysis, which will always, for any type analysis related semantical computation, result in an error type again. In such a case, the semantical computations of *error nodes*, which return a type, can simply return the special error type.

To specify the equations of *non-null reference attributes* is rather easy. The association between the semantical computations S_1 and S_2 must be defined in the standard[7], let's say by the semantical computation S_3. The *non-null reference attribute* can simply reuse the attribute representing S_3 to compute, whether N_2 exists or not and return N_2 if it exists or an appropriate *error node* otherwise. E.g. let's assume S_1 is the type analysis, S_2 is the type of a declarator and S_3 is the name analysis. To compute the type of a variable it is necessary to access the type of the variable's declarator. Thus, the associated declarator node must be found, which might not exist in erroneous C programs. To retrieve always an appropriate declarator node a *non-null reference attribute* is used, which itself reuses the name analysis to look up the declarator and returns it, if it exists, or a declarator *error node* otherwise.

To summarize the concept presented so far consider the definitions of *non-null reference attributes* and *error nodes*:

Definition EH.D.1 (*non-null reference attribute*) *A* non-null reference attribute *is a* reference attribute *implemented based on a semantical computation, which specifies a relationship between syntactical independent language constructs/AST nodes, which valid programs must satisfy. If the semantical computation is satisfied, the* non-null reference attribute's *value is the one of the semantical computation. Otherwise, its value is an* error node *of the node type the semantical computation returns.*

Definition EH.D.2 (*error node*) *An* error node *E is an AST node representing a nonexistent, but semantically required AST node N in such a way, that semantical computations depending on N can use E instead and still evaluate correct, without the need to recognize, that E is an* error node.

In contrast to *non-null reference attributes*, the value of an attribute representing a semantical constraint/computation specified in the standard can be undefined and is, iff the input program does not satisfy the semantical constraint[8]. Hence, the interaction between an attribute A representing a semantical relationship computation[9] and its *non-null reference attribute* is, that the *non-null*

[6]The semantical computation as it is specified in the standard. Hence, without handling error cases not explicitly mentioned in the standard.

[7]Given by definition, since otherwise S_1 would not depend on S_2 or the standard is incomplete or erroneous.

[8]To represent undefined values `null` references or collections with special elements can be used. E.g. if an ordinary namespace entity is looked up, but it is not declared, an empty list is returned and if it is declared several times, a list with several elements is returned.

[9]A semantical computation, which is specified in the language's specification and specifies a relationship between syntactical independent language constructs/AST nodes, which valid programs must satisfy.

reference attribute is used to transform possible undefined results of A in well defined results, which can be used without bothering about $A's$ error cases[10].

It is obvious, that *non-null reference attributes* represent semantic errors, if their value is an *error node* and semantical computations represent errors, if their value is undefined[11]. Consequently, the implementation of the semantic constraint checking and construction of error messages simply traverses the complete AST and checks at each AST node the node's semantical computations and/or *non-null reference attributes*. If an error is encountered an appropriate *error object*, which stores all the error information important for the user[12], is constructed and associated with the erroneous AST node. Since the augmenting of the AST with *error objects* is a side-effect, it is important, that, even if a constraint checking attribute is evaluated several times, it does not augment the same error twice.

5.1.3 Overview of *JastAddC's Core-Concerns*

This section presents the tasks of and interactions between *JastAddC's compiler-core* concerns. The actual implementations will be presented in section 5.2.

JastAddC's compiler-core declares eight concerns: The *access support*, the *AST mapping*, the *constraint verification*, the *error handling*, the *external linkage semantics*, the *implicit information retrieval*, the *name analysis* and the *type analysis*. Table 5.2 summarizes *JastAddC's compiler-core* concerns and their tasks.

Each of *C's* semantical concepts ([6.2;p. 29]) is specified as one concern in the *compiler-core* and each such concern supports all the concept's semantical computations specified in the standard, whereas each semantical computation is represented by one attribute. There are mainly three such concerns, the *type analysis*, the *name analysis* and the *external linkage semantics*. The *type analysis'* task is to compute each expression's type, to compare types, to perform type promotions and to decide if expressions are constant. The *name analysis'* task is to compute the name spaces, i.e. to compute for each name space and AST node the declarations in scope and to associate variables, typedef-names, enumeration constants, tags, labels and structure members with their declarations. The *external linkage semantics* can be regarded and realized as part of the *name analysis*. However, since the association of an entity declared to be external with its final definition is quite complicated, it is convenient to distinguish the *name analysis* between names defined within the same translation unit ([5.1.1.1;p. 9]) and the ones which are not.

[10] The reader might wonder, why such *helper attributes*, which lift undefined results of semantical computations to defined ones, are not always used if one attribute representing a semantical computation depends on another one. Hence, why not introduce simply *non-null attributes* and not only *non-null reference attributes*. The answer is very pragmatic: It is not worth the effort. The equations for semantical computations would not become simplified and *JastAddC's* semantic specifications would be bloated with to many *helper attributes*. However, *non-null reference attributes* simplify the specification of attributes representing semantical computations depending on remote nodes a lot, since the error handling can be shifted from the attributes representing semantical computations to the *error nodes*.

[11] E.g. is `null` or a collection with special elements.

[12] E.g. source file, line and column number, error type and error description.

Table 5.2: Overview of *JastAddC's compiler-core* concerns and their tasks.

Compiler-Core Concern	Tasks
Access Support	Preparation of AST information and distribution of simple context information for further semantical computations. Specification and definition of *non-null reference attributes*.
AST Mapping	Transformation of the AST constructed by the parser into its final form expected by all the other semantical analysis.
Constraint Verification	Checking of local and (sub-)tree constraints. Construction of *error objects* and their association with the erroneous AST nodes.
Error Handling	Supports functionalities to associate errors with AST nodes, to access them (search functions, traversion functions etc.) and to retrieve *error nodes*.
External Linkage Semantics	Supports functionalities to look up entities of different name spaces across translation units.
Implicit Information Retrieval	Tree transformations to make information defined implicit in the standard explicit.
Name Analysis	Introduces *C's* name spaces and functionalities to look up entities.
Type Analysis	Support for compatibility checks, the construction of composite types, type promotions, constant expression checks and *lvalue*, *rvalue* and modifiable checks.

To compute the standard's semantics information stored in the AST must be accessed. Often this is trivial, but sometimes the information associated with an AST node has to be prepared for further semantical computations. Also, information about the AST's actual structure has to be distributed, such that it can be easily accessed in other AST parts. Throughout the thesis, the distribution of the structure given in an AST part (hence, a context[13]) to other AST parts is called the distribution of *context information*[14]. Semantical computations are often implemented by reusing, combining and reasoning about such distributed *context information*. Hence, it is important to support helper functionalities, which prepare the information associated with AST nodes and distribute *context information*. In *JastAddC* this is the *access support* concern's task. It realizes the distribution of *context information* across the AST, the preparation of the information stored in the AST (especially in terminal nodes) and the *non-null reference attributes* described in the preceding section.

To check the AST for semantical errors and to annotate it with appropriate *error objects* is the *constraint verification* concern's task.

The *error handling* concern's task is the support of functionalities to retrieve/construct appropriate *error nodes*, to associate error objects with AST nodes and to search the AST for associated *error objects*.

A few of *C's* language constructs implicitly specify AST parts, e.g. the standard defines, that the type of enumeration constants is `int` [6.4.4.3;2;p. 59]. To extend the AST automatically with such implicitly given parts is the *implicit information retrieval* concern's task.

For a few language parts the parser cannot construct the most appropriate AST, such that a suboptimal AST structure has to be used. However, a convenient AST structure is essential for comprehensible semantical specifications. Consequently, *JastAddC* distinguishes two AST structures: One AST structure for parsing and one for semantical computations. To transform automatically the first AST constructed by the parser into the final AST expected by semantical computations is the *AST mapping* concern's task.

5.2 *Core-Concerns*: Implementations

In the following section the modules implementing *JastAddC's compiler-core*, thus its *core-concerns* are investigated. Throughout the discussion a good understanding of *JastAdd's* specification language and ReCRAGs is required. The presented implementations are with no means complete. Instead the techniques, concepts and tasks typical for each concern's implementation are exemplified. If an overview of a module's complete functionalities is desired appendix B can be consulted. After reading this section the reader should know, how to implement common semantical tasks in a *specification-driven*, extensible way using *JastAdd* and the ReCRAG formalism.

It is worth mentioning, that the following three sections (the *error handling* concerns's, *access support* concerns's and *semantic verification* concern's im-

[13]Each AST node represents a context.

[14]The most common case for the distribution of *context information* also is the most simple one: The distribution of a node's context into its subtree. Thus, the broadcasting of the node into its subtree by an inherited *reference attribute*.

plementation) realize the generic error handling approach presented in section 5.1.2.

5.2.1 Error Handling Module

The *error handling* module realizes two tasks. Firstly, it supports methods to associate *error objects* with AST nodes and to search for associated *error objects*, such that the *constraint verification* module can augment a program's errors and the user can browse them. And secondly, it supports methods to retrieve for each AST node type a corresponding error node. These methods are required to implement *non-null reference attributes*. Hence, the *error handling* module provides functionalities needed to realize the generic error handling concept presented in section 5.1.2.

In *JastAddC*, *error objects* are represented by instances of the handcrafted `jastaddc.SourceError` class. Each *error object* has an error type, error level, unique error number, source file name, line number, column number and error description. The error type specifies the *compilation phase* which produced the error. The error level specifies, whether the error is harmless and can be ignored, is a warning or is a real error[15]. Error types, as well as levels, are represented by public enumeration constants specified in the `SourceError` class. `SourceError` objects implement the `java.lang.Comparable` interface. They are sorted with respect to their level, followed by type, source name, line, column and finally regarding the lexical order of their descriptions. Hence, it is no problem to assort collections of arbitrary *error objects*.

To support the augmenting of the AST with *error objects* is rather easy. The *error handling* module simply specifies, that each AST node has a list of *error objects* and the `addError(jastaddc.SourceError error)` method can be used to add an error to an AST node's *error list*. Errors are only added if the list does not already contain them. It is important to keep in mind, that the adding of an *error object* is a side effect, which is reflected by lowercase attribute names[16]. Listing 5.2 presents the corresponding implementation and listing 5.1 shows the according declarations specified in the *error handling core-concern*.

Listing 5.1: *Error handling* module: *Compiler-core* declarations for the association of *error objects* with AST nodes.

```
1  // Support to associate errors with AST nodes:
2  syn boolean ASTNode.addError(jastaddc.SourceError error);
3
4  // Support to access errors associated with AST nodes:
5  syn java.util.Iterator<jastaddc.SourceError> ASTNode.collectAllErrorsOfTree();
6  syn java.util.Collection<jastaddc.SourceError> ASTNode.collectAllErrorsOfNode
       ();
7  syn ASTNode ASTNode.findNodeInSubtreeErrorIsAssociatedWith(jastaddc.
       SourceError error);
```

[15] Real error: A violation against the language's specification.

[16] In *JastAddC*, the names of imperative attributes start lowercase in contrast to declarative ones. Throughout the thesis imperative attributes will be called methods. To specify imperative attributes is worse programming style. However, since the *compiler-core's* concept (the declaration of functionalities) cannot be supported for ordinary methods, it is sometimes a necessity.

Listing 5.2: *Error handling* module: Implementation realizing the association of *error objects* with AST nodes.

```
1  // Associate with each AST node a list of error objects:
2  protected java.util.ArrayList<jastaddc.SourceError> ASTNode.errors =
3          new java.util.ArrayList<jastaddc.SourceError>();
4
5  // Add an error object to the node's error list:
6  eq ASTNode.addError(jastaddc.SourceError error) {
7      if (error == null)
8          return false;
9      if (!errors.contains(error)) {
10         errors.add(error);
11         return true;
12     }
13     return false;
14 }
15
16 // Collect all errors associated with the tree spaned by the AST node:
17 eq ASTNode.collectAllErrorsOfTree() {
18     FlattingIterator result =
19         new FlattingIterator<jastaddc.SourceError>(
20                 errors.iterator());
21     for (int i = 0; i < getNumChild(); i++) {
22         result.addIterator(getChild(i).collectAllErrorsOfTree());
23     }
24     return result;
25 }
26
27 // Get all errors associated with the AST node:
28 eq ASTNode.collectAllErrorsOfNode() = errors;
29
30 // Search the tree spaned by the AST node for the node
31 // an error is associated with:
32 eq ASTNode.findNodeInSubtreeErrorIsAssociatedWith(jastaddc.SourceError error){
33     if (errors.contains(error))
34         return this;
35     for (int i = 0; i < getNumChild(); i++) {
36         ASTNode result =
37                 getChild(i).findNodeInSubtreeErrorIsAssociatedWith(error);
38         if (result != null)
39             return result;
40     }
41     return null;
42 }
```

To access *error objects* associated with AST nodes the `collectAllErrorsOfTree()`, `collectAllErrorsOfNode()` and `findNodeInSubtreeErrorIsAssociatedWith(jastaddc.SourceError error)` methods can be used. The `collectAllErrorsOfNode` method simply returns the node's *error list*. However, if all errors associated with a (sub-)tree are required, it is not efficient to return a collection containing them, since this would require to copy the *error list* of each node contained in the (sub-)tree. To provide an efficient solution, the `collectAllErrorsOfTree` method traverses the complete (sub-)tree and returns a spe-

cial iterator, which iterates above all the sub-tree's *error objects*. Similar, the `findNodeInSubtreeErrorIsAssociatedWith` method also traverses the (sub-)tree and checks each node's *error list*, whether it contains the given *error object* or not. If a node is encountered, whose *error list* contains the *error object*, the node is returned, otherwise `null`.

The *error handling* module's second task, the supply of error nodes, is required to realize the generic error handling concept's *non-null reference attributes*.

To support the availability of appropriate *error nodes* during the computations of *non-null reference attributes*, the *compiler-core* declares the `ConstructErrorNode(Class nodeType)` attribute, which takes a node type as argument[17] and returns a compatible *error node*. The `ConstructErrorNode` attribute is associated with the `ErrorBuilder` child node of the AST's root node[18]. Hence, always an *error node* is required the AST's `ErrorBuilder` node must be accessed. To do so the synthesized `ErrorBuilder()` attribute specified in the *compiler-core* can be used. Since the construction and management of *error nodes* is the error handling module's task, the *core-AST's* `ErrorBuilder` node type is abstract. Thus, the *error handling* module still has to support an appropriate implementation.

Last but not least, it must be possible to distinguish *error nodes* from their error free (ordinary) counterparts. Since the purpose of *error nodes* is to replace missed AST nodes in semantical computations, it is inconvenient to distinguish *error nodes* and their corresponding ordinary nodes by type. Instead, it is much better to introduce an inherited boolean attribute associated with each AST node, which decides, whether a node is an *error node* or not: The `IsErrorNode()` attribute.

Listing 5.3 summarizes the *error handling* module's *error node* functionalities specified in the *compiler-core*.

Listing 5.3: *Error handling* module: *Compiler-core* declarations for the recognition and supply of error nodes.

```
1  // Provide functionality to access the AST's root node:
2  syn Compilation ASTNode.CompilationRootNode() =
3      getParent().CompilationRootNode();
4  eq Compilation.CompilationRootNode() = this;
5  // Provide functionality to access the error handling's error node builder:
6  syn ErrorBuilder ASTNode.ErrorBuilder() =
7      CompilationRootNode().getErrorBuilder();
8
9  // Check, whether an AST node is an error node or not:
10 inh boolean ASTNode.IsErrorNode();
11
12 // Retrieve an appropriate error node for a given AST node type:
13 syn lazy ASTNode ErrorBuilder.ConstructErrorNode(Class nodeType);
```

[17]The node type is specified using a reflective `Class` object representing one of the classes *JastAdd* generated for the compiler's AST specification.

[18]The root node of *JastAddC* ASTs is the `Compilation` node.

What remains to show is, how the attributes of listing 5.3 are actually implemented.

The key idea of the *error handling's* actual implementation is to statically construct an AST consisting of ordinary AST nodes, which are interpreted as *error nodes*. This statically constructed AST is simply called *error node AST*. It is a child node of the `DefaultErrorBuilder` node type, which implements the *compiler-core's* abstract `ErrorBuilder`. The `ConstructErrorNode` attribute's equation just returns an appropriate node[19] contained in the *error node AST*. Hence, *error node* references (e.g. *non-null reference attributes* based on an unsatisfied semantical computation) of the input program's AST will reference nodes contained in the *error node AST*. Figure 5.1 illustrates this concept. Since

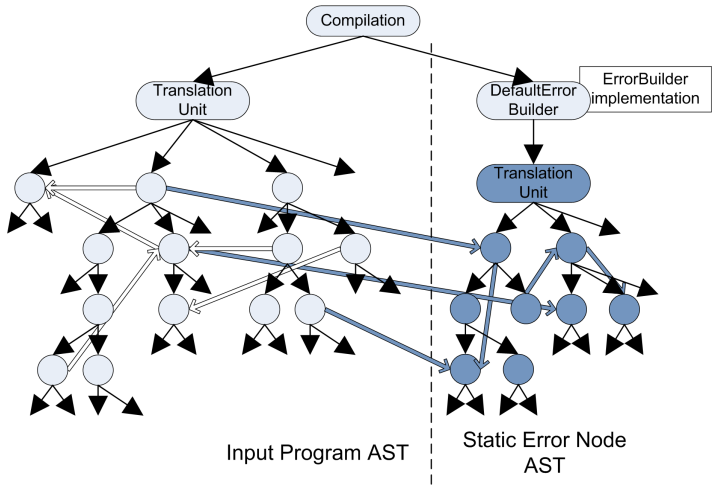

Figure 5.1: *Error handling* module: `DefaultErrorBuilder`, *error node AST* and *non-null reference attributes* pointing to *error nodes*. The *error node AST* is represented by dark blue nodes. The *non-null reference attributes* pointing to *error nodes* are represented by dark blue double-arrows.

the *reference attributes* of *error nodes* will also point into the *error node AST*, one can keep on computing semantics of *error nodes* as long as he likes, the computed results will alway represent errors again. Of course, the *error node AST's* semantical constraint checking has to be deactivated, such that no *error objects* are annotated to *error nodes*.

Listing 5.4 presents the *error handling* module's *error node* implementation.

Listing 5.4: *Error handling* module: Implementation realizing the recognition and supply of error nodes.

```
1  /**
2   * Specify, that nodes in the error node AST represent error nodes and deactivate
3   * the tree's validation and constraint checking.
4   */
5
6  // The root node is no error node:
```

[19] A node of the type given as argument in the `ConstructErrorNode` attribute's call.

```
7  syn boolean Compilation.IsErrorNode() = false;
8  // The TranslationUnit child of the error handling module's
9  // ErrorBuilder represents the error node AST:
10 eq Compilation.getErrorBuilder().IsErrorNode() = true;
11 // Of course, the input program's AST is no error node AST:
12 eq Compilation.getTranslationUnit().IsErrorNode() = false;
13 // Do not check semantical constraints in the error node AST:
14 eq DefaultErrorBuilder.IsCorrect() = true;
15
16 /**
17  * Provide error node construction functionalities :
18  */
19
20 // Collect all error nodes:
21 coll java.util.List<ASTNode> ErrorBuilder.ErrorNodes()
22     [new java.util.LinkedList<ASTNode>()] with add;
23 ASTNode contributes this when IsErrorNode() to ErrorBuilder.ErrorNodes()
24     for ErrorBuilder();
25
26 // Search the collection of all error nodes for a node of the required
27 // type and return it:
28 eq DefaultErrorBuilder.ConstructErrorNode(Class nodeType) =
29     ListFilter.filter(
30             ErrorNodes(),
31             new ASTNodeTypeFilter(nodeType)).get(0);
```

The attentively reader might have recognized, that the inherited `IsErrorNode` attribute is declared again for `Compilation` nodes. This is no error. Since `Compilation` is the AST's root node, it has no parent node. The synthesized `IsErrorNode` attribute simply overrides the inherited one's equation, such that the `IsErrorNode` attribute is well defined for `Compilation` nodes.

The `ConstructErrorNode` attribute's implementation presented in listing 5.4 uses *collection attributes*. The `ErrorBuilder` node has a collection containing all the *error nodes* of its *error node AST*. Hence, instead of traversing the *error node AST* to find an appropriate *error node*, simply the collection can be filtered. To filter lists, *JastAddC* supports a `ListFilter` class, whose `filter(-java.util.List<E> toFilter, AFilterCriteria<E> filterToUse)` method expects a list and a filter and returns a list containing all the elements satisfying the given filter. Consequently, the `ConstructErrorNode` attribute's implementation uses a special handcrafted class[20] to filter the `ErrorBuilder's` collection of *error nodes* with respect to a node type.

The last thing to show is, how the *error node AST* is constructed. To use a *non-terminal attribute* for the construction seems to be the best solution. Unfortunately, the current *JastAdd* version has problems to handle nodes, which contribute to *collection attributes* and are constructed by a *non-terminal attribute*. Such contributions are not performed. A workaround is to use an ordinary *synthesized attribute*, whose value is, similar to the *non-terminal attribute* approach, the *error node AST*. A rewrite can then simply set the `DefaultErrorBuilder's` *error node AST* to the one constructed by the *synthesized attribute*, if the `DefaultErrorBuilder` has not yet an *error node AST*. Hence, the

[20] `ASTNodeTypeFilter`

`DefaultErrorBuilder's` *error node AST* child must be optional. The *Beaver* parser simply does not set an *error node AST*, such that the initialization of the *error node AST* is automatically handled by the *error handling* module's implementation. Listing 5.5 presents this approach.

Listing 5.5: *Error handling* module: Initialization of the `DefaultErrorBuilder's` *error node AST*.

```
// Automatically initialize the error handling module
// (construct the error node AST):
rewrite DefaultErrorBuilder {
    when (!hasTranslationUnit())
    to DefaultErrorBuilder {
        setTranslationUnit(ConstructErrorTree());
        return this;
    }
}

// Attribute performing the actual construction of the error node AST:
syn lazy TranslationUnit DefaultErrorBuilder.ConstructErrorTree() =
    new TranslationUnit(
        ...
        new ExplicitLabeledStatement (
            new SwitchStatement(
                new Variable(new Identifier("[ERROR␣TREE␣VARIABLE]")),
                new EmptyStatement()),
            new Identifier("[ERROR␣TREE␣LABEL]")))
        ...
    );
```

5.2.2 Access Support Module

This section exemplifies the *access support* module's implementation. As example for the information preparation for further semantical computations the support of node addresses and taged type[21] names will be presented. Afterwards, support functionalities for switch statements will be shown, to demonstrate the distribution of *context information* and the implementation and usage of *non-null reference attributes*. Since *non-null reference attributes* are essential for the generic error handling concept presented in section 5.1.2, the *access support* module realizes another of the concept's parts.

For many semantical computations the information, whether

- an AST node is located in a subtree left to another node
- an AST node is located in a subtree right to another node
- an AST node is a successor of another node
- an AST node is an ancestor of another node

[21] structures, unions and enumerations

is required. E.g. in *C*, the *declare before use rule* is applied. Thus, entities must be declared before they can be used. Additionally, often unique AST node labels are needed. E.g. anonymous taged types need unique internal names, such that no redeclaration error happens. To support on the one hand the comparison of AST node locations and on the other hand unique AST node labels, the *Dewey notation* [8] for tree structures can be used[22]. To do so, the *access support core-concern* specifies the *inherited attribute* `NodeAddress()`, which is associated with each AST node and returns the node's *Dewey address* represented by an `jastaddc.semantics.NodeAddress` object. Hence, unique AST node labels are *Dewey addresses* and the comparison of AST node locations is accomplished by the comparison of their *Dewey addresses*. To compare the *Dewey addresses* of AST nodes is simple, since the handcrafted `NodeAddress` class implements the `java.lang.Comparable<NodeAddress>` interface. The comparison of two `NodeAddress` objects A_1 and A_2 associated with arbitrary AST nodes N_1 and N_2 returns

1 if N_1 is left to N_2 or N_2 is in the subtree of N_1

-1 if N_1 is right to N_2

0 if $N_1 = N_2$

Additionally, the boolean methods `isAncestor(NodeAddress otherAddress)` and `isSuccessor(NodeAddress otherAddress)` can be used to decide, whether N_1 is an ancestor/successor of N_2 or not.

Listing 5.6 presents the implementation of the *compiler-core's* inherited `NodeAddress` attribute.

Listing 5.6: *Access support* module: Implementation for the support of unique, comparable node addresses.

```
1  // The AST root node has address 1:
2  syn jastaddc.semantics.NodeAddress Compilation.NodeAddress() =
3      new jastaddc.semantics.NodeAddress().addAddressPart(1);
4
5  // Any other node has the address of its parent node
6  // extended by its child index:
7  eq ASTNode.getChild(int index).NodeAddress() =
8      NodeAddress().clone().addAddressPart(index + 1);
```

The `NodeAddress` attribute's implementation reflects the formal definition of *Dewey addresses*. The root node has address 1. Each other node has the address of its parent node extended by its position in its parent node's list of children. The `NodeAddress' clone(NodeAddress toClone)` method constructs a deep copy and the `addAddressPart(int part)` method[23] extends its current address by appending the given value[24].

[22] E.g. an AST node with the address 1.3.2 is left to an AST node with the address 1.4.

[23] For convenient reasons, the `addAddressPart` method returns the `NodeAddress` object used to invoke it.

[24] E.g. if an `NodeAddress` object represents the address 1.3.2 and `addAddressPart(5)` is performed its new address is 1.3.2.5.

Another typical example for the preparation of the AST's information for further semantical computations are taged type names. Since *C* permits the declaration of anonymous taged types, not every taged type has a name. However, for further semantical computations it would be convenient if each anonymous taged type has an unique name, such that a taged type's name can be accessed without bothering, whether the taged type is anonymous or not. Hence, instead of directly accessing the optional identifier representing a taged type's name, an attribute providing the name is used: The *synthesized attribute* `Name()` associated with each `TagedType` AST node. The `Name` attribute simply checks, whether a taged type has a name or not. If the taged type is not anonymous its name specified in the input program is returned. Otherwise, an unique dummy name is constructed based on the `TagedType` node's *Dewey address*. Listing 5.7 presents the `Name` attribute's implementation.

Listing 5.7: *Access support* module: Implementation handling the access of taged type names.

```
1 eq TagedType.Name() = hasTag() ?
2     getTag().Name() :
3     "[Anonymous␣Taged␣Type␣"+ NodeAddress().toString() +"]";
```

Interesting about this solution is, that it is side effect free. There exists no special entity with an internal state which keeps track of the names already constructed.

The last *access support* example is about functionalities to associate switch labels with their corresponding switch statements. It illustrates the *access support* concern's task to distribute simple *context information* across the AST and to realize *non-null reference attributes*. The attributes used to present these tasks are:

`FindEnclosingSwitchStatement()` *Inherited attribute* of `Statement` nodes. It returns the switch statement[25] currently in scope. If no switch statement is in scope `null` is returned. The `FindEnclosingSwitchStatement` attribute is a typical example for the distribution of *context information* by broadcasting a node's context into its subtree.

`AssociatedSwitchStatement()` Synthesized *non-null reference attribute* of `SwitchLabelStatement` nodes[26]. It returns the switch statement the label is associated with. If no switch statement is in scope an *error node* is returned.

`AssociatedLabels()` *Collection attribute* of `SwitchStatement` nodes. It contains all the labels associated with the switch statement.

Listing 5.8 presents the declarations of these attributes in the *compiler-core's access support* concern and listing 5.9 illustrates their implementation.

Listing 5.8: *Access support* module: *Compiler-core* declarations for the distribution of the switch statement currently in scope, the association of each switch

[25] Represented by a `SwitchStatement` node.

[26] *C's* `case` and `default` labels are represented by AST node types which are sub classes of `SwitchLabelStatement`.

label to a switch statement and the collection of a switch statement's switch labels.

```
1  // Semantical computation to broadcast the switch statement currently
2  // in scope (returns null, if no switch statement is in scope):
3  inh SwitchStatement Statement.FindEnclosingSwitchStatement();
4
5  // The corresponding non-null reference attribute for switch labels:
6  syn SwitchStatement SwitchLabelStatement.AssociatedSwitchStatement();
7
8  // Associate with each switch statement a collection containing all its labels:
9  coll java.util.List<LabeledStatement> SwitchStatement.AssociatedLabels()
10     [new java.util.LinkedList<LabeledStatement>()] with add;
```

Listing 5.9: *Access support* module: Implementation of the *access support* concern's switch statement functionalities.

```
1  // Each switch statement broadcasts itself as the current
2  // switch statement in scope into its subtree:
3  eq TranslationUnit.getDeclaration().FindEnclosingSwitchStatement() = null;
4  eq FunctionDefinitionInitializer.getBody().FindEnclosingSwitchStatement()= null;
5  eq SwitchStatement.getBody().FindEnclosingSwitchStatement() = this;
6
7  // Non-null reference attribute, which returns the switch statement
8  // currently in scope or a switch statement error node:
9  eq SwitchLabelStatement.AssociatedSwitchStatement() =
10     FindEnclosingSwitchStatement() != null ?
11     FindEnclosingSwitchStatement() :
12     (SwitchStatement)ErrorBuilder().ConstructErrorNode(
13             SwitchStatement.class);
14
15 // Transparent usage of the AssociatedSwitchStatement()
16 // non-null reference attribute to support for each switch
17 // statement a collection of its switch labels:
18 SwitchLabelStatement contributes this
19     to SwitchStatement.AssociatedLabels()
20     for AssociatedSwitchStatement();
```

The interactions between these attributes reflect the generic error handling approach presented in section 5.1.2. The `AssociatedSwitchStatement` *non-null reference attribute* lifts undefined results of a switch label node's `FindEnclosingSwitchStatement` attribute to defined values, such that further semantical computations can be implemented without bothering about invalid programs, where a switch label is not embedded in a switch statement[27]. Consequently, switch label nodes contributing to a switch statement's `AssociatedLabels` *collection attribute* can simply use their `AssociatedSwitchStatement` attribute to find the switch statement to contribute themself to. If a switch label is not embedded into a switch statement, it will contribute itself to a switch statement *error node*. Since the `ConstructErrorNode(SwitchStatement.class)` attribute invocation in line 12 and 13 will always return the same *error node*, a nice side effect of the presented approach is, that all switch labels not embedded in a switch statement will be collected at the same *error node*.

[27] E.g. the program: `int main(void){case:; return 1;}`

5.2.3 Constraint Verification Module

This section presents the *constraint verification* concern's implementation, which realizes functionalities to traverse the AST, check it for unsatisfied semantic constraints and annotate it with appropriate *error objects*. Hence, the *constraint verification* module realizes the last part of the generic error handling concept presented in section 5.1.2.

The *constraint verification* concern's tasks are represented by two *synthesized attributes*, both associated with every AST node type and returning a boolean value[28]:

`IsCorrectLocal()` Checks an AST node's constraints, annotates the node with appropriate *error objects* and returns true if all semantic constraints are met and false otherwise.

`IsCorrect()` Similar to the `IsCorrectLocal` attribute, except not only the AST node itself is checked, but also its complete subtree.

The implementation of both attributes is straightforward. The `IsCorrect` attribute can be implemented in a generic way. It simply traverses the AST and calls each encountered node's `IsCorrectLocal` attribute. If any `IsCorrectLocal` call returns false the `IsCorrect` attribute also does. Otherwise, it returns true. Listing 5.10 presents the `IsCorrect` attribute's implementation.

Listing 5.10: *Constraint verification* module: Implementation of the `IsCorrect()` attribute.

```
1  // Check all semantic constraints of the node and its subtree:
2  eq ASTNode.IsCorrect() {
3      boolean result = true;
4      if (!IsCorrectLocal())
5          result = false;
6      for (int i = 0; i < getNumChild(); i++) {
7          if (!getChild(i).IsCorrect())
8              result = false;
9      }
10     return result;
11 }
```

As shown in listing 5.10, it is important, that the `IsCorrect` attribute finishes its traversal, even if an error has been recognized. Otherwise, semantic errors encountered during the further traversal would not be annotated to the AST. All the semantic constraints of the tree, the AST node from which the traversal started spans, must be checked.

The `IsCorrectLocal` attribute cannot be implemented in a generic way, since the semantic constraints to check for a node depend on its type. To receive an impression about `IsCorrectLocal` attribute equations consider listing 5.11, which presents the implementation for switch labels.

Listing 5.11: *Constraint verification* module: Implementation of the `SwitchLabelStatement` node type's `IsCorrectLocal()` attribute.

[28] Of course, these two attributes are declared in the *compiler-core*.

```
1  eq SwitchLabelStatement.IsCorrectLocal() {
2      // Always check the inherited semantic constraints:
3      boolean result = super.IsCorrectLocal();
4      if (AssociatedSwitchStatement().IsErrorNode()) {
5          // Construct an error object with an appropriate description:
6          jastaddc.SourceError error = new jastaddc.SourceError(
7              "Control␣Flow:␣Switch␣label." +
8              "\n\tThere␣exists␣no␣enclosing␣switch␣statement.");
9          // Initialize the error object:
10         error.line = line();
11         error.column = column();
12         error.sourceName = sourceName();
13         error.errorLevel = jastaddc.SourceError.ErrorLevel.ERROR;
14         error.errorType = jastaddc.SourceError.ErrorType.SEMANTICAL;
15         // Augment the AST node with the error object:
16         addError(error);
17         result = false;
18     }
19     return result;
20 }
```

The only semantic constraint to check for switch labels is specified in line four. There, the `AssociatedSwitchStatement` *non-null reference attribute* introduced in section 5.2.2 is used to check, if the switch label is embedded in a switch statement. If it is not, an appropriate error is constructed. Much more interesting is the `super` call in line three. Since `SwitchLabelStatement` inherits form `LabeledStatement`, a `SwitchLabelStatement` has all the semantics a `LabeledStatement` has and consequently also all its semantic constraints. `LabeledStatement` itself inherits from `Statement` which inherits from `ASTNode`[29]. The `super` call guarantees, that the `SwitchLabelStatement` must satisfy all its super class's semantic constraints. It is obvious, any `IsCorrectLocal` attribute equation should always call its super class's `IsCorrectLocal` attribute, as done in listing 5.11, such that all the semantic constraints associated with the node type are checked.

5.3 Generic Mapping Technique

In this section a technique will be presented, which maps an AST constructed by a parser to an AST much more suitable for semantic analysis. The technique will be described in such a way, that it becomes apparent how to apply it in other compiler projects using ReCRAGs. Hence, the technique will be presented in a generic way, whereas *C* is used to exemplify its application and advantages. The specifications of *JastAddC's AST mapping* concern will not be presented. Readers interested in the technique's implementation for the *JastAddC* compiler can consult *JastAddC's* source code.

The next section will motivate the need for such a technique and the next but one section will present it.

[29] In *JastAddC*, the `ASTNode` node type's `IsCorrectLocal` attribute performs a few general semantic and consistency checks, all AST nodes must satisfy.

5.3.1 Generic Mapping Technique: Motivation

In the following it will be shown, that the trees parsers construct are often suboptimal for semantic analysis. Even if it is possible for parsers to construct trees abstracted a lot from the concrete syntax the parser is based on[30], it might be impossible to eliminate all complicated concrete syntax structures. Hence, the constructed tree might be hard to understand for human readers and consequently unsuitable for the specification of semantics. The following discussion will illustrate, why *C's* syntax results in ASTs uncomfortable for semantic analysis and account the need for a technique to map the AST constructed by the parser to another AST structure which is better suited for semantic analysis.

The standard's most complicated CFG parts handle types and declarations (see appendix A part A.2.2). E.g. consider the declaration:

$$\texttt{int} * (*(\texttt{func}(\texttt{long} * \texttt{a}, \texttt{void}(*(\texttt{b}[\texttt{3}]))(\texttt{void}))))(\texttt{long}, \texttt{void} * (*)(\texttt{void}))$$

Its standard conforming syntax tree is shown in figures 5.2 and 5.3 and the AST *JastAddC's* parser constructs for it is shown in figures 5.4 and 5.5. For now, it is not necessary to understand the declaration. Its type will be discussed later on. Instead, it is **important**[31] to remember, that *JastAddC's* parser is based on the standard's syntax (see chapter 4.2) and to recognize, that both trees have similar structures. Of course, the tree constructed by *JastAddC's* parser is more abstract. Many chain rules are eliminated and most tokens are omitted, since they are only needed for parsing purposes. Also, nodes are typed, such that their types reflect the language constructs they represent. The tree constructed by the parser is an AST. However, the declaration it represents, especially its type, is extremely complicated to read. And as will be shown in the following, this is the case, because the syntax the parser is based on[32] is not suited for semantical computations and it is impossible for the parser to bridge the gap between its syntax and a well suited one.

To back this claim, consider again the declaration:

$$\texttt{int} * (*(\texttt{func}(\texttt{long} * \texttt{a}, \texttt{void}(*(\texttt{b}[\texttt{3}]))(\texttt{void}))))(\texttt{long}, \texttt{void} * (*)(\texttt{void}))$$

To read its type in the AST *JastAddC's* parser constructed[33] the reader must travers it in the order presented by the labels 1 to 21. Its type is (each bolded number denotes the labeled node in figure 5.4 or 5.5, which represents the type information):

1 `func` is a
2 Function with two parameters and returning
  3 A pointer to
  4 A function with two parameters and returning

[30] E.g. elimination of chain rules, discarding of parsing control characters like `{`,`}`,`(`,`)`,`=`,`+`,`-`,...

[31] I highly suggest to take a few minutes to compare and investigate the standard conforming tree and the one *JastAdd's* parser constructed! It is a good idea to print figures 5.2, 5.3, 5.4, 5.5 and 5.6 on different sheets, such that there is no need for page flipping throughout the following discussion. THANKS!

[32] The standard's syntax. See chapter 4.2

[33] figures 5.4 and 5.5

5 A pointer to
6 `int`
Its first parameter is
7 Of type `long`
Its second parameter is
8 A pointer to
9 A function with one parameter and returning
10 A pointer to
11 `void`
Its first parameter is
12 Of type `void` (the function has no parameter at all)
Its first parameter is
13 Named `a` and of type
14 Pointer to
15 `long`
Its second parameter is
16 Named `b` and of type
17 Array of
18 Pointers to
19 A function with one parameter and returning
20 `void` (the function returns nothing)
Its first parameter is
21 Of type `void` (the function has no parameter at all)

As can be easily seen[34], to read the declaration's type it is necessary to jump through the tree. The type information associated with an AST node may be contained in tree parts above, left, right or below[35] the node and even worse a combination thereof. E.g. a function's type may consist of tree parts below the function-declarator[36], above the function-declarator, left to it and right to it, as presented by the labels 3 to 12[37] (return type), 13 to 15[38] (first parameter) and 16 to 21[39] (second parameter). However, for semantic specifications and the comprehensibility of the AST it would be much more convenient if the type associated with an AST node would be represented by the node itself, i.e. the tree the node spans.

To underpin the advantages of a more elaborated tree consider figure 5.6. It presents the AST *JastAddC's* semantical computations expect for the discussed declaration. Again, nodes are labeled corresponding to the type description presented above. It is obvious, the type is much more easy to read. In fact, the type of each AST node is completely represented by the tree the node spans. Hence, a node's type can be read in a depth-first manner. Since computations to find a node's type information are not required anymore, semantics are much easier to implement, not to mention the consequential readability advantages.

[34] If one follows the reading path presented in figures 5.4 and 5.5.
[35] The tree spanned by an AST node is below it.
[36] *JastAddC's* function-declarator models a direct-declarator, which represents a function. See appendix A rule 6.7.5 rule alternatives 7 and 8.
[37] below: 3; above: 4; left: 5, 6; right: 7 - 12
[38] below
[39] below

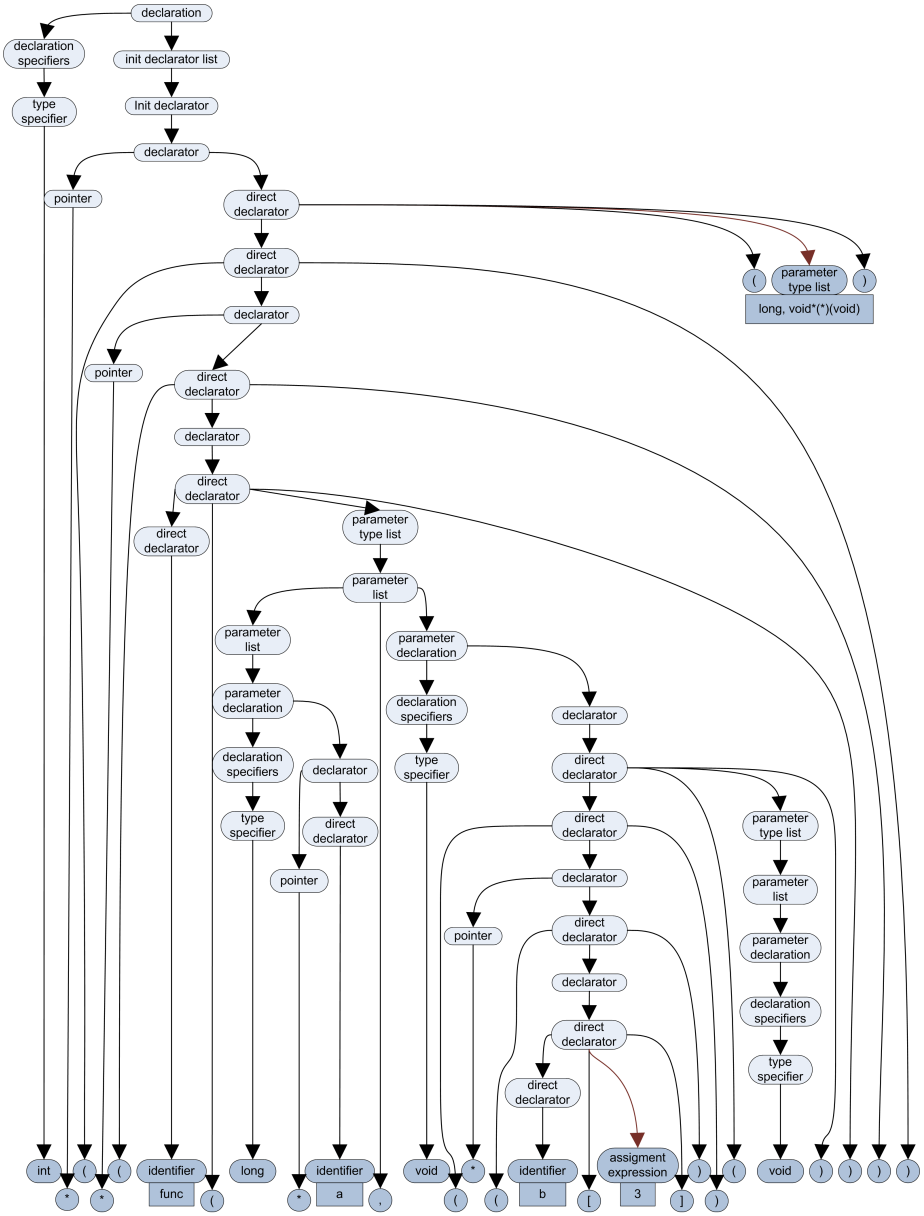

Figure 5.2: The standard's syntax tree for a complicated *C* declaration (part I).

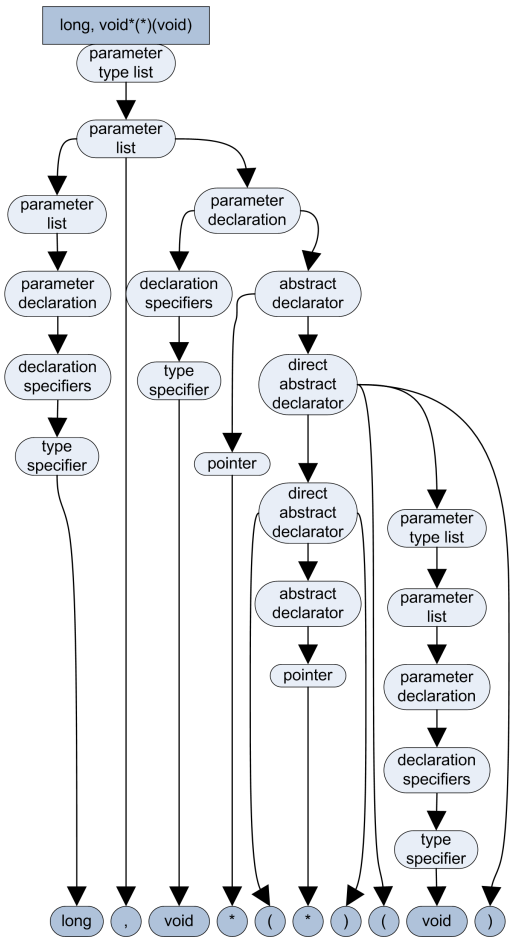

Figure 5.3: The standard's syntax tree for a complicated *C* declaration (part II).

What remains to show is, why the parser does not construct a better AST in the first place. As mentioned above, the syntax the parser is based on is the standard's one. Additionally, it has been noted, that the standard conforming syntax tree[40] and the AST the parser constructed[41] have a similar structure. If one labels the standard conforming syntax tree in the same way as the parser's one, it becomes obvious, that the jumps through the tree to find the next type parts are similar. Hence, if a parser would like to construct the AST presented in figure 5.6, it would have to perform the computations presented by these jumps, i.e. it must find a node's type information. However, as mentioned above, a node's type information is scattered across the tree, especially the type a node represents may be contained in tree parts below, above, left and right to the node at the same time. Though, the parser reads the input from left to right and constructs the AST bottom up. Consequently, the parser cannot

[40] Figures 5.2 and 5.3.
[41] Figures 5.4 and 5.5.

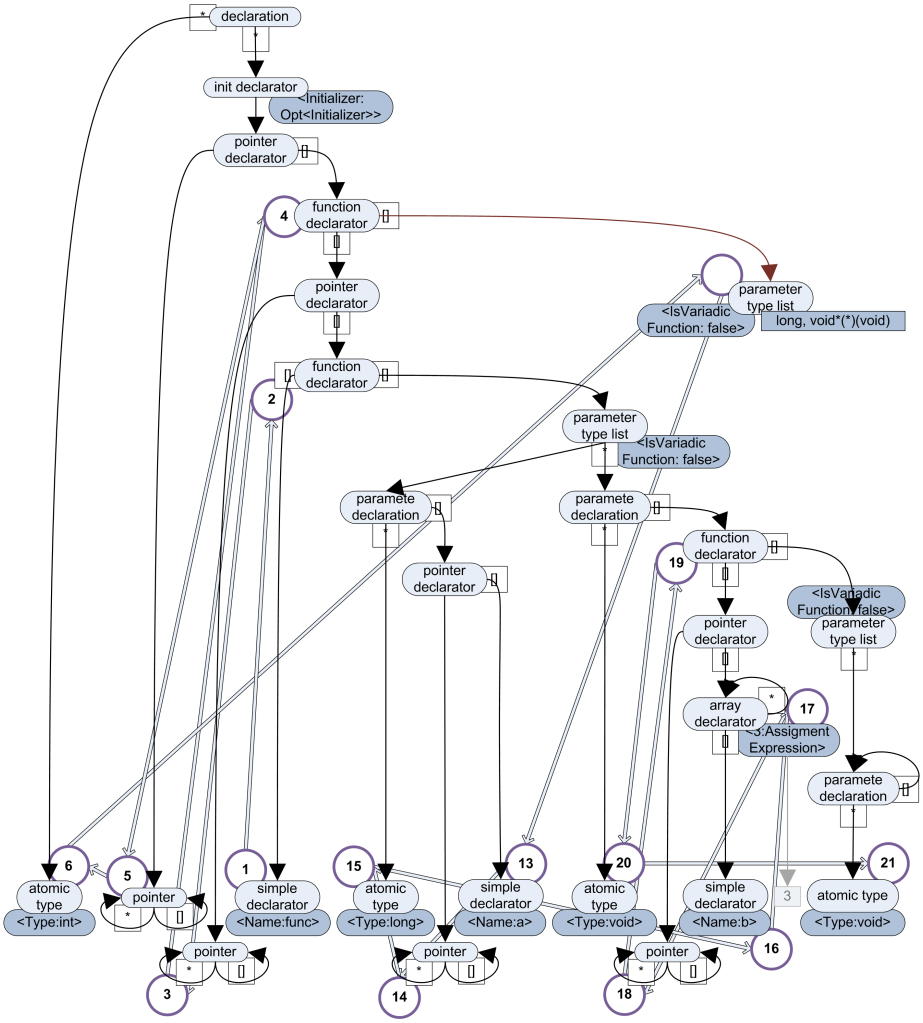

Figure 5.4: *Concrete to AST mapping* module: The AST constructed by *JastAddC's Beaver* parser for a complicated *C* declaration (part I). The `[]` and ∗ notes remark *JastAdd* list and optional children. Dark blue rounded rectangles represent terminal nodes. Number labels refer to the text's type description and double arrows show the path one has to follow to read the declaration's type.

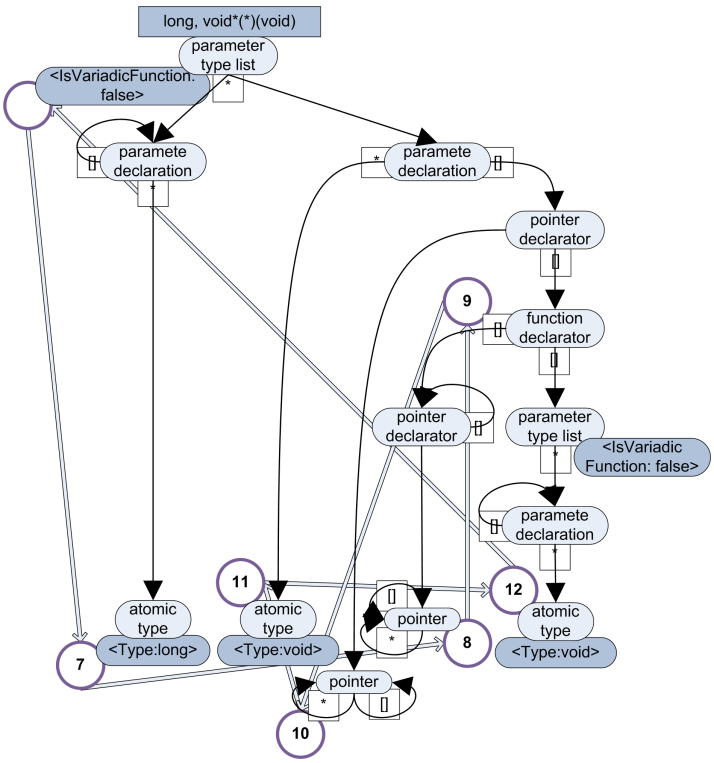

Figure 5.5: *Concrete to AST mapping* module: The AST *JastAddC's Beaver* parser constructs for a complicated *C* declaration (part II).

simulate the computations required to construct the AST presented in figure 5.6. Even if a *bottom-up* parser supports synthesized and *inherited attributes* the construction of the AST presented in figure 5.6 would require some kind of complicated "hack", since the construction of an AST node representing a type would have to be postponed until all the node's type information[42] are reduced and pushed on the stack. Similar problems are encountered if *top-down* parsing is used. Thus, there is a need for a mapping technique, which automatically maps ASTs constructed by the parser to ASTs the semantical analysis expects.

It may look strange, that on the one hand *JastAddC's* AST has a structure very well suited for semantic specifications and on the other hand figures 5.4 and 5.5 present an AST constructed by *JastAddC's* parser[43]. In fact *JastAddC* distinguishes two different AST structures. The parser constructs a first AST, which the *compiler-core's AST mapping* concern will automatically transform into the AST all other semantic specifications expect. Hence, the *AST mapping* concern realizes a *demand-driven* technique based on ReCRAGs, which maps AST structures. The next section will present this generic technique.

[42]Beware, the node's type information are not its context information. Quite the contrary, the actual problem is, that a node's type is represented by nodes not contained in the node's context and these nodes are scattered "around" the node.

[43]As explained in chapter 2.3.1 and 4.2, it is the parser's task to construct the *JastAdd* AST used for semantic analysis.

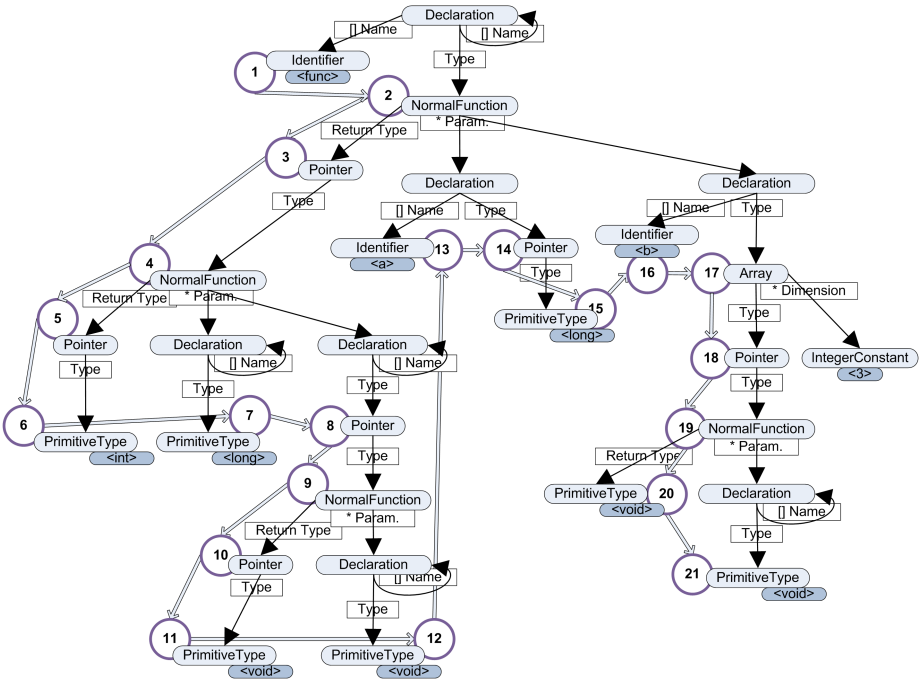

Figure 5.6: *Concrete to AST mapping* module: The AST *JastAddC's* semantical computations expect for a complicated *C* declaration. The *JastAdd* list and optional AST nodes for type qualifier, storage class specifier, function specifier and initializer are left for convenient reasons. They are missed anyway, since the declaration has no such constructs. Quite the contrary, if the declaration would have such constructs the AST would be even more convenient than the one presented in figures 5.4 and 5.5.

5.3.2 Generic Mapping Technique: Realization

This section presents a generic mapping technique, which can be used to transform in a well-founded way ASTs of a structure A in ASTs satisfying structure B. Hence, the presented mapping technique can be used for the automatically and transparently transformation of ASTs constructed by the parser to ASTs the semantical analysis expects. The section is separated in three parts. First, the technique and its constructs, terms, tasks and workflow is informally described. Afterwards, a formal specification of the technique and its properties is presented. And finally, a few interesting remarks about the technique's specification are given.

Generic Mapping Technique: Informal Presentation

Throughout the following informal discussion the mapping of ASTs constructed by a parser to ASTs the semantic analysis expects is used as abstract example scenario. Hence, two kinds of AST nodes are distinguished. Firstly, AST nodes representing a context as expected by the semantic analysis and secondly, AST nodes representing an intermediate context, which is unknown for the semantic analysis and still must be mapped. The first kind of node type is called *final node* and the second kind *intermediate node*. Both kinds must be disjunct, to reflect the fact, that either a context is known to the semantic analysis and its semantics can be evaluated or it is unknown and must be mapped. Figure 5.7 illustrates the mapping techniques application.

Since the parser may be able to construct for some parts of the input program the desired AST and for other parts only an intermediate tree, both kind of types are mixed and nested in each other at the beginning and during the mapping. Such mixed trees are called *final ASTs*, whereas the special case of a tree only consisting of *intermediate nodes* is called *intermediate AST* and a tree only consisting of *final nodes* is called *complete final AST*. Of course, the objective of the mapping is to retrieve automatically and transparently a *complete final AST*.

The actual mapping happens at *mapping nodes*, which span trees of *intermediate nodes*. To be able to represent the nesting of intermediate and final AST parts the parser constructed, *intermediate nodes* can have other AST parts, spanned by a *final node*, as terminal child nodes, so called *final AST terminal nodes*. The mapping of a *mapping node* inserts such *final AST terminal nodes* at the right place in the tree constructed for the *mapping node*. Because of the mentioned nesting of final and intermediate parts, the AST part represented by a *final AST terminal node* may again contain *intermediate nodes*. These nodes will be mapped in further mapping steps. Hence, the mapping itself will be incremental. Intermediate tree parts are replaced mapping by mapping with more refined AST structures until a *complete final AST* is constructed.

To realize the mapping, each *mapping node* has an attribute which computes, depending on the *mapping node's* subtree and its attributation, the mapped tree replacing it. These attributes are called *final AST computation attributes*. The replacement is simply an unconditional rewrite, a *mapping rewrite*, which replaces the *mapping node* and its subtree by the value of its *final AST computation attribute*. To guarantee, that the replacement is possible, i.e. the mapped tree fits in its parent node's context, *relationship node types* are used. Such node

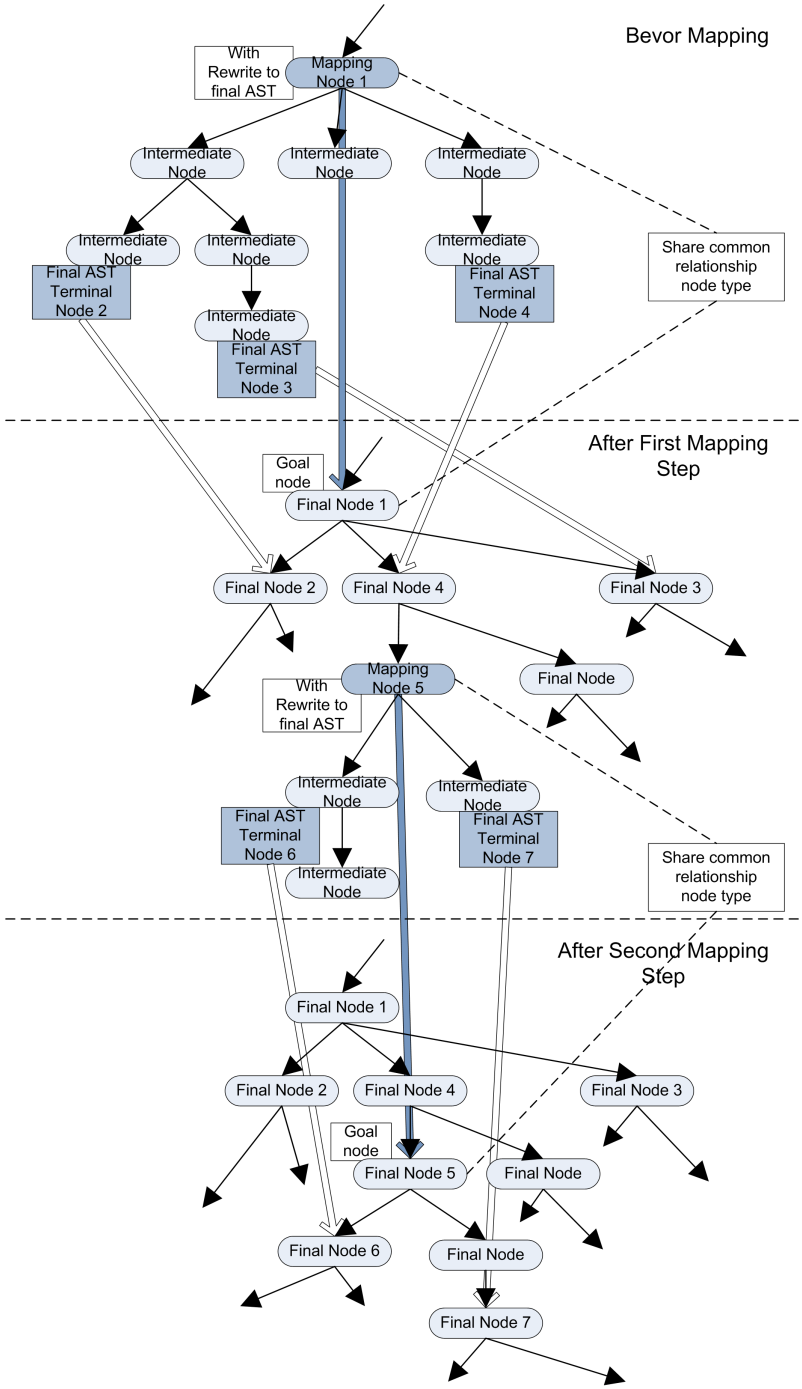

Figure 5.7: Mapping technique: Imaginary example for the mapping of a *final AST* to its corresponding *complete final AST* in two steps. Dark blue nodes are *mapping nodes*. Each *mapping node* and its subtree is mapped to the tree spanned by the node a dark blue double arrow points to. White double arrows represent the reusing of nested program parts, for which the parser has already been able to construct (at least for the root node spanning the tree representing the part) an appropriate tree. Such *final AST terminal nodes* are placed into the mapped tree by the mapping at the place, the white double arrow points to.

types are always used in the final AST's structure specification, when an AST, which has to be mapped, occurs as a *final node's* child node. Each *mapping node* inherits from a *relationship node type* R and replaces itself by a *final node*, which also inherited from R: A so called *goal node*. Consequently, *goal nodes* inheriting from R represent the nodes to which *mapping nodes* inheriting from R will be mapped to.

Generic Mapping Technique: Formal Specification

In the following, the actual generic technique realizing the so far informal presented AST mappings will be specified. The specification consists of a set of definitions, each handling a mapping construct and it's tasks. Surely, the definitions are hard to understand at first. To get used to them, it is a good idea to compare them with the informal introduction. Last but not least, it is highly suggested to comprehend the proofs following them, since they clarify the purpose and implications of the definitions.

Of course, the first definition coalesces all the other definitions and represents the actual mapping technique:

Definition MT.D.1 (ReCRAG AST mapping) *A* ReCRAG AST mapping κ *specifies*

1. *an* AST structure $\triangle$ *for* complete final ASTs *consisting of* final nodes, relation ship node types *and* goal nodes
2. *an* AST structure $\bigtriangledown$ *for* intermediate ASTs *consisting of* intermediate nodes *and* mapping nodes, *whereas the root nodes of* $\bigtriangledown$ *are* mapping nodes.
3. *the* mapping rewrites *and* final AST computation attributes *of* mapping nodes

in such a way, that the definitions MT.D.2 to MT.D.11 are satisfied. κ*'s input is a* final AST *satisfying the AST structure* $\bigtriangledown \cup \triangle$. κ*'s* mapping rewrites *transform stepwise the input AST. Each* mapping rewrite *results in a transformed AST, such that a chain* $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$ *of ASTs results. The execution of a* mapping rewrite *is called mapping step and can be written down using the left-associative binary infix operator* $\succ$. $\Gamma_i \succ_i \Gamma_{i+1}$ *means, that the mapping step* $\succ_i$ *transforms the AST* Γ_i *into the AST* Γ_{i+1}. *And* $\Gamma \succ^* \Omega$ *means, that the AST* Γ *can be transformed into the AST* Ω *in arbitrary many mapping steps.*

The other definitions are ordered, starting with the description of mapping constructs and ending with the specification of tasks which manipulate these constructs.

Definition MT.D.2 (intermediate node/final node) *Let* S_i *be the set of* intermediate nodes *and* S_f *be the set of* final nodes. *It holds that:* $S_i \cap S_f = \emptyset$. *Let* P_n *be the set of node types a node* n *inherits from (transitive inheritance relationship).* $\forall n \in S_f \forall p \in P_n$ *it holds that:* $p \in S_f$. *Let* $IsMappingNode(n)$ *be the function, which takes a node* n *and returns true iff* n *is a* mapping node. $\forall n \in S_i \wedge \neg IsMappingNode(n) \forall p \in P_n$ *it holds that:* $p \in S_i$.

Definition MT.D.3 (complete final AST) *A* complete final AST *is an AST with an arbitrary number of* final AST nodes.

Definition MT.D.4 (intermediate AST) *An* intermediate AST *is an AST consisting of an arbitrary number of* intermediate nodes *and exactly one* mapping node*, which is the* intermediate AST's *root node.*

Definition MT.D.5 (final AST) *A* final AST *is an AST consisting of an arbitrary number of* final nodes *and* intermediate ASTs.

Remark: MT.D.5 includes the special case, that a *final AST* is an *intermediate AST* or a *complete final AST.*

Definition MT.D.6 (final AST terminal node) *A* final AST terminal node *is a terminal child of an* intermediate node *whose value is a* final AST.

Definition MT.D.7 (intermediate to final AST relationship node type) Intermediate to final AST relationship node types *are also simply called* relationship node type. *A* relationship node type *is an abstract AST node type, which represents a* final node*, from which at least one node type representing a* mapping node *and exactly one node type representing a* goal node *directly inherit.*

Definition MT.D.8 (goal node) *A* goal node *is a* final node *inheriting from a* relationship node type.

Definition MT.D.9 (mapping node) *A* mapping node M *is an* intermediate node*, which inherits from exactly one* realtionship node type R*, has exactly one* mapping rewrite *and exactly one* complete final AST computation attribute. *Its* mapping rewrite *rewrites* M *to a* goal node G *of type* R *and its* final AST computation attribute's *value is of* G'*s type. The tree* Δ*,* M *spans, must be consumed by its* mapping rewrite*, i.e. the tree returned by its* final AST computation attribute *must not contain any of* Δ*'s* non-terminal *nodes.*

Definition MT.D.10 (mapping rewrite) *A* mapping rewrite *is an unconditional rewrite associated with a* mapping node M*, which automatically and transparently rewrites* M[44] *(in a* demand-driven *way) to the tree returned by* M'*s* final AST computation attribute.

Definition MT.D.11 (final AST computation attribute) *A* final AST computation attribute *is a parameterless* synthesized attribute *associated with a* mapping node M*, whose equation only depends on the tree spanned by* M*.* Final AST computation attributes *must not manipulate the AST's structure. Additionally, they must be declarative and side-effect free*[45]*.*

In the following the properties correctness, confluence and termination of *ReCRAG AST mappings* will be shown. However, before doing so, a few lemmas

[44] And consequently its subtree.
[45] Otherwise, the mapping's confluence property presented later on is lost.

stating interesting properties are introduced. The first lemma guarantees a very nice property, already assumed in section 5.3.1, where an example mapping for a complicated declaration has been shown. It would be nice, if the mapping of the declaration does not change the tree containing it. Lemma MT.L.1 ensures this:

Lemma MT.L.1 *(Mapping steps only manipulate the tree spaned by the mapped* mapping node*.)* Trivial, since each mapping rewrite (hence, mapping step) replaces its *mapping node M* by the AST returned from the *mapping node's final AST computation attribute* (MT.D.10; MT.D.11; MT.D.9) and these attributes only depend on the tree spanned by M (MT.D.11).

The next lemma is very important and has serious implications, even it may sound trivial at first. Most other proves are based on it.

Lemma MT.L.2 *(Any AST* Δ *occurring during a* ReCRAG AST mapping κ *will be a* final AST*.)* Assume a tree Δ occurs during the mapping. To prove the lemma, three cases have to be considered:

Δ consists only of *final nodes* Proved by definition of *final AST*.

Δ consists only of *intermediate nodes* It has to be proved, that Δ is an *intermediate AST* (MT.D.5). However, since each mapping replaces a *mapping node* by a *goal node* (MT.D.9), which is a *final node* (MT.D.8), no mapping has been executed yet and Δ is the input tree. Hence, Δ must be an *intermediate AST*, since κ is a *ReCRAG AST mapping* (MT.D.1).

Δ consists of an arbitrary number of *intermediate* and *final nodes* Since *intermediate ASTs* do not contain *final nodes* (MT.D.4), it must be proved, that Δ's *final nodes* only have *final nodes* as parent nodes (MT.D.5). Assume Θ is a valid *ReCRAG AST mapping* tree. Let M be any *intermediate node*, which has a *final node* as parent node. Since Θ is a valid *ReCRAG AST mapping* tree, M must be a *mapping node* (MT.L.2; MT.D.1; MT.D.4). The mapping replaces M by a *goal node* (MT.D.9). Since the mapping leaves the rest of Θ correct (MT.L.1) and *goal nodes* are *final nodes* (MT.D.8) the resulting tree is again a *final AST*.

Hence, starting from a valid *ReCRAG AST mapping* tree only *final ASTs* will occur during the mapping.

Immediate implications of lemma MT.L.2 are:

Lemma MT.L.3 *(*Intermediate nodes *are contained in an* intermediate AST*.)* Trivial, since only *final ASTs* occur during a mapping (MT.L.2).

Lemma MT.L.4 *(Mappings never manipulate* final nodes*.)* Trivial, since only *final ASTs* occur during a mapping (MT.L.2) and mapping steps only manipulate the tree spanned by the mapped *mapping node* (MT.L.1).

Lemma MT.L.4 is quite important, since it also implies, that mappings do not evaluate attributes of *final nodes*, if the attributes of *final AST terminal nodes*

are not accessed.

To prove correctness of *ReCRAG AST mappings*, i.e. if a *ReCRAG AST mapping* terminates the AST is a *complete final AST* and each mapping step only results in an AST satisfying the specified AST structure, is rather easy.

Proof MT.P.1 *(If a* ReCRAG AST mapping *terminates, the AST is a* complete final AST *and each mapping step only results in an AST satisfying the specified AST structure.)* The prove consists of two parts, MT.P.1.1 and MT.P.1.2:

Proof MT.P.1.1 *(Iff a* ReCRAG AST mapping *terminates, the AST is a* complete final AST.*)* Since *intermediate nodes* are only contained in *intermediate ASTs* (MT.L.3), whose root node must be a *mapping node* (MT.D.4), *mapping nodes* will be rewritten to *final nodes* (MT.D.9; MT.D.8), in doing so their spanned tree is consumed (MT.D.10), *mapping rewrites* are unconditional (MT.D.10), consequently mappings are as long applicable as *intermediate nodes* exist, follows, that *ReCRAG AST mappings* only terminate, iff all *intermediate nodes* are consumed[46].

Proof MT.P.1.2 *(Mapping steps of* ReCRAG AST mappings *result in ASTs satisfying the specified AST structure.)* Since a mapping step only replaces the tree a *mapping node* spans by another one (MT.D.9) and the rest of the AST does not change[47] (MT.L.1), it is sufficient to prove, that the tree a *mapping node* is rewritten to fits in it's parent node's context. This is guaranteed by definition, since the *mapping node* inherits from a *relationship node type* R and the mapping guarantees, that it is replaced by a type compatible *goal node* G (MT.D.9). Of course, subtypes of R fit in the contexts R fits.

After proving the correctness of *ReCRAG AST mappings*, it remains to show, that they always terminate. However, for termination a few additional constraints must be satisfied. Fortunately, this is no problem for any realistic language.

Definition MT.D.12 (expanded AST) *Let* Δ *be a* final AST*. Let* $FindTerminalAST(\Gamma)$ *be the function, which returns a* final AST terminal node *contained in the tree* Γ*. If* Γ *does not contain any* final AST terminal node*,* $FindTerminalAST(\Gamma)$ *returns a dummy* final AST terminal node*. Let* $Expand(T, \Lambda)$ *be the function, which returns the tree resulting, if the* final AST terminal node T*, contained in the tree* Λ*, is no terminal node anymore, but instead another child node spanning a tree. If* T *is not contained in* Λ*,* $Expand(T, \Lambda)$ *returns* Λ*. Let* $*$ *be the operator executing a term until a fix-point is reached. The* expanded AST $\Delta_{expanded}$ *of* Δ *is the AST resulting, if the following algorithm is executed:*

1. $\Delta_{expanded} := \Delta$

[46] If the *ReCRAG AST mapping* is implemented based on *JastAdd* it is *demand-driven*, such that mappings only occur if required.

[47] Hence, especially the context in which a *mapping* node is does not change, except the replacement of the *mapping node*.

2. $(\Delta_{expanded} := Expand(FindTerminalAST(\Delta_{expanded}), \Delta_{expanded}))*$

Remark: Let Δ be a *final AST*. Δ's *expanded AST* may not be a *final AST*.

Proposition MT.D.13 (Termination Constraints for *ReCRAG AST mappings*) A ReCRAG AST mapping κ *terminates, if:*

1. *the* expanded AST *of κ's input* final AST *is finite*
2. *the AST values of κ's* final AST computation attributes *do not contain any* invented intermediate ASTs
3. *the equations of κ's* final AST computation attributes *have a fix-point*

To understand the second constraint it is necessary to clarify, what an *invented intermediate AST* is.

Definition MT.D.14 (invented *intermediate AST*) *Let S_i be the set of* intermediate nodes. *Let $S_{fAtn}(\Delta)$, with Δ is an AST, be the set of trees spanned by Δ's* final AST terminal nodes. *And let $S_{sub-AST}(\Delta)$, with Δ is an AST, be the set of Δ's subtrees. The value Γ of a* final AST computation attribute *associated with a* mapping node *M, whereas M spans the tree $\Delta_{intermediate}$, contains an* invented intermediate AST $\Delta_{new} \in S_i \wedge \Delta_{new} \in S_{sub-AST}(\Gamma)$, *iff*

$$\Delta_{new} \notin \bigcup_{\forall \Omega \in S_{fAtn}(\Delta_{intermediate})} S_{sub-AST}(\Omega).$$

Clearly, the ASTs of any realistic language will satisfy the first two constraints. If the first constraints is not satisfied, the parser constructed an infinite AST. And if the second constraint is not satisfied, a *final AST computation attribute* simply invented a new program part not given in the original input, which has to be mapped again. If the third constraint is not satisfied, the specified mapping is not computable. However, usually non-termination, because of infinite computations of *final AST computation attributes*, is a hint for an implementation error[48].

Finally, the termination of *ReCRAG AST mappings* satisfying MT.D.13 can be proved.

Proof MT.P.2 *(Any* ReCRAG AST mapping κ *satisfying MT.D.13 terminates.)* Let Δ be κ's input *final AST*. Since κ satisfies MT.D.13, Δ's *expanded AST* $\Delta_{expanded}$ is finite. Let M_i be the set containing $\Delta_{expanded}$'s *intermediate nodes*. Since *mapping nodes* are *intermediate nodes* (MT.D.9) and during κ's execution no *invented intermediate ASTs* occur (MT.D.13), M_i contains all *mapping nodes* which can occur throughout the mapping of Δ. However, since $\Delta_{expanded}$ is finite, M_i is also. Hence, there exists a finite number of mappings. Additionally, the computation of each mapping terminates (MT.D.13). Consequently, the mapping of Δ terminates.

[48]The alternative would be, that the mapping is really not computable, which is pretty strange for computations based on trees. Anyway, i am highly interested about practical tree transformations, which cannot be computed based on a ReCRAG.

Last but not least, the confluence of *ReCRAG AST mappings* can be proved. To do so, another lemma is introduced first.

Lemma MT.L.5 *(The tree* Δ *a* mapping node M *spans does not contain any applicable mappings, except the mapping of* M*.)* Assume any of Δ's subtrees can be mapped. Since mappings can only occur at *mapping nodes* (MT.D.10), another *mapping node* M_2 must be contained in Δ's subtree. However, since *mapping nodes* are *intermediate nodes* (MT.D.9) and *intermediate nodes* are only contained in *intermediate ASTs* (MT.L.3) and *intermediate ASTs* only contain one *mapping node*, which is the trees root node (MT.D.4), a node M_2 cannot exist in Δ's subtree.

Proof MT.P.3 *(*ReCRAG AST mappings *are confluent*[49]*.)* Confluence follows from the facts, that mappings can only occur at *mapping nodes* (MT.D.10), the tree a *mapping node* M spans does not contain any applicable mappings, except the mapping of M (MT.L.5), mapping steps only manipulate the tree spanned by the mapped *mapping node* (MT.L.1), consume and replace it completely (MT.D.9) and only depend on it (MT.D.11; MT.D.9).

Generic Mapping Technique: Remarks for practical applications

The generic mapping technique presented is as less restrictive as possible, such that still correctness and confluence can be guaranteed in general and termination for practical applications. Only one restriction and one language construct not needed to guarantee these properties have been defined.

Firstly, the mapping technique could have been specified without *final AST terminal nodes*. However, in such a case its range of possible applications would drastically shrink, since one of its nicest properties is, that the parser can nest final and intermediate parsed program parts, if the specified *intermediate AST* supports appropriate *final AST terminal nodes*.

Secondly, MT.D.7 requires, that at most one *goal node* directly inherits from each *relationship node type*. This restriction is not required to guarantee any of the *ReCRAG AST mapping's* properties. However, if the components expecting mapped trees are considered, the restriction eases the access to a mapped node M a lot. If only one *goal node* F inherits directly from M's *relation ship node type*, a *synthesized attribute* A supporting the access of M as a node of type F can be introduced, such that semantic computations expecting a successful mapping, i.e. M is of type F, can access A to get M appropriately typed.

Another point to consider is, that in practice *goal nodes* will never occur as child nodes in *complete final AST* structure specifications. Instead always their *relationship node types* are used. If a *goal node* occurs as child of a node type T, the component constructing the AST to map must already construct for T's context the *goal node* and its subtree. However, the purpose of the mapping is to permit the construction of *intermediate nodes*, since the correct AST structure cannot be constructed in the first place.

[49] Hence, they are declarative.

Furthermore interesting is the classification of *ReCRAG AST mappings* with respect to *non-terminal attributes/higher-order attributes* [7, 48]. One could imagine to realize a mapping using *non-terminal attributes*, since *non-terminal attributes* compute ASTs based on a given AST and its attributation. However, *non-terminal attributes* extend ASTs in a monotonic way, i.e. the base AST a *non-terminal attribute* depends on still exists after its evaluation. Consequently, the computations expecting the mapped AST still would have to bother with the original AST's inconvenient structure and the need for complicated traversing computations, as presented in section 5.3.1, is only attenuated. This problem becomes even worse, if on considers the nesting of already correctly constructed AST parts and intermediate ones. *Non-terminal attributes* alone do not satisfy the main concern, why a mapping is introduced.

In fact, a *non-terminal attribute* computing a *mapped tree* is simply a *final AST computation attribute*. Or to say it in another way, the *ReCRAG AST mapping's final AST computation attributes* can be implemented as *non-terminal attributes*. Although, there is no need to do so, since *mapping nodes* are replaced either way.

ReCRAG AST mappings are closely related to *term rewriting* [39]. A *ReCRAG AST mapping* is a special *term rewriter* implemented based on ReCRAGs, which rewrites ASTs instead of arbitrary terms. Its ReCRAG based implementation also is the *ReCRAG AST mapping's* great advantage compared to ordinary term rewriter, since the fact, that the transformed AST will be further analysed by an AG depending on its new structure is appropriately incorporated. The nesting of correctly constructed AST parts in *final AST terminal nodes* in combination with the need, that *final AST computation attributes* only depend on the *intermediate AST* spanned by their *mapping node*, permits the specification of the *complete final AST's* further semantical analysis without the need to consider the *intermediate AST's* structure. E.g. an AG A specifying further analysis does not need to define the equations of *inherited attributes* in the context of *final AST terminal nodes*. Assume a *final AST terminal node* T has an *inherited attribute* i. If T would be an ordinary child of its intermediate parent node P (and no terminal node), i's value would have to be specified for P's context. But since T is a terminal node, the attributes associated with the tree it represents simply do not have to be defined. It is clear by the mapping's definition, that the *intermediate AST* will be transformed in a *complete final AST*, such that it is sufficient to specify A's semantics only with respect to *complete final ASTs*. Hence, it is valid to say, that *ReCRAG AST mappings* map ASTs in an automatically and transparent[50] way. Of course, on the other hand *ReCRAG AST mappings* are not as general as ordinary *term rewriter*. Their "only" focus is the transformation of ASTs.

The last practical remark is about the possibility to introduce arbitrary long chains of *ReCRAG AST mappings* to transform trees along tree structures $S_1, S_2, \ldots, S_n$. Of course, on a practical view point one has to ask if such a chain is necessary or if the direct transformation S_1 to S_n is more efficient.

[50] Except for the need of the above mentioned access attributes to retrieve typed *goal nodes* or alternatively upcasts from *relation ship node types* to the actual types of the *goal nodes*.

5.4 Unresolved Problems

The *JastAdd* ReCRAG generator system used to implement *JastAddC's* semantics satisfies all the objectives and requirements mentioned in chapter 2.1. Especially the modularization across concerns can be realized (see chapter 3.2). Consequently, the following problems are on a technical level or are improvement suggestions.

First of all, the *JastAdd* version used to generate the *JastAddC* compiler has a minor bug. If a *collection attribute* contribution is associated with `ASTNode`, the *JastAdd* system weaves a `super.contributeTo_NodeTypeAndCollection-AttributeName(collection)` call in `ASTNode`'s automatically generated `contributeTo_NodeTypeAndCollectionAttributeName` method. Of course, `ASTNode`'s super class does not have such a method. The `super` call simply has to be deleted.

Another problem regarding *collection attributes* already has been mentioned in section 5.2.1. The AST Δ a *non-terminal attribute* N represents does not contribute to *collection attributes*. Hence, if Δ contains nodes which should contribute to a *collection attribute*, these contributions will not happen, since N is not automatically evaluated during the collection of all the nodes with contributions. As workaround the already presented solution, simply to use an ordinary synthesized attribute computing the tree and an unconditional rewrite, can be applied.

A nice feature, currently missed in *JastAdd*, would be the possibility to declare rewrites, similar as attributes can be declared without being defined immediately. Hence, *JastAddC's* current *compiler-core AST mapping* concern does not specify, which parts of the AST must be rewritten to which ones. Only the *AST mapping's* implementation declares and defines mapping rewrites. However, the *AST mapping* module should only implement rewrites, whereas the *compiler-core* should specify the need for a rewrite.

Another nice feature would be, if AST node types could also be interfaces, such that node types can inherit from several other node types. The *JastAdd* system could recognize, whether several node types inherit from a node type T or not. If several node types do, T could become an interface I, whose associated semantics are automatically woven in all the types inheriting from it. T itself could be implemented as another node type inheriting from I.

Chapter 6

JastAddC's Development Status

This chapter gives a short overview about *JastAddC's* development status. First, the *C99* parts *JastAddC* realizes are listed and afterwards the parts still missing.

6.1 Implemented *C* Parts

The *JastAddC C99* compiler front-end is not complete. Several language constructs specified by the standard [23] are not supported or their semantics are not implemented. The selection criteria for the implemented parts have been:

1. To show a general architectural concept for a complete, independent compiler (chapter 3), integrating the lexical, syntactical (chapter 4) and semantical analysis (chapter 5).

2. To present a compiler extension concept (chapter 3).

3. To develop techniques to construct a nice AST and to show general concepts, which realize a logical, well-founded transformation of input information to refined representations, so information are much more easy to access. E.g. the mapping from the concrete syntax to the abstract one (chapter 5.3) and the AST access support module (chapter 5.2.2).

4. To support a stable, consistent and comprehensible AST, representing all *C99* constructs.

5. To show in general, how language semantics can be implemented by augmenting the AST with attributes (chapter 5.2).

6. The semantic parts implemented should cover the most complex semantic language constructs as well as basic concepts, other semantic constraints depend on, e.g. the name analysis.

7. To show a general implementation concept for the realization and integration of the most common language independent compiler tasks. E.g. the error handling (chapter 5.2.1)[1].

8. To show a general semantic constraint checking concept separated from, but reusing, actual context computations.

Altogether, the implemented *C99* front-end is a basic skeleton for arbitrary *C* compilers. Missing language parts are easy to integrate and the concepts needed to implement them are demonstrated throughout the thesis (especial chapter 5). The compiler architecture is clear and no further overlooked problems should occur. The only exception is the *typedef problem*, for which a solution is suggested in chapter 7.2.

The best way to get an overview about all the semantic constraints *JastAddC* checks is to investigate the constraint checking module's *IsCorrectLocal()* attribute equations, which specify for each AST node type its associated semantic constraints (see chapter 5.2.3).

6.2 Missing *C* Parts

Table 6.1 summarizes still outstanding tasks, which have to be implemented so *JastAddC* becomes a full *C99* compiler-frontend. For each part the semantic constraints as well as support functions have to be implemented. The *preprocessor* should be implemented as its own compiler, which is executed during the lexical analysis, so it becomes another lexer in the lexer stack[2]. The point "Implicit information retrieval" in table 6.1 already suggests its implementation: The semantics the standard implicitly defines for certain language constructs have to be made explicit by rewriting the AST.

The most important missing parts are the type analysis, the external translation unit semantics[3] and the preprocessor. But also semantic checks for other language constructs are left, like all semantics associated with the keywords `volatile` and `restrict`. The lexical analysis is nearly complete, except identifiers do not support universal characters and it is not checked if universal characters in string literals or character constants are within the permitted value range. The syntactical analysis is complete. However, old function style definitions are not supported, since the standard already states, that they are an obsolescent feature. The specified AST does not support array parameter declarations augmented with the `static` keyword between their square-brackets, e.g. `void func(int array[static])`. Last but not least, *JastAddC* is only a compiler front-end, thus no code generation module exists. To implement such a module, the techniques presented in chapter 5 can be reused (see chapter 7.3).

Except the *typedef problem*, the implementation for the missing semantics should be straight forward. But it can become much more complicated than

[1] The error handling is concerned about the internal representation of errors and interacts with the recognition of errors, their presentation for the user and the constraint checking (chapter 5.2.3)

[2] For details about the lexer stack see chapter 4.1

[3] The external translation unit semantics specifies how objects and functions defined in one source code file can be reused in other ones and how functions are matched accross source code units.

Table 6.1: *C* parts not implemented in *JastAddC*.

Universal characters	
1	In identifier names [6.4.2.1:3;p. 51].
2	Checks if universal characters are in the permitted value range [6.4.3:2;p. 53].
Preprocessor [6.10;p. 145]	
External linkage semantics [6.2.2;p. 30]	
1	`extern` keyword semantics.
2	Function call and definition mapping across translation units.
3	Object mapping across translation units.
Type analysis	
1	Analysis if an object is a *lvalue* or *rvalue* [6.3.2.1:1;p. 46].
2	Analysis if *lvalues* are modifiable [6.3.2.1:1;p. 46].
3	Compatibility checks between simple types, arrays, pointers, taged types and functions [6.2.7:1;p. 40].
4	Construction of composite types for compatible types [6.2.7:3;p. 40].
5	Type promotions. E.g. in expressions or promoted function call arguments [6.5.2.2:4,6,7;p. 71-72].
6	Analysis to check if an expression is constant [6.6;p. 95] and, if so, its evaluation.
Keyword semantics	
1	`extern`, `inline`, `register`, `restrict`, `sizeof`, `volatile`
Implicit information retrieval - Automatically triggered AST rewrites	
1	Array objects are treated like pointers to the type of the array elements [6.3.2.1:3;p. 46].
2	Function names are treated like pointers, which type is the one of the function definition [6.3.2.1:4;p. 46].
3	Initializer semantics: The computation of default values, implicitly given initializer designations and implicitly specified array sizes [6.7.9;p. 125].
The standard library	

expected, because *C* is very close to the hardware. Additionally, many *C* parts are not completely defined. Instead it's up to the compiler developer to define undefined parts. Unfortunately, such low level parts have consequences up to high level semantics. It also has to be considered, that *JastAddC* is running on the Java Virtual Machine (JVM) [35], but the program *JastAddC* analysis may be written for any machine. To illustrate the problem, consider the evaluation of constant expressions. Often a semantic constraint for some expression has to be constant and to be in some value range (e.g. to be greater or equal to 0). Thus, it is necessary for the compiler to evaluate constant expressions. But the value of a constant expression is machine dependent, i.e. it is compiler dependent. E.g. `1 « 17` is a constant expression, but because of integer overflow the value for a machine with 16bit integers depends on the machine's behavior. Additionally, it is important if one- or two-complement is used to represent negative values. To support the evaluation of constant expressions in *JastAddC*, not only attributes evaluating such expressions would be necessary, but also a module specifying the machine arithmetics. If *C* is implemented in itself and no cross compiler is developed, as it is most times the case, no such problems occur. On the other hand, if a way to specify the destination machine exists, *JastAddC*'s extensibility will make it very easy to adapt it for many different backend machines.

Chapter 7

Further Development and Solution Concepts

In this chapter earlier mentioned problems will be investigated and solutions not yet implemented in *JastAddC* or the tools used will be suggested. These solutions are with no means complete and must be further investigated. Maybe some even turn out to be to hard to realize. To understand the techniques presented the reader should be familiar with GLR and SGLR parsing. A good introduction to GLR parsing is [37]. Details about SGLR are presented in [47].

The two main problems, which cannot be handled without new concepts, are the modularization of lexical and syntactical specifications (chapter 3.3) and the *typedef problem* (chapter 4.3).

After presenting solution concepts for these two problems, it will be shown how the techniques and concepts already used in *JastAddC* can be reused to solve missing language parts. To do so, a code generation concept will be presented.

Last but not least, a further development schedule will be suggested.

7.1 Supporting full Modularization across Concerns: SGLR

This section shortly recapitulates the modularization problems for the lexical and syntactical analysis presented in section 3.3 and suggests a solution concept: SGLR parsing.

7.1.1 Overview: Lexical and Syntactical Modularization Problems

In chapter 3.3 the modularization problems of *JastAddC's* lexical and syntactical specifications have been presented. These problems do not only occur just because *Beaver* and *JFlex* are used. Instead the theoretical concepts *Beaver* and *JFlex* depend on cannot support modularization in general.

As mentioned in chapter 3.3 LR(k) grammars are not closed under unification and regular-expression based lexers may require special token precedence rules. The problem, with respect to extensibility, is not only, that it is impossible to know a compiler's concrete configuration in advantage, i.e. to forsee all applied extensions and the conflicts they introduce, but also, that it is necessary to change existing modules' implementation to solve their conflicts.

Consider an extension A introducing a new keyword `foreach`. Since extension modules should be independent of each other, how should the lexer generator know if the `foreach` keyword is preferred to the identifier `foreach` specified in another module B? The developer of A may even not know B exists. Using explicitly annotated priority rules will break the *Black-Box-Principle* between modules.

An even more complex example is the embedding of one language in another one. E.g. [50] embedded Structured Query Language (SQL) in *Java*. One could do the same for the *JastAddC* compiler. Now the token `WHERE` can be the identifier `WHERE` or the SQL keyword `WHERE` depending on the context it occurred. Listing 7.1 presents a short *C* program using the SQL and `foreach` keyword extensions.

Listing 7.1: *C* example for SQL expressions and the `foreach` keyword.

```
1  SQL_RESULT* get_NUMBERs_by_WHERE(char* WHERE, char* dbs[99])
2  {
3      SQL_RESULT res[99];
4      int i=0;
5      foreach (DB db:dbs;99)
6      {
7          res[i++] = SELECT NUMBER FROM db WHERE WHERE = WHERE;
8      }
9      return res;
10 }
```

Although, listing 7.1 exemplifies not a lexical (regular) problem, but instead a syntactical (context-free) one, solutions for it require to change the lexer as well, if the traditional lexer parser separation is given. [50] suggests a technique, which uses the parser's look ahead information to reduce the set of tokens the lexer can recognize[1]. Thus, a modified lexer can solve such problems. However, as has been brought up in chapter 4.3 and will be discussed in more detail in section 7.2, cyclic dependencies between lexer, parser and semantic evaluator should be avoided. The SGLR approach presented in the next section solves both modularization problems, the lexical as well as the syntactical. Additionally, it is theoretically backed and the basis for a *typedef problem* solution (chapter 4.3), which will be presented in section 7.2.

7.1.2 Solution Concept: SGLR Parsing

SGLR parsing is a GLR based technique, where the lexical analysis is embedded within the syntactical one, such that there is no need for an external lexer. A

[1] A feedback loop from the parser to the lexer is introduced. Each time the next token has to be recognized the parser tells the lexer which tokens it expects. In return the lexer can discard all the tokens not contained in the set of possible tokens. [50] showed, that most lexical problems introduced by embedding languages can be handled using this technique.

SGLR parser's input is the raw program text, instead of a stream of tokens. Consequently, its specification must define how to match input characters, i.e. perform the lexical analysis. Since regular languages are a subset of the context-free languages, for each regular expression an equivalent CFG can be specified. Thus, each token can be represented by a *non-terminal*, whose productions describe the token. In practice extended Backus-Naur Form (EBNF) is used to specify context-free languages and any regular expression can be expressed using a single EBNF rule. Hence, the recognition of tokens can directly be embedded in EBNF based CFG rules. Only often used, complex tokens should be expressed by their own *non-terminals* and rules. Listing 7.2 presents a short SGLR example specification for the `enum` specifier, as it is defined in the standard's grammar (see appendix A rule 6.7.2.2).

Listing 7.2: SGLR specification for the `enum` specifier. Program input characters are embedded between `"`.

```
enum_specifier =
    "enum" identifier? "{" enumerator_list ","? "}" |
    "enum" identifier;

identifier =
    identifier_nondigit |
    identifier identifier_nondigit |
    identifier digit;

identifier_nondigit =
    nondigit |
    universal_character-name;

nondigit =
    "_" |
    "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
    "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" |
    "A" | "B" | "C" | "D" | "E" | "F" | "G" | "h" | "I" | "J" | "K" | "L" | "M" |
    "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";

digit =
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

universal_character_name =
    "\u" hex_quad |
    "\U" hex_quad hex_quad;

hex_quad =
    hexadecimal_digit hexadecimal_digit hexadecimal_digit hexadecimal_digit;

hexadecimal_digit =
    digit |
    "a" | "b" | "c" | "d" | "e" | "f" |
    "A" | "B" | "C" | "D" | "E" | "F";
```

Listing 7.2 might look complicated, but it shows, that the standard's grammar specification can be treated as a SGLR specification. Just compare the specification given here with the one for identifiers in the standard (see ap-

pendix A rule 6.4.2.1). Of course, the specification can be much shorter, as demonstrated in listing 7.3, if character value ranges and counted repetitions are used[2].

Listing 7.3: Simplified SGLR specification for *identifier*.

```
1  identifier =
2      identifier_nondigit (digit | identifier_nondigit)*;
3
4  identifier_nondigit =
5      "_" | [a-z] | [A-Z] | universal_character-name;
6
7  digit =
8      [0-9];
9
10 universal_character_name =
11     "\u" hexadecimal_digit{4} |
12     "\U" hexadecimal_digit{8};
13
14 hexadecimal_digit =
15     digit | [a-f] | [A-F];
```

GLR parsing solves our modularization problem for syntactical specifications (see 3.3) since GLR parsers can be constructed for any context-free language and context-free languages are closed under unification. So the final grammar specification can be split across several sources. Each source can represent one compiler module. Hence, a modularization concept analog to *JastAddC*'s modularization concept for the semantical analysis (see 3.2.2) can be implemented. Other modules can reuse grammar parts by using the corresponding *non-terminals*. Language extensions can be done by introducing additional production alternatives. Supporting the modularization for lexical specifications in SGLR is simply achieved by the support for the modularization of syntactical specifications in GLR. Thus, based on SGLR parsing and ReCRAGs, each extension/module can specify all it's lexical, syntactical and semantical components. The SGLR parser generator can weave the different parser specifications together in a similar way as *JastAdd* does for ReCRAGs. Thus, an AOP technique similar to the one described in chapter 2.2 for ReCRAGs can be developed for SGLR. The basic contributions woven into the CFG are *non-terminals*, *terminals* and *rules*.

Listing 7.4 and 7.5 are SGLR grammar specifications for the `foreach` and SQL extensions. The existence of a basic grammar, which is part of the compiler core[3] and specifies the *non-terminal* `expression`, is assumed.

Listing 7.4: SGLR specification for the `foreach` extension module.

```
1  expression =
```

[2]Character value ranges are specified using the notation `[charter1-character2]`, where *character1* is the lower and *character2* the upper bound. Bounded repetitions are specified using the notation `{iterations}`, where *iterations* specifies how often the construct on the left consecutively occurs.

[3]Similar to the *compiler-core* concept for semantical specifications, described in chapter 3.2.2, a *compiler-core* for syntactical specifications can be defined.

```
2     "foreach" "(" Expression identifier ":" Expression ";" Expression ")"
          Statement;
```

Listing 7.5: SGLR specification for the SQL extension module.

```
1  /*
2   * We assume simple SELECT FROM WHERE SQL expressions.
3   * It would be no problem to support much more
4   * complicated ones. E.g.:
5   * SELECT a.att1, COUNT(b.att2)
6   * FROM db1 b LEFT OUTER JOIN db2 a
7   *        ON b.att2 = a.att3
8   * WHERE a.att1 = value
9   */
10 expression =
11     "SELECT" identifier "FROM" identifier "WHERE" identifier "=" identifier;
```

With respect to chapter 3.2, the `foreach` and SQL extension modules must also contain the constructs' semantical specifications beside the syntactical ones presented in listing 7.4 and 7.5. Figure 7.1 illustrates the resulting compiler architecture/modularization.

Of course, GLR parsing also has drawbacks. If several modules are integrated and the resulting grammar is ambiguous, the parser can find more than one derivation for a given input, i.e. its result can contain more than one tree. In such a case it is necessary to disambiguate the language. The disambiguation can be done by the semantical evaluator. An attribute grammar can be used to evaluate the different trees and discard all of them except one. Section 7.2 describes this technique in more detail, introducing a schema for *JastAdd*. For some ambiguities the disambiguation can also be implemented in the SGLR parser, thus the parser's specifications can be extended with disambiguation constructs [46].

If we have to worry about disambiguation techniques, because of module composition, it raises the question about the SGLR parsing's applicability for the modularization of lexical and syntactical specifications. To tell the bad news first: If ambiguities occur the compiler developer composing the modules has to find a manual solution. The good news is he can use elaborated techniques, which are very powerful: Attribute grammars[4]. Additionally, the solution will be clean. Only theoretically backed concepts are used: SGLR parsing and ReCRAGs. Also the compiler's disambiguation can be implemented as its own compiler module. No changes in any of the composed modules are necessary. All tailoring can be specified in the disambiguation module.

[4]For details see section 7.2

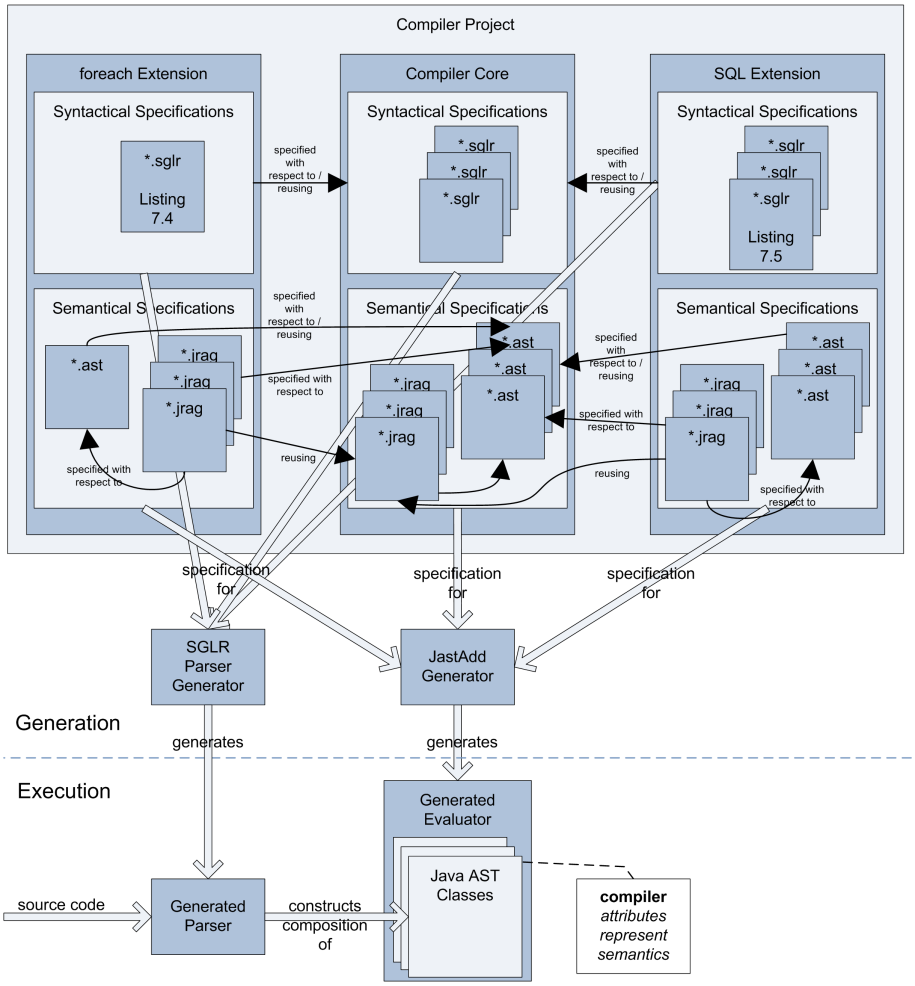

Figure 7.1: Complete compiler modularization across concerns with SGLR and ReCRAGs. The *JastAddC* project structure of figure 2.2 and the SGLR parser generator project structure together form the complete compiler project structure. The *Ant* [15] *build scripts* presented in figure 2.5 as well as the modules implementing the *core-concerns* are left for convenient reasons.

7.2 Introducing Disambiguation Techniques: JastAdd, GLR-Parsing and the *typedef problem*

In chapter 4.3 the *typedef problem* has been presented. The *typedef problem* is an instance of a generic problem class: Dependency cycles between lexical, syntactical and semantical analysis. E.g. a lexer may need syntactical context information or a parser may depend on semantical information. Such kind of problems are worse, since the logical program structure is broken. The theoretically nice concepts, where information is refined by each compilation phase, are lost and ungenerateable "hacks" must be applied to solve the problem. It would be much more convenient if such dependency cycles could be broken, such that there is no need to "hack" manual written code snippets[5] into generated components.

A closer look clarifies, that the information exchanged in a cyclic dependency supports a logical earlier compilation task T_A with decision information from the logical next compilation task T_B, such that a deterministic decision can be done by T_A. If the decision information is not supported, T_A simply becomes non-deterministic. Thus, T_A would not deliver one task output consumed by T_B, but instead a set of task outputs. If T_B can disambiguate the task outputs of T_A, which means T_B is able to filter the set of task outputs from T_A in such a way that at most one output element is kept, the dependency cycle can be broken and the tasks can be executed in sequential order. It is clear by definition: Iff a dependency cycle between a logical first task T_A and its successor task T_B exists and it can be computed, a disambiguation algorithm to filter the outputs of T_A, if it is made non-deterministic, must exist for T_B. The proof is rather simple:

In the cyclic dependency solution T_B provides the needed decision information for T_A, thus T_B must compute it. If the dependency cycle is broken, T_B simply uses this computation to disambiguate the results of T_A. This proof and consequently the breaking technique for cyclic dependencies only holds, iff T_B can compute its decision information without requiring the next result from T_A. Hence, the proof only holds if no *fix-point semantics* are needed to compute a **single** result of T_A.

Figure 7.2 illustrates the breaking of cyclic dependencies between *compilation phases.*

It is a good idea to break dependency cycles between generated compiler components in the way described above, because the theoretical concept behind T_B is expressive enough to specify the disambiguation of T_A's results[6]. Thus, a solution can be generated using usual specifications for the generators without "hacking" some imperative code snippets into the compiler. Performance issues introduced by the non-deterministic computations, especial if dependency cycles

[5]Code snippet is a much to friendly paraphrase. These "snippets" tend to become the main anchors, where the rest of the compiler is build around. Thus, one often has the feeling the "snippets" do not supplement the compiler, instead the compiler's architecture are the "snippets" plus X.

[6]The solution described above is transitive. If a logical first task T_A depends on a task T_B, which itself depends on a task T_C, the complete disambiguation can be done by T_C.

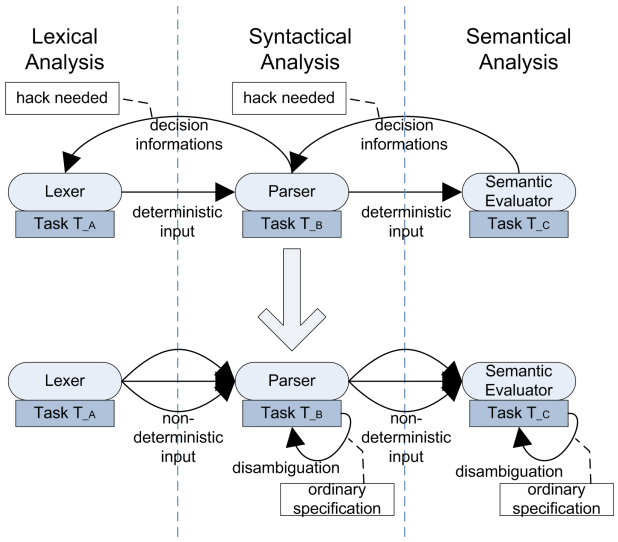

Figure 7.2: Breaking cyclic dependencies between *compilation phases*.

have to be broken between several compilation phases, may be a drawback.

An advantage of the SGLR technique is, that no explicit lexical compilation phase exits. Together with the powerful ReCRAGs concept, it is enough to distinguish two compilation phases: The syntactical and semantical. So, dependency cycles only have to be broken between these two. Also SGLR parsers automatically realize the non-deterministic result computations, i.e. their result is a parse forest. What remains to show is how the disambiguation of the parse forests computed by SGLR parsers can be done using *JastAdd*.

7.2.1 Disambiguation Concept for *JastAdd*

The concept presented in the following is inspired by [5] and [30]. But instead of term rewriting, ASTs are rewritten and instead of specific filters, the ASTs are filtered using filter attributes specified in *JastAdd*. Thus, the concept is tailored to provide a disambiguation concept for *JastAdd*.

The first problem encountered, if the parser generates sets of ASTs is, that *JastAdd* only expects one AST to evaluate. Fortunately it is possible for the GLR parser to construct always exactly one AST, which contains all the non-determinism encountered during parsing. The concept of *ambiguous nodes* can be used to transform a set of syntax trees, whereas each tree represents the same input, to one tree representing the complete set.

For each non-determinism encountered by the parser an *ambiguous node* will be constructed in some further parsing step. The *ambiguous node's* child nodes are different tree alternatives $\Delta_1, \ldots, \Delta_n$. Each alternative $\Delta_1, \ldots, \Delta_n$ represents the **smallest** input part w of the complete input ywz, such that the grammar rules involved in the parser conflict derive w or a part of it and

all alternatives $\Delta_1, \ldots, \Delta_n$ have the same root node type[7]. This *optimization condition* is of key importance. It ensures that the deepest appropriate node will become the *ambiguous node* representing alternative derivations resulting from a *non-determinism* encountered during parsing. Of course, an arbitrary number of *non-determinisms* can be encountered, so an arbitrary number of *ambiguous nodes* may be constructed. It is important to understand, that an *ambiguous node* does not represent the splitting of the GLR parser stack. *Ambiguous nodes* represent points in a syntax tree, where several different subtrees may occur, such that each possible subtree represents the same word w. Thus, a *non-determinism* encountered during parsing step n will result during a further parsing step $n+x$ to the construction of an *ambiguous node*. Of course, if one of the parser's alternatives later turns out to be no solution[8] the tree constructed so far for it will be discarded. If only one alternative is left no *ambiguous node* has to be constructed at all.

The *JastAdd* evaluator's input finally is the AST constructed by the GLR parser, i.e. a tree containing all ambiguities represented by *ambiguous nodes*. Figure 7.3 shows an arbitrary AST with three *ambiguous nodes*. The *ambigious nodes*, as well as their possible subtree children, are highlighted with purple borders.

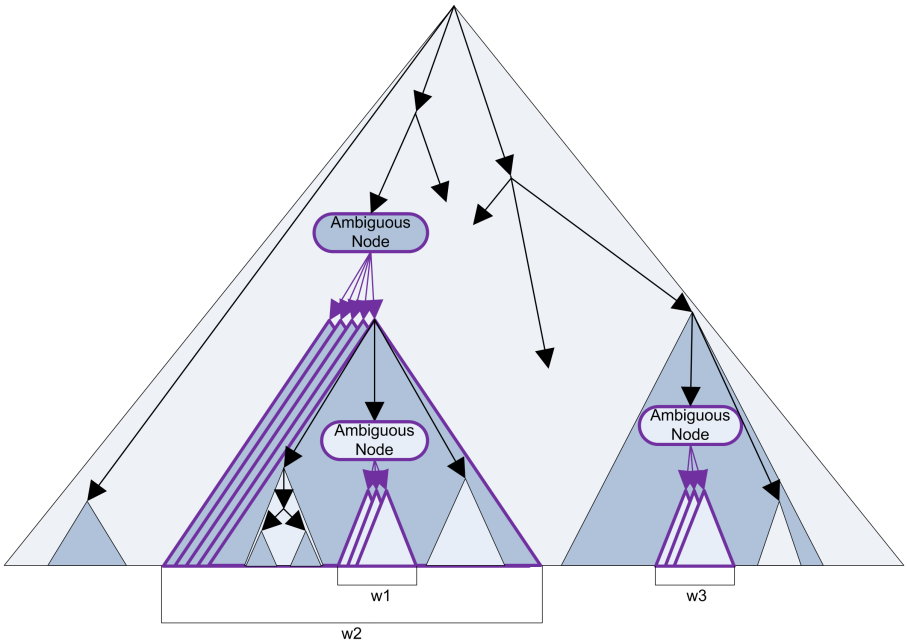

Figure 7.3: Imaginary AST with *ambiguous nodes*.

The construction of appropriate ASTs representing parse forests is quiete complicated, because of the *optimization condition* mentioned before and the

[7]Since the syntax tree is constructed in a *bottom-up* manner, the alternative trees $\Delta_1, \ldots, \Delta_n$ are known in some later parsing step, which results if all the non-determinism's possible successor states are further investigated.

[8]The GLR parser cannot derive the input for the alternative.

delayed construction of *ambiguous nodes*. However, it is the SGLR parser's task to know the points in a parse tree, where several already constructed alternative trees can be consumed by a reduction. To say it in another way, of course the SGLR parser must know all possible input derivations. Nevertheless, for development purposes it is necessary to know all the CFG's *non-terminals*[9] from which several derivations for equal words **may** exist[10], so all required *ambiguous node* types can be specified. To compute this information may be very complicated and has to be further investigated.

The disambiguation is straight forward and very similar to the concept presented in chapter 5.3. For each *ambiguous node* an attribute `Disambiguate()`, which selects one of the node's children (thus trees), is specified. Additionally an unconditional rewrite for each *ambiguous node* to the tree the `Disambiguate` attribute returns is given. The `Disambiguate` attribute represents a filter and its equation can be any valid *JastAdd* equation[11]. To be able to replace the *ambiguous node* by any of the alternatives contained, the *ambiguous node* must be of the same node type the contained alternatives are. Since an *ambiguous node's* children represent alternative trees with the same *non-terminal* as root node, it is obvious, that all contained alternatives share a common smallest ancestor node type. This type is sufficient for the *ambiguous node*, i.e. it can be used as child node in further reductions.

To support the disambiguation technique just described one problem to solve remains. A `Disambiguate` attribute's equation may reuse any other attributes, especially *inherited attributes* of the *ambiguous node's* children. Unfortunate, since the *ambiguous node* is in-between, such *inherited attributes* require an equation in the context of *ambiguous nodes*. But it is obvious, if the child node replaces the *ambiguous node* all its *inherited attributes* are defined. Thus, it would be nice if all *ambiguous nodes* A automatically forward the correct *inherited attribute* value if an *inherited attribute* of C is called and C is child of A. Of course, such an behavior can be specified manually using ordinary `*.jrag` specifications. However, an automatically approach would be much more convenient.

Classifying the disambiguation technique for *JastAdd* using [30]'s schema it follows, that the `Disambiguate` attribute is a filter, which is

[9] The *non-terminals* and **not** the parser's shift-reduce and reduce-reduce conflicts.

[10] Of course, the problem if a CFG is ambiguous or not is undecidable **in general**. However, it is not by chance, that we like to know the *non-terminals* for which an ambiguity **may** exist. In fact, a GLR parser is based on a concrete LR(k) parsing table. Hence, it is known during the parser's construction, if the parser is non-deterministic or not. If shift-reduce or reduce-reduce conflicts are given, the parser can encounter situations where several possible solutions how to keep on parsing exist. This solutions may or may not result in several possible parse trees. Important is, if the GLR parser has a conflict, it "thinks", that it might find several parse trees for the same input. However, the fact that the GLR parser "thinks" so, does not mean that the grammar is ambiguous. Assume the grammar is LR(2) but a GLR(1) parser is used. If the GLR parser encounters any non-determinism it keeps on parsing all possible alternatives. However, all except one alternative will be discarded in further parsing steps. How often such false alarm (false-positives) will be reported in practice has to be further investigated. Of course, the solution suggested in this section is only useful, if it is in practice possible for a developer to specify a disambiguation. Hence, the developer should not have to bother about to many false alarms. For now it is important, that if the grammar is ambiguous a GLR parser will report a conflict.

[11] So the attribute can represent any computable filter.

1. Independent from the CFG specifying the parser
2. Independent from the parsing method
3. Executed after parsing (post-parse filter)

The disambiguation by attributes depends only on the AST the parser constructed. Of course, if an incomplete AST's `Disambiguate` attribute can be computed without any problems, the parser could trigger the disambiguation during the parsing process. To do so semantic actions could be specified for the parser, which start the AST's disambiguation. This optimization is always possible, if the `Disambiguate` attribute's value only depends on the subtree spanned by the AST node it is associated with.

Also, other techniques to disambiguate GLR parse forests during parsing exist. Especially if an SGLR parser is used many ambiguities may occur. [46] presents special SGLR disambiguation techniques by introducing the following concepts: *follow restrictions* (applicable during parse table generation for look ahead 1, for look aheads > 1 applicable during parsing), *reject productions* (applicable during parsing), *priority and associativity* (applicable during parse table generation) and *preference attributes* (applicable after parsing). Each of this disambiguation techniques[12] can be implemented as a post-parse filter. The advantage of executing a disambiguation strategy as early as possible, preferable during the lexer/parser construction, is a much better compiler performance[13]. However, the disambiguation techniques presented in [46] are on the context-free level, whereas the disambiguation technique presented above is on the context-sensitive level. Thus, the disambiguation technique presented in this chapter is more powerful than the ones presented in [46].

Recapitulating the *typedef problem* presented in chapter 4.3, a possible solution concept now becomes clear. The *typedef problem's* ambiguity is introduced if rule 6.7.7 of the standard's grammar (see chapter A) is implemented as specified, i.e. instead of introducing an explicit `typedef_name` token, simply the `identifier` token is used. The consequence are shift-reduce and reduce-reduce conflicts in the standard's LALR(1) grammar. Especially it becomes ambigious (see chapter 4.3.2). Hence, the SGLR parser used will produce parse forests. To break the cyclic dependencies in the way just described, the disambiguation must be handled by the compilation phase, which follows the phase producing the non-deterministic output. Thus, the disambiguation has to be handled by the *JastAdd* evaluator using the *ambiguous node* concept. *Ambiguous nodes* and the `Disambiguate` attributes together with the *ambiguous nodes'* unconditional rewrites must be introduced.

The original *typedef problem* crossed all three compilation phases. The suggested solution shifts the *typedef problem* into the semantical analysis, as it should be the case. Finally, the *typedef problem* becomes understandable and manageable using a well founded concept.

[12] filter

[13] To understand why, it is sufficient to reconsider the concept, how cyclic dependencies between *compilation phases* can be broken. As later the disambiguation is done, as more non-deterministic *compilation phases* are given. Hence, as more non-deterministic results must be constructed. Also, each *compilation phase* based on non-deterministic results has to process all the possible results and becomes non-deterministic itself, if it does no disambiguation.

Unfortunately, another problem seems to arise: Worse interactions between the mapping technique introduced in chapter 5.3 and the just presented disambiguation technique for *JastAdd*. The next section will show, that both can be used simultanously without any problems.

7.2.2 Integrating Disambiguation and Mapping Techniques

The mapping technique presented in chapter 5.3, as well as the disambiguation technique just shown, are based on attributes and rewrites. With respect to compiler extensibility, it is important to guarantee, that both techniques can be applied at the same time without the need to adapt them. Thus, independent extension modules can use these techniques, which may lead to concurrent applications of mappings and disambiguations, but still each extension is only aware of its applied techniques. It is important, that no compiler part has to worry his mappings or disambiguations might interact in a worse way with a mapping or disambiguation implemented in another part. The mapping and disambiguation technique must automatically integrate in a correct way. Fortunately, such a guarantee can be given, if the attributes and rewrites are implemented without side-effects[14].

Let's assume the parser encounters a non-deterministic situation. Additionally, assume an arbitrary number of the alternatives are represented by *intermediate nodes*, thus they are subtrees which have to be rewritten to *complete final ASTs* using mappings. The SGLR parser just constructs an *ambiguous node* with the alternatives as children. It is important to acknowledge, that as soon as the *ambiguous node* is accessed it's disambiguation rewrite is triggered. Hence, any access of alternatives can only be caused by the *ambiguous node's* disambiguation rewrite, i.e. by its `Disambiguate` attribute. Thus, the AST behaves like no *ambiguous nodes* exist at all. The AST user only encounters the disambiguated tree. Consider any *intermediate node* being child of an *ambiguous node* is accessed during the *ambiguous node's* disambiguation. The *intermediate node* will trigger its own unconditional rewrite, which realizes it's mapping to a *complete final AST*. Thus, for the *ambiguous node* all its children look like *complete final ASTs*. To summarize the behavior:

1. The AST is automatically disambiguated if needed, because as soon as an *ambiguous node* is accessed its disambiguation rewrite is triggered.

2. To do the disambiguation the *ambiguous node's* alternative trees may be accessed[15].

3. As soon as an alternative tree is accessed, which is an *intermediate tree*, the tree's unconditional mapping rewrite is triggered.

Altogether, *JastAdd's* demand driven approach guarantees correct behavior for any mapping and disambiguation combination. Of course, the demand driven approach must not be broken, i.e. the mappings and disambiguations must be

[14] *JastAdd* specifications should always be implemented side-effect free, since only an implementation without side-effects guarantees the advantages of ReCRAGs: Declarative, *demand-driven*, extensible specifications.

[15] At least one, which has been chosen as disambiguation result.

side-effect free. Figure 7.4 illustrates the interaction between mapping and disambiguation.

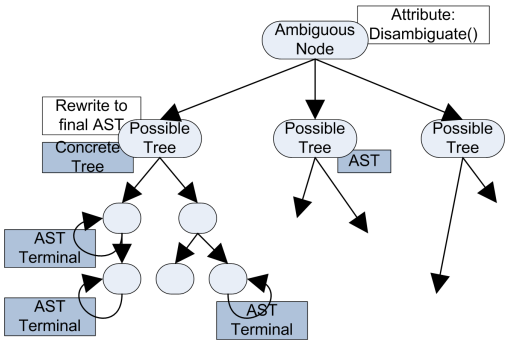

Figure 7.4: Automatically correct interaction between *ReCRAG AST mappings* and the disambiguation technique.

The interaction between mapping and disambiguation also is very interesting from a compiler designers viewpoint. Since GLR parsing can be used to parse any context-free language, it is possible to construct a parser based on the abstract syntax. The mapping module used to transform the first constructed syntax tree to the *complete final AST* seems to be dispensable. But in fact it is not, because most abstract syntaxes are ambiguous. There is a tradeoff between the effort put into the mapping and the disambiguation. If the parser depends on a more restrictive concrete syntax less effort for the disambiguation is needed, but the constructed tree may not be ideal. If the language's semantics are implemented based on such a concrete tree, the implementation will become very complicated. So a lot of effort has to be put into the mapping. On the other hand, if the parser's syntax is very similar to the abstract one, the mapping's implementation becomes much more easy. Unfortunately, at the same time additional disambiguation is needed.

One can even try to view the mapping technique of chapter 5.3 as a special disambiguation case. But such a viewpoint is misleading, because the mapping requires no *ambiguous nodes*, thus the parser never encounters any non-determinism. It becomes clear, that the mapping module's task is to refine the tree, but not to add any semantic information, whereas the disambiguation module's task is to compute additional semantics and add them to the tree by throwing away wrong alternatives. This statement seems to stay in conflict with the possibility to trade mapping effort for additional disambiguation effort and vice versa. But in fact, if a more restrictive, thus concrete parse syntax is used, semantics are put into the grammar. So semantics are shifted from the semantic evaluator to the parser and less disambiguation, which represents the addition of semantic information to the tree, is requires. On the other hand, additional mapping effort to refine already computed semantics without adding some additional semantic information, must be preformed. Analog, semantics can be shifted to the semantic evaluator by using a more ambiguous grammar for the parser and disambiguate the trees constructed afterwards.

Table 7.1: Comparing the mapping technique of chapter 5.3 and the disambiguation technique of chapter 7.2.1.

	mapping	disambiguation
purpose	refine the AST to become more comprehensible	transform a non-deterministic AST representing several possible AST structures to a deterministic one
technique	final AST is computed based on *mapping nodes* representing subtrees to map; *intermediate trees* are replaced by *complete final ASTs*	disambiguation attribute selects exactly one tree from an *ambiguous node's* children
semantics	since tree is only restructured no semantic changes	semantics are added to the tree by discarding erroneous interpretations
more effort to implement needed if	parser grammar is more restrictive, thus is more different from the abstract syntax	parser grammar is less restrictive, thus contains more ambiguities

It is important to keep in mind: Semantics **cannot** always be shifted from the semantic evaluator to the parser, since context-sensitive languages are no subset of the context-free languages. But the other way around does always work, thus semantics can always be shifted from the parser to the semantic evaluator. But it does not always make sense. The right tradeoff between the parser's syntax and the abstract syntax used for the semantic evaluation is an important quality criteria for compilers.

The *typedef problem* cannot be described in a context-free way, thus it cannot be shifted into the parser. Consequently, even the disambiguation technique described in this chapter can handle it, the mapping technique of chapter 5.3 cannot.

Table 7.1 summarizes the differences between mapping and disambiguation techniques.

For *JastAddC*, it is the best to stick to the standard's grammar for the parser and keep the mapping as it is. Only the artificial `typedef` token should be removed and an ordinary identifier used instead. To handle the resulting *typedef problem* the LALR(k) parser should be switched for a GLR parser and the disambiguation technique just presented can be used to disambiguate the parse forests produced. The advantages using the standard's grammar are worth the mapping effort. The standard's grammar is the back bone for *C* developers. It supports a well known base to talk about the language. So the compiler implementation can be mapped to the standard and extensions are more easy to develop, since extension developers are comfortable with the grammar used.

7.3 Constructing a Full Compiler: Generating Executeable Programs

To implement a code generation module the techniques used for already implemented parts can be reused. The *compiler-core* can be extended with an additional attribute `GenerateCode()` associated with each AST node, which triggers the generation of code for the subtree represented by the node. The code generation attribute of a node A, with the child nodes $B_1, B_2, \ldots, B_n$ can be implemented by calling the code generation attributes of $B_1, \ldots, B_n$ and enrich the result code by A's own code constructs[16]. Code generation attributes can check if the AST part they generate code for is correct by reusing the semantic constraint checking module. Exchanging the code generation module would be no problem.

An interesting possibility is to introduce an abstract machine, represented by an AST. An abstract machine AST consists only of *code unit nodes* and *machine instruction nodes*. *Machine instruction nodes* and *code unit nodes* share a common ancestor node type, the *abstract machine node* type. *Machine instruction nodes* correspond to specific machine instructions like `load` ***`Address`***, `store` ***`Address`***, `jumpz` ***`Address`***, `add` ***`value_1 value_2`*** etc. *Code unit nodes* simply contain an arbitrary number of ordered *abstract machine nodes* as child nodes. The `GenerateCode()` attributes could return such abstract machine ASTs[17]. Thus the `GenerateCode()` attributes would be some kind of *non-terminal attributes*, though instead of *JastAddC* ASTs they return abstract machine ASTs. Of course, the final real machine code generation now has to be handled by an attribute of the abstract machine AST, which can be done by another module specified for the compiler core of the abstract machine compiler[18]. The most interesting advantage is the possibility to introduce an optimization module interface for abstract machine ASTs. This way optimizers can be exchanged, depending on the destination machine and several optimizers can be combined, by adding each of them as an extension module. The idea to separate the code generation into its own compiler (the abstract machine AST) also seems to be a nice software design.

The *JastAddC* compiler would use the abstract machine compiler and if optimizations for the abstract machine code need additional program information, *JastAddC*'s AST can be made available for the abstract machine AST. To do so an inherited attribute, which broadcasts the associated *JastAddC* tree can be specified for the abstract machine's AST root node and the abstract machine AST can become part of *JastAddC*'s AST, decoupled by an abstract node type

[16]Similar to the semantic constraint checking approach presented in chapter 5.2.3. The main difference is, that no explicit local code generation attribute is needed, but instead the attribute traversing the AST and the local evaluation attribute will be merged into one code generation attribute.

[17]Similar to the attributes specified by the concrete to abstract syntax mapping module of chapter 5.3, which compute final AST parts.

[18]The abstract machine AST represents a compiler in the same way as *JastAddC* does. A compiler core for the abstract machine can be specified, so a modularization concept can be developed. *JastAddC*'s concepts can be applied to the abstract machine compiler, introducing the same advantages, especial extensibility.

`AbstractMachine`[19]. A bijective mapping between *JastAddC* AST nodes and abstract machine *code unit nodes* is possible. The association of *JastAddC* AST nodes to abstract machine *code unit nodes* is simple. The code generation attributes are representing it. To be able to associate abstract machine *code unit nodes* with *JastAddC* AST nodes it is sufficient to extend each abstract machine *code unit node* with an additional terminal child node, which represents a link from the *code unit node* to the AST node whose `GenerateCode()` attribute constructed it. Two alternative concepts for the additional terminal node's type exist. It can be an *inter-AST reference*[20] [51] terminal node, which points to the associated *JastAddC* AST node, or, if *JastAddC's* AST and the abstract machine AST have to be decoupled, an ordinary terminal node, which contains the node address of the associated *JastAddC* AST node. Both solutions are easily to implement[21].

It is important to acknowledge that *JastAdd*'s demand driven approach still holds for such a code generation design. Hence, the abstract machine AST can be constructed side effect free.

To illustrate the presented code generation concept consider the code generation for a *for statement* as presented in listing 7.6.

Listing 7.6: *C* code skeleton for a *compound statement* containing a *for statement*.

```
1 {
2     for (declaration; expression; expression)
3         statement
4     statement
5 }
```

First, the `GenerateCode()` attribute of the *for statement* calls `GenerateCode()` of the *Declaration*. Afterwards the `GenerateCode()` attribute of the first expression is called followed by a conditional jump machine instruction, which checks whether the value computed by the expression is 0 or not and if it is 0 a jump to the *statement* following the for is executed. Now, the `GenerateCode()` attribute of the *for body statement* is called. Than the `GenerateCode()` attribute of the *update expression* is called, followed by an unconditional jump machine instruction, which jumps just before the *condition expression* of the *for statement*. Figure 7.5 illustrates the *JastAddC* AST and the associated abstract machine AST generated by the `GenerateCode()` attributes. The thick blue arrows represent the links between *JastAddC*'s AST nodes and the code generated for these nodes. The angled dark blue rectangles represent *machine instruction nodes*. The `GenerateCode()` attribute is abbreviated with *GC()*.

[19] Similar to the error handling approach presented in chapter 5.2.1, where the error node tree is part of the AST, decoupled by the abstract `ErrorBuilder` node.

[20] An *inter-AST reference* is a terminal node of an AST A_1, which points to a node of another AST A_2.

[21] The node/node address to associate with each *code unit node* is known during the construction of it. If *inter-AST references* are used, simply the node the `CodeGeneration()` attribute is associated with is the *inter-AST reference's* value. If ordinary terminal nodes representing a node address are used, simply the `NodeAddress()` attribute can be called.

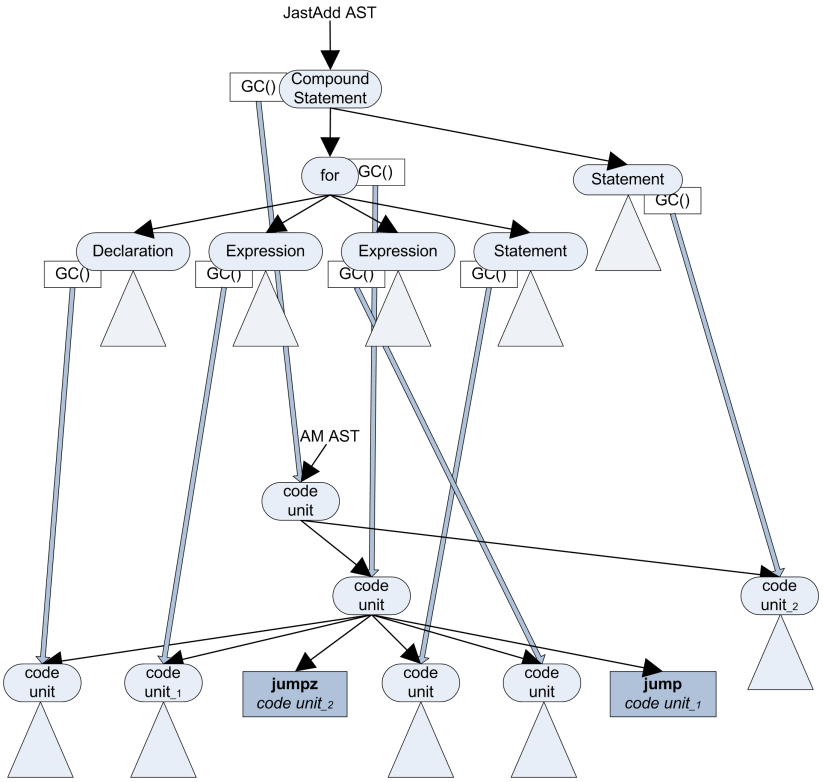

Figure 7.5: *JastAddC* AST with associated generated code, the abstract machine AST.

Of course, the AST constructed for the input program can also be optimized and normalized for the code generation. [18] presents such code generation AST preparations based on *JastAdd*.

For further details about code generation independent from *JastAdd* and *C*, [40] and [49] provide a good overview, whereas [40]'s focus is on the hardware design. Nevertheless, a complete working machine is developed together with it's instruction set, so the transformation of source code into machine code becomes obvious. Additionally interesting themes for compiler developers like code optimizations (e.g. *loop unrolling*) are discussed.

7.4 Suggesting a further work schedule

In the following a rough further development schedule for the next implementation steps of *JastAddC* will be suggested. The schedule is with no means complete. It is only a framework. A lot of smaller semantics of language constructs can be freely specified in-between.

Further development should start with the implementation of the type analysis. A constant expression check should be added and analysis for *lvalues*, *rvalues* and *modifyable lvalues*. Also the comparison between types should be

added. The checks for type errors in expressions[22] can be implemented after the *type promotions*. Also the implicit *type promotions* of arrays and functions[23] should be specified. No external translation unit type semantics should be added yet.

Afterwards, the implicit information retrieval module should be realized. This module triggers automatic tree transformations (rewrites) to add implicit given semantics. E.g. a rewrite adding the implicitly given type of enumeration constants, which is `int`, is already implemented as well as a rewrite, which assigns each enumeration constant without an explicit initializing *constant-expression* (rule 6.7.2.2) its implicitly defined value. But especially initializer semantics are not specified yet. The standard specifies implicit the meaning of *designations* (rule 6.7.8,) and the values for elements, which have no value assigned during an initialization. These semantics have to be added using rewrites, similar to the *enum-specifier* semantics already implemented.

Now, the switch of the parsing technique used should be done. The *Beaver* parser should be exchanged for a SGLR parser and the lexical specifications should be integrated into the parser's specifications. Maybe a few lexers of the lexer stack (see chapter 4.1) have to be kept and executed before the SGLR parser gets the program input. E.g. It may not be possible to integrate the line splicing lexer's semantics into the SGLR parser specifications, since to many possible token splicings exist. Also, the parser may become to slow. After successfully switching the parser, the *typedef problem* should be solved using the techniques described in section 7.2.1. To do so, the *compiler core* should be extended by another module, the *typedef disambiguation module*.

The next big development step should be the implementation of a *preprocessor compiler*. It has to be checked, how such a compiler can be integrated with the SGLR parser. Unfortunately to the current lexer stack solution of chapter 4.1 the *preprocessor compiler* cannot simply be pushed in-between the lexers of the lexer stack if a SGLR parser is used, since the SGLR parser replaces the current lexer stack and does not support a lexer stack itself. A concept to avoid double lexing has to be developed.

Maybe, as indicated above, the preprocessor should be its own compiler, thus it should be developed like *JastAddC* with its own AST and attributes, which trigger the actual preprocessing. The preprocessor's output computed by its attributes should be the preprocessed program AST. Thus, *JastAddC's* parser is replaced by the *preprocessor compiler*, which constructs *JastAddC* ASTs. To do so, the *preprocessor compiler* now contains a SGLR parser constructing a preprocessor AST. The preprocessor AST's attributes transform the preprocessor AST to a preprocessed *JastAddC* AST. Such a solution is best achieved by reusing the SGLR parser solution developed in the step before as the basis for the preprocessor implementation. Implementing the preprocessor requires to extend the *JastAddC* AST, so several translation units can be child nodes of the `Compilation` root node. It is the `#include` directive's task to control the

[22] The type error checks in expressions should be part of the constraint checking module presented in chapter 5.2.3.

[23] Arrays are treated like pointers to the type of the array elements and function names are treated like pointers, which type is the one of the function definition.

construction of the `TranslationUnit` subtrees.

Finally, the external translation unit semantics can be specified.

Chapter 8

Conclusions

The thesis illustrated the *specification-driven* development of extensible compilers and presented an example implementation for *C99*, the *JastAddC* compiler front-end. Figure 8.1 summarizes the handled topics. First, the term

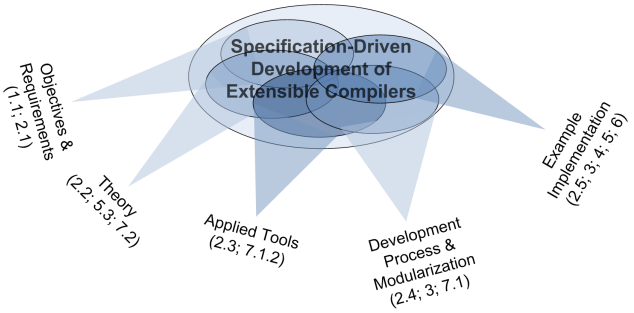

Figure 8.1: *Specification-Driven* Development of Extensible Compilers: Topics handled throughout the thesis and the chapters presenting them.

specification-driven compiler development has been introduced. Especially, the requirements for the specification languages and consequently tools have been presented. It has been highlighted, that because of automatically unification functionalities supported by the formalisms used to implement a compiler, the *specification-driven* compiler development permits the easy extension of compilers. Afterwards, the specification formalisms, *regular expressions* for the lexical analysis, CFGs/EBNF for the syntactical analysis and ReCRAGs for the semantical analysis, used to implement *JastAddC* have been introduced. A short presentation of the tools *JFlex*, *Beaver* and *JastAdd* realizing these techniques followed. Finally, chapter 2 concluded with a small example compiler, a calculator for expressions, which illustrated the *specification-driven* development process and the integration of the presented tools.

The next chapter presented a modularization technique based on specifications, which permits the separate specification of a compiler's different tasks: The modularization across concerns. Its interaction with the well-known modularization across *compilation phases* has been shown and its requirements on the formalisms, applied tools are based on, have been justified. It has been elab-

orated, why lexer generators and deterministic parser generators[1] cannot support the modularization across concerns, whereas ReCRAGs support it. Also a concept to organize the interactions between separate modules has been presented: The *compiler-core*.

Afterwards, *JastAddC's* lexical and syntactical components have been presented. It has been shown, how the lexical and syntactical tasks of realistic languages can be specified. To handle the stepwise recognition of a language's tokens, as it is often needed for realistic languages, the *lexer stack* concept has been introduced. Also, the *typedef problem* has been used to illustrate the hassle, dependency cycles between *compilation phases* introduce.

Chapter 5 presented *JastAddC's* semantical components. A generic error handling concept based on ReCRAGs, which avoids the mixing of actual semantical computations[2], the error representation and the error recognition/reporting has been presented. It has been shown, how *non-null reference attributes* can be used to lift undefined values of relationship computations to defined ones, such that semantical computations based on these relationship computations can be implemented without bothering about a relationship computation's error cases. Afterwards, *JastAddC's* implementation of *non-null reference attributes* and *error nodes* realizing the generic error handling concept has been investigated. Finally, a technique to map ASTs of a structure A to ASTs of structure B based on ReCRAGs has been presented, so called *ReCRAG AST mappings*. Such mappings can be used to solve the well-known problem to transform an AST the parser constructed into the AST the semantical computations expect. Hence, semantical computations can be specified using the most appropriate AST, whereas the parser can still be implemented with respect to the language's most appropriate CFG. The properties correctness, confluence and termination for *ReCRAG AST mappings* have been proved.

Finally, chapter 7 suggested SGLR parsing as a solutions for the lexical and syntactical modularization problems, such that the modularization across concerns becomes possible for all *compilation-phases*. Also, a disambiguation concept, based on ReCRAGs and *ambiguous nodes*, to break dependency cycles between *compilation-phases* in a well defined way, has been investigated. Last but not least, an advanced code generation concept, where the code to generate is represented as its own *abstract machine AST* has been suggested. The advantage of such *abstract machine* compilers is, that they can be developed in a *specification-driven* extensible way, similar like the base compiler which constructs their AST.

Related Work

JastAddC's implementation approach is very uncommon for *C*. Most *C* compilers are implemented in *C* itself. A typical, very well documented *C* implementation is *lcc*. [16] presents *lcc's* implementation in detail. The differences to *JastAddC* are obvious. The *lcc* compiler is not generated, but instead completely handcrafted. Also, its implementation does not consider language extensions, except the exchange of back-ends. Hence, *lcc* supports a very well designed

[1] e.g. LR(k)

[2] as specified in the language's specification

back-end architecture and many different back-ends exist. However, as presented in chapter 7.3, in *JastAddC* the support for code generation would just be one possible compiler extension and is nothing special. *JastAddC's* extensibility is simply uncommon in handcrafted *C* compilers.

Often compiler extensions introduce additional program analysis and transformations in a monotonic way on top of a language. A typical *C* example is *CIL*[3] [38], an analysis and source to source transformation *C* compiler front-end. The objective of *CIL* is to analyse a *C* program's constructs, break them down into simpler ones (some kind of normalization) and print out the resulting *C* program. Based on such normalized *CIL* representations additional extensions can perform their analysis. E.g. *CCure* inserts runtime checks in *C* programs to prevent memory safety violations. If a memory safety violation is encountered, the *C* program is simply stopped. In contrast to *JastAddC*, *CIL* works on a lower level than ASTs, though its program presentations are still more high-level than typical intermediate languages[4]. *CIL* is implemented using *OCaml* [22]. In practice *CIL* extensions are ordinary *OCaml* programs, which use *CIL's* API functionalities to manipulate input programs' *CIL* representation. Of course, such an approach is not *specification-driven*. Additionally, it is in the nature of *CIL's* intermediate representations, that high level analysis cannot be supported or only suboptimal. Hence, *CIL* can be used to introduce very specific *C* analysis, which is typical for *C* code analyser. In contrast, *JastAddC's* aim is to provide a *C* front-end, which can be easily extended to support a broad range of different additional program analysis. An interesting further project would be, to implement a *JastAddC* extension, which constructs *CIL* representations.

A compiler project implemented in a very similar way as *JastAddC* is the *JastAddJ* [11, 18] *Java* compiler. In fact, *JastAddC's* implementation is highly inspired by *JastAddJ*. In contrast to *JastAddC*, *JastAddJ* is complete. Its a high quality, heavily tested, extensible *Java* compiler. Many ReCRAG techniques *JastAddC* is based on have been demonstrated by *JastAddJ*. Nevertheless, *JastAddC* contributes a few additional concepts, like the *compiler-core*, the generic error handling, the *ReCRAG AST mappings* and a first survey of a disambiguation based on ReCRAGs and *ambiguous nodes*.

[3] *C* Intermediate Language
[4] E.g. three-address code [40].

Appendix A

ISO/IEC 9899:1999 Grammar

The following appendix contains the standard's official grammar [23]. It is *LALR(1)*, if *typedef-name* (6.7.7) is treated as token. If rule 6.7.7 is implemented as given, i.e. *typedef-names* are simply *identifiers*, 7 shift-reduce and 13 reduce-reduce conflicts result. The standard's grammar, with the separation of *identifiers* and *typedef-names*, is the grammar used for *JastAddC's Beaver* specification.

Language syntax summary

A.1 Lexicalgrammar

A.1.1 Lexicalelements

(6.4) *token:*

keyword
identifier
constant
string-literal
punctuator

(6.4) *preprocessing-token:*

header-name
identifier
pp-number
character-constant
string-literal
punctuator
each non-white-space character that cannot be one of the above

A.1.2 Keywords

(6.4.1) *keyword:* one of

`auto`	**`enum`**	**`restrict`**	**`unsigned`**
`break`	**`extern`**	**`return`**	**`void`**
`case`	**`float`**	**`short`**	**`volatile`**
`char`	**`for`**	**`signed`**	**`while`**
`const`	**`goto`**	**`sizeof`**	**`_Bool`**
`continue`	**`if`**	**`static`**	**`_Complex`**
`default`	**`inline`**	**`struct`**	**`_Imaginary`**
`do`	**`int`**	**`switch`**	
`double`	**`long`**	**`typedef`**	
`else`	**`register`**	**`union`**	

A.1.3 Identifiers

(6.4.2.1) *identifier:*
identifier-nondigit
identifier identifier-nondigit
identifier digit

(6.4.2.1) *identifier-nondigit:*
nondigit
universal-character-name
other implementation-defined characters

(6.4.2.1) *nondigit:* one of
`_abcdefghijklm`
`nopqrstuvwxyz`
`ABCDEFGHIJKLM`
`NOPQRSTUVWXYZ`

(6.4.2.1) *digit:* one of
`0123456789`

A.1.4 Universal character names

(6.4.3) *universal-character-name:*
`\u` *hex-quad*
`\U` *hex-quad hex-quad*

(6.4.3) *hex-quad:*
hexadecimal-digit hexadecimal-digit
hexadecimal-digit hexadecimal-digit

A.1.5 Constants

(6.4.4) *constant:*
integer-constant
floating-constant
enumeration-constant
character-constant

(6.4.4.1) *integer-constant:*
decimal-constant integer-suffix $_{opt}$
octal-constant integer-suffix $_{opt}$
hexadecimal-constant integer-suffix $_{opt}$

(6.4.4.1) *decimal-constant:*
nonzero-digit
decimal-constant digit

(6.4.4.1) *octal-constant:*

`0`

octal-constant octal-digit

(6.4.4.1) *hexadecimal-constant:*

hexadecimal-prefix hexadecimal-digit

hexadecimal-constant hexadecimal-digit

(6.4.4.1) *hexadecimal-prefix:* one of

`0x 0X`

(6.4.4.1) *nonzero-digit:* one of

`123456789`

(6.4.4.1) *octal-digit:* one of

`01234567`

(6.4.4.1) *hexadecimal-digit:* one of

`0123456789`

`abcdef`

`ABCDEF`

(6.4.4.1) *integer-suffix:*

unsigned-suffix long-suffix $_{opt}$

unsigned-suffix long-long-suffix

long-suffix unsigned-suffix $_{opt}$

long-long-suffix unsigned-suffix $_{opt}$

(6.4.4.1) *unsigned-suffix:* one of

`uU`

(6.4.4.1) *long-suffix:* one of

`lL`

(6.4.4.1) *long-long-suffix:* one of

`ll LL`

(6.4.4.2) *floating-constant:*

decimal-floating-constant

hexadecimal-floating-constant

(6.4.4.2) *decimal-floating-constant:*

fractional-constant exponent-part $_{opt}$ *floating-suffix*$_{opt}$

digit-sequence exponent-part floating-suffix $_{opt}$

(6.4.4.2) *hexadecimal-floating-constant:*
hexadecimal-prefix hexadecimal-fractional-constant
binary-exponent-part floating-suffix $_{opt}$
hexadecimal-prefix hexadecimal-digit-sequence
binary-exponent-part floating-suffix $_{opt}$

(6.4.4.2) *fractional-constant:*
digit-sequence$_{opt}$ **.** *digit-sequence*
digit-sequence **.**

(6.4.4.2) *exponent-part:*
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

(6.4.4.2) *sign:* one of
+-

(6.4.4.2) *digit-sequence:*
digit
digit-sequence digit

(6.4.4.2) *hexadecimal-fractional-constant:*
hexadecimal-digit-sequence$_{opt}$ **.**
hexadecimal-digit-sequence
hexadecimal-digit-sequence **.**

(6.4.4.2) *binary-exponent-part:*
p *sign*$_{opt}$ *digit-sequence*
P *sign*$_{opt}$ *digit-sequence*

(6.4.4.2) *hexadecimal-digit-sequence:*
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

(6.4.4.2) *floating-suffix:* one of
flFL

(6.4.4.3) *enumeration-constant:*
identifier

(6.4.4.4) *character-constant:*
' *c-char-sequence* **'**
L' *c-char-sequence* **'**

(6.4.4.4) *c-char-sequence:*
c-char
c-char-sequence c-char

(6.4.4.4) *c-char:*
anymember of the source character set except
the single-quote `'`,backslash `\`,ornew-line character
escape-sequence

(6.4.4.4) *escape-sequence:*
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence
universal-character-name

(6.4.4.4) *simple-escape-sequence:* one of
`\' \" \? \\`
`\a \b \f \n \r \t \v`

(6.4.4.4) *octal-escape-sequence:*
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

(6.4.4.4) *hexadecimal-escape-sequence:*
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

A.1.6 Stringliterals

(6.4.5) *string-literal:*
`"` *s-char-sequence*$_{opt}$ `"`
`L"` *s-char-sequence*$_{opt}$ `"`

(6.4.5) *s-char-sequence:*
s-char
s-char-sequence s-char

(6.4.5) *s-char:*
anymember of the source character set except
the double-quote `"`,backslash `\`,ornew-line character
escape-sequence

A.1.7 Punctuators

(6.4.6) *punctuator:* one of

```
[] () {} . ->
++ -- & * + - ~ !
/ % << >> < > <= >= == != ^ | && ||
? : ; .  ..
= *= /= %= += -= <<= >>= &= ^= |=
, ###
<: :> <% %> %: %:%:
```

A.1.8 Headernames

(6.4.7) *header-name:*

`<` *h-char-sequence* `>`
`"` *q-char-sequence* `"`

(6.4.7) *h-char-sequence:*

h-char
h-char-sequence h-char

(6.4.7) *h-char:*

anymember of the source character set except
the new-line character and `>`

(6.4.7) *q-char-sequence:*

q-char
q-char-sequence q-char

(6.4.7) *q-char:*

anymember of the source character set except
the new-line character and `"`

A.1.9 Preprocessing numbers

(6.4.8) *pp-number:*

digit
`.` *digit*
pp-number digit
pp-number identifier-nondigit
pp-number `e` *sign*
pp-number `E` *sign*
pp-number `p` *sign*
pp-number `P` *sign*
pp-number `.`

A.2 Phrasestructuregrammar

A.2.1 Expressions

(6.5.1) *primary-expression:*
identifier
constant
string-literal
`(` *expression* `)`

(6.5.2) *postfix-expression:*
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `(` *argument-expression-list*$_{opt}$ `)`
postfix-expression `.` *identifier*
postfix-expression `->` *identifier*
postfix-expression `++`
postfix-expression `--`
`(` *type-name* `){` *initializer-list* `}`
`(` *type-name* `){` *initializer-list* `,}`

(6.5.2) *argument-expression-list:*
assignment-expression
argument-expression-list `,` *assignment-expression*

(6.5.3) *unary-expression:*
postfix-expression
`++` *unary-expression*
`--` *unary-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof` `(` *type-name* `)`

(6.5.3) *unary-operator:* one of
`&*+-~!`

(6.5.4) *cast-expression:*
unary-expression
`(` *type-name* `)` *cast-expression*

(6.5.5) *multiplicative-expression:*
cast-expression
multiplicative-expression `*` *cast-expression*
multiplicative-expression `/` *cast-expression*
multiplicative-expression `%` *cast-expression*

(6.5.6) *additive-expression:*

multiplicative-expression
additive-expression **+** *multiplicative-expression*
additive-expression **-** *multiplicative-expression*

(6.5.7) *shift-expression:*

additive-expression
shift-expression **<<** *additive-expression*
shift-expression **>>** *additive-expression*

(6.5.8) *relational-expression:*

shift-expression
relational-expression **<** *shift-expression*
relational-expression **>** *shift-expression*
relational-expression **<=** *shift-expression*
relational-expression **>=** *shift-expression*

(6.5.9) *equality-expression:*

relational-expression
equality-expression **==** *relational-expression*
equality-expression **!=** *relational-expression*

(6.5.10) *AND-expression:*

equality-expression
AND-expression **&** *equality-expression*

(6.5.11) *exclusive-OR-expression:*

AND-expression
exclusive-OR-expression **^** *AND-expression*

(6.5.12) *inclusive-OR-expression:*

exclusive-OR-expression
inclusive-OR-expression **|** *exclusive-OR-expression*

(6.5.13) *logical-AND-expression:*

inclusive-OR-expression
logical-AND-expression **&&** *inclusive-OR-expression*

(6.5.14) *logical-OR-expression:*

logical-AND-expression
logical-OR-expression **||** *logical-AND-expression*

(6.5.15) *conditional-expression:*

logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

(6.5.16) *assignment-expression:*
conditional-expression
unary-expression assignment-operator assignment-expression

(6.5.16) *assignment-operator:* one of
`=*=/=%=+=-=<` `<= >>= &= ^= |=`

(6.5.17) *expression:*
assignment-expression
expression `,` *assignment-expression*

(6.6) *constant-expression:*
conditional-expression

A.2.2 Declarations

(6.7) *declaration:*
declaration-specifiersi nit-declarator-list$_{opt}$ `;`

(6.7) *declaration-specifiers:*
storage-class-specifier declaration-specifiers $_{opt}$
type-specifier declaration-specifiers $_{opt}$
type-qualifier declaration-specifiers $_{opt}$
function-specifier declaration-specifiers $_{opt}$

(6.7) *init-declarator-list:*
init-declarator
init-declarator-list `,` *init-declarator*

(6.7) *init-declarator:*
declarator
declarator `=` *initializer*

(6.7.1) *storage-class-specifier:*
`typedef`
`extern`
`static`
`auto`
`register`

(6.7.2) *type-specifier:*

void
char
short
int
long
float
double
signed
unsigned
_Bool
_Complex
struct-or-union-specifier
enum-specifier
typedef-name

(6.7.2.1) *struct-or-union-specifier:*

struct-or-union identifier $_{opt}$ **{** *struct-declaration-list* **}**
struct-or-union identifier

(6.7.2.1) *struct-or-union:*

struct
union

(6.7.2.1) *struct-declaration-list:*

struct-declaration
struct-declaration-list struct-declaration

(6.7.2.1) *struct-declaration:*

specifier-qualifier-list struct-declarator-list **;**

(6.7.2.1) *specifier-qualifier-list:*

type-specifier specifier-qualifier-list $_{opt}$
type-qualifier specifier-qualifier-list $_{opt}$

(6.7.2.1) *struct-declarator-list:*

struct-declarator
struct-declarator-list **,** *struct-declarator*

(6.7.2.1) *struct-declarator:*

declarator
declarator$_{opt}$ **:** *constant-expression*

(6.7.2.2) *enum-specifier:*

enum *identifier*$_{opt}$ **{** *enumerator-list* **}**
enum *identifier*$_{opt}$ **{** *enumerator-list* **,}**
enum *identifier*

(6.7.2.2) *enumerator-list:*

enumerator
enumerator-list **,** *enumerator*

(6.7.2.2) *enumerator:*

enumeration-constant
enumeration-constant **=** *constant-expression*

(6.7.3) *type-qualifier:*

const
restrict
volatile

(6.7.4) *function-specifier:*

inline

(6.7.5) *declarator:*

pointer$_{opt}$ *direct-declarator*

(6.7.5) *direct-declarator:*

identifier
(*declarator* **)**
direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
direct-declarator **[static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list*$_{opt}$ ***]**
direct-declarator **(** *parameter-type-list* **)**
direct-declarator **(** *identifier-list*$_{opt}$ **)**

(6.7.5) *pointer:*

***** *type-qualifier-list*$_{opt}$
***** *type-qualifier-list*$_{opt}$ *pointer*

(6.7.5) *type-qualifier-list:*

type-qualifier
type-qualifier-list type-qualifier

(6.7.5) *parameter-type-list:*

parameter-list
parameter-list **, . . .**

(6.7.5) *parameter-list:*
parameter-declaration
parameter-list **,** *parameter-declaration*

(6.7.5) *parameter-declaration:*
declaration-specifiersd eclarator
declaration-specifiersa bstract-declarator$_{opt}$

(6.7.5) *identifier-list:*
identifier
identifier-list **,** *identifier*

(6.7.6) *type-name:*
specifier-qualifier-list abstract-declarator $_{opt}$

(6.7.6) *abstract-declarator:*
pointer
pointer$_{opt}$ *direct-abstract-declarator*

(6.7.6) *direct-abstract-declarator:*
(*abstract-declarator* **)**
direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
direct-abstract-declarator$_{opt}$ **[static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list* **static** *assignment-expression* **]**
direct-abstract-declarator$_{opt}$ **[*]**
direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

(6.7.7) *typedef-name:*
identifier

(6.7.8) *initializer:*
assignment-expression
{ *initializer-list* **}**
{ *initializer-list* **,}**

(6.7.8) *initializer-list:*
designation$_{opt}$ *initializer*
initializer-list **,** *designation*$_{opt}$ *initializer*

(6.7.8) *designation:*
designator-list **=**

(6.7.8) *designator-list:*

designator
designator-list designator

(6.7.8) *designator:*

`[` *constant-expression* `]`
`.` *identifier*

A.2.3 Statements

(6.8) *statement:*

labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

(6.8.1) *labeled-statement:*

identifier `:` *statement*
`case` *constant-expression* `:` *statement*
`default` `:` *statement*

(6.8.2) *compound-statement:*

`{` *block-item-list*$_{opt}$ `}`

(6.8.2) *block-item-list:*

block-item
block-item-list block-item

(6.8.2) *block-item:*

declaration
statement

(6.8.3) *expression-statement:*

expression$_{opt}$ `;`

(6.8.4) *selection-statement:*

`if` `(` *expression* `)` *statement*
`if` `(` *expression* `)` *statement* **`else`** *statement*
`switch` `(` *expression* `)` *statement*

(6.8.5) *iteration-statement:*

while (*expression*) *statement*
do *statement* **while** (*expression*) ;
for ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$) *statement*
for (*declaration expression* $_{opt}$; $expression_{opt}$) *statement*

(6.8.6) *jump-statement:*

goto *identifier* ;
continue ;
break ;
return $expression_{opt}$;

A.2.4 External definitions

(6.9) *translation-unit:*

external-declaration
translation-unit external-declaration

(6.9) *external-declaration:*

function-definition
declaration

(6.9.1) *function-definition:*

declaration-specifiersd eclarator declaration-list $_{opt}$ *compound-statement*

(6.9.1) *declaration-list:*

declaration
declaration-list declaration

A.3 Preprocessing directives

(6.10) *preprocessing-file:*

$group_{opt}$

(6.10) *group:*

group-part
group group-part

(6.10) *group-part:*

if-section
control-line
text-line
*non-directive*

(6.10) *if-section:*

if-group elif-groups $_{opt}$ *else-*$group_{opt}$ *endif-line*

(6.10) *if-group:*

#if *constant-expression new-line group* $_{opt}$
#ifdef *identifier new-line group* $_{opt}$
#ifndef *identifier new-line group* $_{opt}$

(6.10) *elif-groups:*

elif-group
elif-groups elif-group

(6.10) *elif-group:*

#elif *constant-expression new-line group* $_{opt}$

(6.10) *else-group:*

#else *new-line group* $_{opt}$

(6.10) *endif-line:*

#endif *new-line*

(6.10) *control-line:*

#include *pp-tokens new-line*
#define *identifier replacement-list new-line*
#define *identifier lparen identifier-list* $_{opt}$ **)**
replacement-list new-line
#define *identifier lparen* **...** **)** *replacement-list new-line*
#define *identifier lparen identifier-list* **, ...** **)**
replacement-list new-line
#undef *identifier new-line*
#line *pp-tokens new-line*
#error *pp-tokens*$_{opt}$ *new-line*
#pragma *pp-tokens*$_{opt}$ *new-line*
*new-line*

(6.10) *text-line:*

pp-tokens$_{opt}$ *new-line*

(6.10) *non-directive:*

pp-tokens new-line

(6.10) *lparen:*

a **(** character not immediately preceded by white-space

(6.10) *replacement-list:*

pp-tokens$_{opt}$

(6.10) *pp-tokens:*

preprocessing-token

pp-tokens preprocessing-token

(6.10) *new-line:*

the new-line character

Appendix B

JastAddC's Compiler-Core

The following appendix presents *JastAddC's* complete *compiler-core*. With respect to chapter 3, it consists of a *core-AST* and *core-concerns*. Both together form the basis against which the modules implementing the compiler and further compiler extensions are specified.

B.1 *Core-AST*

Listing B.1 presents the *compiler-core's complete final AST* specification and listing B.2 presents its *intermediate AST* specification. For details about the relationship between *intermediate* and *complete final AST* see chapter 5.3.

Listing B.1: *Compiler-core*: *Complete final AST* specification.

```
/**
 * Final AST specification (see chapter 5.3.2) for the JastAddC C99 compiler.
 *
 * IMPORTANT: The names of intermediate node types start with "_i_". The
 *  names of relationship and mapping node types start with "_r_" and
 *  "_m_" respectively. Final node types start with an upper case letter
 *  For deteails about these nodes' purpose see chapter 5.3.2.
 *
 * @author C. Brger
 */

/* ****************************** Terminals ****************************** */

/**
 * In contrast to the C99 standard, JastAddC's string literals are constants.
 */
abstract Constant:Expression ::=
        <Lexem:String>;

    IntegerConstant:Constant ::=
            <NumericValue:String>
            Radix:NumeralSystemRadix
            Type:PrimitiveType;
    FloatingConstant:Constant ::=
            <WholeNumberPart:String>
            <FractionalPart:String>
            <Exponent:String>
            Radix:NumeralSystemRadix
```

```
29                  Type:PrimitiveType;
30          CharacterConstant:Constant ::=
31                  <CharacterValue:String>
32                  <IsWideCharacter:boolean>;
33          StringLiteral : Constant ::=
34                  <StringValue:String>
35                  <IsWideString:boolean>;
36
37  abstract NumeralSystemRadix;
38
39          NumeralSystemOctal:NumeralSystemRadix;
40          NumeralSystemHexadecimal:NumeralSystemRadix;
41          NumeralSystemDecimal:NumeralSystemRadix;
42
43  abstract StorageClassSpecifier:_i_declaration_specifiers;
44
45          Auto:StorageClassSpecifier;
46          Extern:StorageClassSpecifier;
47          Register : StorageClassSpecifier ;
48          Static : StorageClassSpecifier ;
49          Typedef:StorageClassSpecifier;
50
51  abstract FunctionSpecifier:_i_declaration_specifiers;
52
53          Inline : FunctionSpecifier ;
54
55  abstract TypeQualifier:_i_specifier_qualifier;
56
57          Const:TypeQualifier;
58          Restrict : TypeQualifier;
59          Volatile : TypeQualifier;
60
61   Identifier  ::=
62              <Value:String>;
63
64  /* ****************** External Definitions [A.2.4;p.416] ******************* */
65
66  Compilation ::=
67              TranslationUnit
68              ErrorBuilder;
69
70  /**
71   * Needed for JastAddC's generic error handling approach
72   * (see chapters 5.1.2 and 5.2.1).
73   */
74  abstract ErrorBuilder;
75
76  TranslationUnit ::=
77              Declaration:_r_DeclarationCollection*;
78
79  /* ********************** Declarations [A.2.2;p.411] ********************** */
80
81  abstract _r_DeclarationCollection:Statement;
82
83          /**
84           * A single C99 declaration can declare several entities . Each entity
85           * declared will be represented as it 's own Declaration node.
86           */
87          DeclarationCollection:_r_DeclarationCollection ::=
88                  Declaration*;
89
90  /**
```

```
 * In contrast to the standard, JastAddC models declarations,
 * function definitions and function prototypes just as declarations:
 * - Definitions are declarations with a binded name, except function
 *   definitions additionally must have an initializer representing the
 *   function's body.
 * - Function prototypes are declarations with function type and without
 *   an initializer .
 *
 * PARSER-CONSTRAINT: In C99, declarations are no statements. The parser
 *  must guarantee, that declarations do not occur where only statements
 *  are permitted. E.g.: The C99 grammar does not permit constructs like
 *
 *  Label: int a;    // Error
 *  if (i)
 *      int i;       // Error
 *  switch(i)
 *      int i=2;     // Error
 *  do
 *      int i=2;     // Error
 *  while(!i);
 *  while (i--)
 *      int i=2;     // Error
 *  for (i=3;i>0;i--)
 *      int i;       // Error
 */
 Declaration:Statement ::=
        StorageClassSpecifier*
        FunctionSpecifier*
        Type:_r_Type
        [BindedName:Identifier]
        [ Initializer ];

abstract Initializer ;

    FunctionDefinitionInitializer : Initializer ::=
            Body:CompoundStatement;

   abstract NormalInitializer: Initializer ;

        /**
         * PARSER-CONSTRAINT: Comma expressions are not permitted as Value.
         */
        AtomicInitializer : NormalInitializer ::=
                Value:Expression;
        CompoundInitializer:NormalInitializer ::=
                InitializerElement *;

        /**
         * REWRITE: After parsing, initializer elements have no designator and
         *  consequently no designation. The C99 standard defines the current
         *  designator and its designation (i.e. the referenced object) of
         *   initializers implicitely ([6.7.8; p. 125]).
         */
        abstract InitializerElement ::=
                [DesignatorAccess];

            InnerInitialization : InitializerElement ::=
                    CompoundInitializer;
            ElementInitialization : InitializerElement ::=
                    Value:Expression;

/* ************************* Types [A.2.2;p.411] ************************* */
```

```
153
154 abstract _r_Type;
155
156 /**
157  * Type nodes represent "type-names" ([6.7.6];p. 122]).
158  */
159 abstract Type:_r_Type ::=
160         TypeQualifier*;
161
162     /**
163      * AtomicType nodes represent "type-specifier" ([6.7.2];p.99).
164      *
165      * All Type implementations are recursive, except AtomicTypes. This
166      * guarantees, that "at least one type specifier shall be given in the
167      * declaration specifiers in each declaration, and in the specifier-
168      * qualifier list in each struct declaration and type name"
169      * ([6.7.2:2; p.99]).
170      *
171      * AtomicTypes can represent:
172      * - A Type created by typedef
173      * - A structure, union or enum tag
174      * - A standard type, in JastAddC called primitive type
175      */
176     abstract AtomicType:Type;
177
178         /**
179          * Primitive types model:
180          * - void
181          * - Character types ([6.2.5:15;p. 35])
182          * - Standard signed integer types ([6.2.5:4; p.33])
183          * - Standard unsigned integer types ([6.2.5:6; p.33])
184          * - Standard real floating types ([6.2.5:10; p.34])
185          * - Standard complex types ([6.2.5:11;p.34])
186          *
187          * PARSER-CONSTRAINT / REWRITE: The parser guarantees following
188          * rewrites ([6.7.2:2; p.99]):
189          * 1) short                --> signed short int
190          *    signed short         --> signed short int
191          *    short int            --> signed short int
192          * 2) unsigned short       --> unsigned short int
193          * 3) int                  --> signed int
194          *    signed               --> signed int
195          * 4) unsigned             --> unsigned int
196          * 5) long                 --> signed long int
197          *    signed long          --> signed long int
198          *    long int             --> signed long int
199          * 6) unsigned long        --> unsigned long int
200          * 7) long long            --> signed long long int
201          *    signed long long     --> signed long long int
202          *    long long int        --> signed long long int
203          * 8) unsigned long long   --> unsigned long long int
204          */
205         PrimitiveType:AtomicType ::=
206                 <Type:jastaddc.semantics.PrimitiveTypes>;
207         TypedefTypeReference:AtomicType ::=
208                 ReferenceName:Identifier;
209         StructureTypeReference:AtomicType ::=
210                 ReferenceName:Identifier;
211         UnionTypeReference:AtomicType ::=
212                 ReferenceName:Identifier;
213         EnumTypeReference:AtomicType ::=
214                 ReferenceName:Identifier;
```

```
/**
 * To represent multidimensional arrays an Array can have another Array as
 * ElementType. An array type's first dimension can be left to represent an
 * incomplete array type ([6.2.5:22; p.36], [6.7.5.2:4; p.116]).
 *
 * IMPORTANT: Variable length array types are NOT supported.
 *
 * REWRITE: If an incomplete array type is declared and an initialization
 *  is given, a rewrite will be executed, setting the array's first
 *  dimension to the appropriate size ([6.7.8:22; p. 127]). E.g.:
 *
 *  int a[] = {0,0,0};          --> int a[3] = {0,0,0};
 *  int a[][2] = {1,2,3};       --> int a[2][2] = {{1,2},{3,0}};
 */
Array:Type ::=
        ElementType:_r_Type
        [Dimension:Expression];
Pointer:Type ::=
        Type:_r_Type;

/**
 * If no parameter declarations are specified (empty list) the function
 * has no parameter specification. If the function has no parameters,
 * the parameter declaration list contains one element, the declaration
 * of a nameless parameter of type void ([6.7.5.3:10; p. 119]).
 *
 * IMPORTANT: Static storage class specifiers in function array parameters
 *  are NOT supported. E.g.: void func(int array[static])
 *
 * REWRITE: An array parameter of type T has to be rewritten to a qualified
 *  pointer to T ([6.7.5.3:7; p.119]). The pointer's type qualifiers are
 *  those within the array's brackets. E.g.: Assume a function fun with one
 *  parameter para of type constant pointer to a pointer to integer
 *
 *  void fun(int* para[const]); --> void fun(int** const para);
 *
 * PARSER-CONSTRAINT / REWRITE: Old style function definitions have to be
 *  rewritten to new ones. E.g.:
 *
 *  void func(a) int a; {...}    --> void func(int a) {...}
 */
abstract FunctionType:Type ::=
        ReturnType:_r_Type
        Parameter:Declaration*;

    NormalFunction:FunctionType;
    VariadicFunction:FunctionType;

abstract TagedType:Type ::=
        [Tag: Identifier ];

    Enumeration:TagedType ::=
            EnumerationElement*;

    /**
     * REWRITE: The value of an enumeration is implicitly given
     *  ([6.7.2.2;3; p.105]).
     *
     * REWRITE: The type of enumeration elements is implicitly given
     *  to be int ([6.7.2.2:3; p.105]).
     */
```

```
        EnumerationElement ::=
                /Type:_r_Type/
                BindedName:Identifier
                [Value:Expression];

        /**
         * Incomplete structures and unions are represented by empty element
         * lists  ([6.2.5:22; p.36]).
         */
        abstract StructuredType:TagedType ::=
                Element:UnionOrStructureElement*;

            Structure:StructuredType;
            Union:StructuredType;

        abstract UnionOrStructureElement ::=
                Type:_r_Type;

            NormalUnionOrStructureElement:UnionOrStructureElement ::=
                    Name:Identifier;
            BitField:UnionOrStructureElement ::=
                    [Name:Identifier]
                    Size:Expression;

/* ********************** Statements [A.2.3;p.415] ********************** */

abstract Statement;

    EmptyStatement:Statement;
    ExpressionStatement:Statement ::=
            Expression;
    CompoundStatement:Statement ::=
            Statement*;

    abstract ConditionalControlFlowStatement:Statement ::=
            Condition:Expression
            Body:Statement;

        IfStatement:ConditionalControlFlowStatement ::=
                [Alternative:Statement];
        SwitchStatement:ConditionalControlFlowStatement;
        WhileStatement:ConditionalControlFlowStatement;
        DoWhileStatement:ConditionalControlFlowStatement;

        /**
         * REWRITE: Empty conditions are true ([6.8.5.3:2;p.136]). E.g.:
         *  for(expr1;;expr2)        --> for(expr1; 1 ;expr2)
         *  for(decl;;expr)          --> for(decl; 1 ;expr)
         */
        ForStatement:ConditionalControlFlowStatement ::=
                [ Initialization :Expression]
                [Iteration :Expression];
        ForStatement_Decl:ConditionalControlFlowStatement ::=
                [ Initialization : _r_DeclarationCollection]
                [Iteration :Expression];

    abstract JumpStatement:Statement;

        GotoJump:JumpStatement ::=
                Label: Identifier ;
        ContinueJump:JumpStatement;
        BreakJump:JumpStatement;
```

```
        ReturnJump:JumpStatement ::=
                [Result:Expression];

    abstract LabeledStatement:Statement ::=
            Continuation:Statement;

        ExplicitLabeledStatement:LabeledStatement ::=
                Label: Identifier ;

        abstract SwitchLabelStatement:LabeledStatement;

            CaseLabel:SwitchLabelStatement ::=
                    Condition:Expression;
            DefaultLabel:SwitchLabelStatement;

/* ********************** Expressions [A.2.1;p.409] ********************** */

abstract Expression;

    // Constant expressions are defined under the "Terminals" section.

    CommaExpression:Expression ::=
            Expression*;
    Variable:Expression ::=
            Name:Identifier;
    Cast:Expression ::=
            Type:_r_Type
            Operand:Expression;
    FunctionCall:Expression ::=
            FunctionObject:Expression
            Arguments:Expression*;
    CompoundLiteral:Expression ::=
            Type:_r_Type
            Values: Initializer ;

    /**
     * PARSER CONSTRAINT: Only the designations of initializer elements'
     *  designators are optional.
     */
    abstract DesignatorAccess:Expression ::=
            [Designation:Expression];

        /**
         * PARSER-CONSTRAINT / REWRITE: The structure pointer operator (->)
         *  is represented as an indirection operation and an associated
         *  member access. E.g.:
         *
         *  sp1->sp2->m          --> (*((*sp1).sp2)).m
         */
        MemberAccess:DesignatorAccess ::=
                Member:Identifier;
        ArrayAccess:DesignatorAccess ::=
                Designation2:Expression;

    abstract SizeOfExpression:Expression;

        SizeOfObject:SizeOfExpression ::=
                Object:Expression;
        SizeOfType:SizeOfExpression ::=
                Type:_r_Type;

    abstract AssignmentExpression:Expression ::=
```

```
401             lValue:Expression
402             rValue:Expression;
403
404         NormalAssignment:AssignmentExpression;
405         MultiplyAssignment:AssignmentExpression;
406         DivideAssignment:AssignmentExpression;
407         ModuloAssignment:AssignmentExpression;
408         AdditionAssignment:AssignmentExpression;
409         SubtractionAssignment:AssignmentExpression;
410         LeftShiftAssignment:AssignmentExpression;
411         RightShiftAssignment:AssignmentExpression;
412         AndAssignment:AssignmentExpression;
413         ExclusiveOrAssignment:AssignmentExpression;
414         InclusiveOrAssignment:AssignmentExpression;
415
416     abstract UnaryExpression:Expression ::=
417             Operand:Expression;
418
419         AdressOf:UnaryExpression;
420         Indirection : UnaryExpression;
421         PlusSign:UnaryExpression;
422         MinusSign:UnaryExpression;
423         OneComplement:UnaryExpression;
424         Negation:UnaryExpression;
425         IncrementBefore:UnaryExpression;
426         IncrementAfter:UnaryExpression;
427         DecrementBefore:UnaryExpression;
428         DecrementAfter:UnaryExpression;
429
430     abstract BinaryExpression:Expression ::=
431             Operand1:Expression
432             Operand2:Expression;
433
434         Multiplication : BinaryExpression;
435         Division : BinaryExpression;
436         Modulo:BinaryExpression;
437         Addition:BinaryExpression;
438         Subtraction:BinaryExpression;
439         LeftShift : BinaryExpression;
440         RightShift : BinaryExpression;
441         Lesser : BinaryExpression;
442         Greater:BinaryExpression;
443         LesserOrEqual:BinaryExpression;
444         GreaterOrEqual:BinaryExpression;
445         Equal:BinaryExpression;
446         NotEqual:BinaryExpression;
447         And:BinaryExpression;
448         ExclusiveOr:BinaryExpression;
449         InclusiveOr : BinaryExpression;
450         LogicalAnd:BinaryExpression;
451         LogicalOr:BinaryExpression;
452
453     abstract TernaryExpression:Expression ::=
454             Condition:Expression
455             Result1:Expression
456             Result2:Expression;
457
458         /**
459          * REWRITE: Result1's and Result2's type must be the same, thus it may
460          *  be necessary to promote one or both expressions
461          *  ([6.5.15:5,6; p. 90]). Such a promotion has to be represented by an
462          *  explicit type cast. E.g.: Assume a is of type integer and b is of
```

```
463         * type long
464         *
465         * a < b ? a : b            --> a < b ? (long)a : b;
466         */
467         ConditionalExpression:TernaryExpression;
```

Listing B.2: *Compiler-core*: *Intermediate AST* specification.

```
1  /**
2   * Intermediate AST specification (see chapter 5.3.2) for the JastAddC
3   * C99 compiler. For details about naming conventions for intermediate,
4   * mapping, relationship and final nodes see the final AST specification
5   * (C99AST.ast).
6   *
7   * Intermediate and mapping nodes are named after the standard's non-terminal
8   * ([A.2.1;p. 409]) they represent.
9   *
10  * @author C. Brger
11  */
12
13 /**
14  * MAPPING: Declarations have to be mapped to a collection containing a
15  *  declaration for each entity declared.
16  */
17 _m_declaration:_r_DeclarationCollection ::=
18         _i_declaration_specifiers*
19         _i_init_declarator*;
20
21 /**
22  * MAPPING: Function definitions have to be mapped to a
23  *  DeclarationCollection containing one FunctionDeclaration.
24  *
25  * PARSER-CONSTRAINT: The declarator must not be an abstract-declarator.
26  */
27 _m_function_definition:_r_DeclarationCollection ::=
28         _i_declaration_specifiers*
29         _i_declarator
30         <Body:CompoundStatement>;
31
32 _i_pointer ::=
33         <TypeQualifier:List<TypeQualifier>>
34         [_i_pointer];
35
36 /**
37  * PARSER-CONSTRAINT: The declarator can only be left for abstract-declarators.
38  */
39 _i_parameter_declaration ::=
40         _i_declaration_specifiers*
41         [_i_declarator];
42
43 _i_parameter_type_list ::=
44         _i_parameter_declaration*
45         <IsVariadicFunction:boolean>;
46
47 /**
48  * Abstract-declarators are modeled as declarators without a binded name.
49  */
50 abstract _i_declarator;
51
52     /**
53      * If a declarator's derivation contains the rule "direct-declarator ->
54      * identifier" it never represents an abstract-declarator.
```

```
     */
    _i_simple_declarator:_i_declarator ::=
            <Name:Identifier>;
    /**
     * PARSER-CONSTRAINT: Only for abstract-declarators the declarator may be left.
     */
    _i_pointer_declarator:_i_declarator ::=
            _i_pointer
            [_i_declarator];
    /**
     * PARSER-CONSTRAINT: Only for abstract-declarators the declarator may be left.
     */
    _i_array_declarator:_i_declarator ::=
            [_i_declarator]
            <TypeQualifier:List<TypeQualifier>>
            <Dimension:Opt<Expression>>;
    /**
     * PARSER-CONSTRAINT: Only for abstract-declarators the parameter-type-list
     *  and/or declarator may be left.
     */
    _i_function_declarator:_i_declarator ::=
            [_i_declarator]
            [_i_parameter_type_list];

/**
 * PARSER-CONSTRAINT: The declarator must not be an abstract-declarator.
 */
_i_init_declarator ::=
        _i_declarator
        < Initializer :Opt<Initializer>>;

abstract _i_declaration_specifiers;

    abstract _i_specifier_qualifier:_i_declaration_specifiers;

        abstract _i_type_specifier:_i_specifier_qualifier;

            /**
             * Atomic types are:
             *  - void
             *  - char
             *  - int
             *  - long
             *  - float
             *  - double
             *  - signed
             *  - unsigned
             *  - _Bool
             *  - _Complex
             *  - _Imaginary
             */
            _i_atomic_type:_i_type_specifier ::=
                    <TypeName:String>;
            /**
             * Incomplete structures or unions ([6.2.5:22; p.36]) have no
             * struct-declaration. Their struct-declaration-list is empty.
             */
            _i_struct_or_union_specifier:_i_type_specifier ::=
                    _i_struct_or_union
                    <Tag:Opt<Identifier>>
                    _i_struct_declaration*;
            _i_enum_specifier:_i_type_specifier ::=
```

```
                        <Tag:Opt<Identifier>>
                        <_enumerators:List<EnumerationElement>>;
                _i_typedef_name:_i_type_specifier ::=
                        <Name:Identifier>;

abstract _i_struct_or_union;

    _i_struct:_i_struct_or_union;
    _i_union:_i_struct_or_union;

_i_struct_declaration ::=
        _i_specifier_qualifier*
        _i_struct_declarator*;

/**
 * PARSER-CONSTRAINT: The declarator must not be an abstract-declarator.
 */
abstract _i_struct_declarator;

    _i_normal_struct_declarator:_i_struct_declarator ::=
            _i_declarator;
    _i_bitfield_struct_declarator:_i_struct_declarator ::=
            [_i_declarator]
            <BitFieldSize:Expression>;

/**
 * MAPPING: Type-name's have to be mapped to Type nodes.
 *
 * PARSER-CONSTRAINT: The declarator must be an abstract-declarator.
 */
_m_type_name:_r_Type ::=
        _i_specifier_qualifier*
        [_i_declarator];
```

B.2 *Core-Concerns*

Listing B.3 presents *JastAddC's core-concerns*. Each concern is represented by an aspect, specifying its API. For details about their tasks and implementations chapter 5 can be consulted.

Listing B.3: *Compiler-core*: *Core-concerns* specification.

```
import jastaddc.semantics.*;

/**
 * The following aspects specify JastAddC's core-concerns (see chapter 3.2.2).
 *
 * @author C. Brger
 */

/**
 * This aspect specifies the mapping-concern's tasks and API (see chapter 5.3).
 * In JastAddC, all declarations and type-names have to be mapped. For
 * expressions and statements the parser already can construct final ASTs.
 *
 * @author C. Brger
 */
aspect ConcreteToASTMapping {
    /**
     * Nodes representing declarations have to be mapped:
```

```
19      */
20      syn DeclarationCollection _r_DeclarationCollection.RetrieveFinalNode();
21      eq _r_DeclarationCollection.RetrieveFinalNode() {
22          throw new jastaddc.semantics.ConcreteToASTFailure(
23                  this, DeclarationCollection.class);
24      }
25      eq DeclarationCollection.RetrieveFinalNode() = this;
26
27      /**
28       * Nodes representing types have to be mapped:
29       */
30      syn Type _r_Type.RetrieveFinalNode();
31      eq _r_Type.RetrieveFinalNode() {
32          throw new jastaddc.semantics.ConcreteToASTFailure(
33                  this, Type.class);
34      }
35      eq Type.RetrieveFinalNode() = this;
36  }
37
38  /**
39   * This aspect specifies the access-support-concern's API (see chapter 5.2.2),
40   * which distributes context information across the AST, prepares information
41   * associated with AST nodes for further semantical analysis and realizes
42   * JastAddC's non-null-reference attributes.
43   *
44   * @author C. Brger
45   */
46  aspect AccessSupport {
47      /**
48       * Check if the declaration declares a function.
49       *
50       * CONSTRAINT: (IsFunctionDeclaration() == true) ==>
51       *       ((getType() instanceof FunctionType) == true)
52       */
53      syn boolean Declaration.IsFunctionDeclaration();
54      /**
55       * Check whether, the declaration tries to define a function or not.
56       * From the fact, that a function definition is given, cannot be
57       * concluded, that a valid function declaration is given. Thus:
58       * (IsFunctionDefinition() == true) =/=>
59       *       (IsFunctionDeclaration() == true)
60       *
61       * CONSTRAINT: A valid function definition is given, iff:
62       *  (IsFunctionDeclaration() && IsFunctionDefinition()) == true
63       */
64      syn boolean Declaration.IsFunctionDefinition();
65      /**
66       * Check if the declaration declares a function prototyp.
67       *
68       * CONSTRAINT: (IsFunctionPrototyp() == true) ==>
69       *       (IsFunctionDeclaration() == true)
70       * CONSTRAINT: (IsFunctionPrototyp() == true) ==>
71       *       (IsFunctionDefinition() == false)
72       */
73      syn boolean Declaration.IsFunctionPrototyp();
74
75      /**
76       * Check if a function has a parameter specification.
77       */
78      syn boolean FunctionType.IsWithoutParameterSpecification();
79      /**
80       * Check if a function's parameter specification defines the function
```

```
81      * to be parameterless.
82      */
83     syn boolean FunctionType.IsParameterlessFunction();
84
85     /**
86      * Retrieve information about specifiers and qualifiers:
87      */
88     syn boolean Declaration.IsSpecifiedBy(StorageClassSpecifier spec);
89     syn boolean Declaration.IsSpecifiedBy(FunctionSpecifier spec);
90     syn boolean Type.IsQualifiedBy(TypeQualifier qual);
91
92     /**
93      * Retrieve information about pointers:
94      */
95     syn boolean Pointer.IsFunctionPointer();
96     syn boolean Pointer.IsVoidPointer();
97
98     /**
99      * Distribute certain nodes important for context information:
100     */
101    inh Enumeration EnumerationElement.EnumerationContainingElement();
102    inh StructuredType
103        UnionOrStructureElement.StructuredTypeContainingMember();
104    // CONSTRAINT: Returns null, if not.
105    inh Declaration FunctionType.IsPartOfFunctionDeclaration();
106    // CONSTRAINT: Returns null, if not.
107    inh FunctionType Declaration.IsParameterDeclaration();
108    // CONSTRAINT: Returns null, if no enclosing switch statement is given.
109    inh SwitchStatement Statement.FindEnclosingSwitchStatement();
110    // CONSTRAINT: Return null, if no enclosing loop statement is given.
111    inh ConditionalControlFlowStatement
112        Statement.FindEnclosingLoopStatement();
113    // CONSTRAINT: Return null, if no enclosing function is given.
114    inh  FunctionDefinitionInitializer
115        Statement.FindEnclosingFunctionDefinition();
116
117    /**
118     * Support non-null reference attributes for the generic error handling
119     * concept (see chapter 5.1.2):
120     */
121    syn StructuredType MemberAccess.DesignationType();
122    syn Expression ArrayAccess.DesignationExpression();
123    syn Expression ArrayAccess.IndexExpression();
124    syn UnionOrStructureElement MemberAccess.MemberDeclaration();
125
126    syn IOrdinaryNameDeclaration Variable.VariableDeclaration();
127
128    syn Declaration TypedefTypeReference.ReferencedTypedefDeclaration();
129    syn Enumeration EnumTypeReference.ReferencedEnumerationType();
130    syn Structure StructureTypeReference.ReferencedStructureType();
131    syn Union UnionTypeReference.ReferencedUnionType();
132
133    syn LabeledStatement GotoJump.ReferencedStatement();
134    syn ConditionalControlFlowStatement ContinueJump.ReferencedLoop();
135    syn ConditionalControlFlowStatement
136        BreakJump.ReferencedConditionalControlFlowStatement();
137    syn  FunctionDefinitionInitializer
138        ReturnJump.ReferencedFunctionDefinition();
139
140    syn SwitchStatement SwitchLabelStatement.AssociatedSwitchStatement();
141
142    /**
```

```
143      * Associate with each switch statement a collection containing all its labels.
144      */
145     coll java.util.List<LabeledStatement> SwitchStatement.AssociatedLabels()
146         [new java.util.LinkedList<LabeledStatement>()] with add;
147
148     /**
149      * Associate with each function definition a collection containing all its
150      * return statements.
151      */
152     coll java.util.List<ReturnJump>
153         FunctionDefinitionInitializer.AssociatedReturnJumps()
154         [new java.util.LinkedList<ReturnJump>()] with add;
155
156     /**
157      * Retrieve an AST node's unique Dewey address.
158      * Addresses are comparable, so it can be decided whether:
159      *  - A node is an ancestor of another node
160      *  - A node is a successor of another node
161      *  - A node is in a subtree left to another node
162      *  - A node is in a subtree right to another node
163      */
164     inh jastaddc.semantics.NodeAddress ASTNode.NodeAddress();
165
166     /**
167      * Provide names for special AST nodes, even if the node has no explicit
168      * name in the source code:
169      */
170     syn String Identifier.Name();
171     syn String EnumerationElement.Name();
172     syn String TagedType.Name();
173     syn String MemberAccess.MemberName();
174     syn String UnionOrStructureElement.Name();
175     syn String Variable.Name();
176     syn String Declaration.Name();
177     syn String LabeledStatement.Name();
178     syn String GotoJump.DestinationName();
179     syn String AtomicType.Name();
180
181     /**
182      * Print human readable type descriptions.
183      */
184     syn String Type.ReadableTypeName();
185
186     /**
187      * Associate with each node a source and a line and column position:
188      */
189     public Integer ASTNode.line = null;
190     public Integer ASTNode.column = null;
191     public String ASTNode.sourceName = null;
192     public int ASTNode.line() {
193         if (line != null)
194             return line;
195         if (getParent() != null)
196             return getParent().line();
197         return 0;
198     }
199     public int ASTNode.column() {
200         if (column != null)
201             return column;
202         if (getParent() != null)
203             return getParent().column();
204         return 0;
```

```
205     }
206     public String ASTNode.sourceName() {
207         if (sourceName != null)
208             return sourceName;
209         if (getParent() != null)
210             return getParent().sourceName();
211         return "UNKNOWN_SOURCE";
212     }
213     public String ASTNode.sourceCodeLocation() {
214         return "(" + sourceName() +":"+ line() +";"+ column() +")";
215     }
216 }
217
218 /**
219  * This aspect declares the implicit information retrieval concern, which
220  * has to support rewrites to add AST nodes/semantics given implicitly in
221  * the C99 standard (see chapter 5.1.3).
222  *
223  * @author C. Brger
224  */
225 aspect ImplicitInformationRetrieval {
226     /**
227      * Default tree parts represented by non-terminal attributes:
228      */
229     // Enumeration members have type int ([6.7.2.2:3;p. 105]).
230     syn lazy _r_Type EnumerationElement.getType();
231
232     /**
233      * REWRITES:
234      * - If no value for an enumeration constant is given, its value is 0,
235      *   if it is the enumeration's first constant. Otherwise its value is
236      *   the previous enumeration constant's one plus 1 ([6.7.2.2:3; p. 105]).
237      */
238 }
239
240 /**
241  * The name analysis module provides functionalities for
242  * - scoping rules ( visibility of declarations)
243  * - extent (lifetime of variables and functions)
244  * - overloading classes (name spaces)
245  *
246  * C99 distinguishes 5 name spaces (see "C: A Reference Manual", 5th edition,
247  * 4.2.4, Table 4-2, p. 78):
248  * - preprocessor macro names
249  * - statement labels
250  * - structure, union and enumeration tags
251  * - component names (members)
252  * - ordinary names (variables, functions, typedef names,
253  *   enumeration constants)
254  * The name analysis provides for each name space methods to look up entities.
255  *
256  * @author C. Brger
257  */
258 aspect NameAnalysis {
259     /**
260      * Search for an ordinary name in scope:
261      */
262     syn boolean ASTNode.IntroducesNewOrdinaryNameBlock();
263     syn ASTNode ASTNode.FindNodeIntroducingThisOrdinaryNameBlock();
264     syn java.util.List<IOrdinaryNameDeclaration>
265         ASTNode.LookUpOrdinaryName(String name);
266     syn java.util.List<IOrdinaryNameDeclaration> ASTNode.LookUpOrdinaryName(
```

```
267                 String name, jastaddc.semantics.NodeAddress upToAddress);
268
269         Declaration implements IOrdinaryNameDeclaration;
270         EnumerationElement implements IOrdinaryNameDeclaration;
271
272         /**
273          * Search for a label in scope, which is no switch label:
274          */
275         syn boolean ASTNode.IntroducesNewLabelBlock();
276         syn ASTNode ASTNode.FindNodeIntroducingThisLabelBlock();
277         syn java.util.List<LabeledStatement> ASTNode.LookUpLabel(String name);
278
279         /**
280          * Search for a member with a given name in a structured type.
281          */
282         syn UnionOrStructureElement
283             StructuredType.LookUpMember(String memberToFind);
284
285         /**
286          * Search for a structure, union or enumeration tag in scope:
287          */
288         syn boolean ASTNode.IntroducesNewTagBlock();
289         syn ASTNode ASTNode.FindNodeIntroducingThisTagBlock();
290         syn java.util.List<TagedType> ASTNode.LookUpTag(String name);
291         syn java.util.List<TagedType> ASTNode.LookUpTag(
292                 String name, jastaddc.semantics.NodeAddress upToAddress);
293
294         /**
295          * Retrieve a structured type's definition.
296          *
297          * CONSTRAINT: Assume a structured type T. Assume T is incomplete.
298          *  Iff T has a definition, T.FindCompleteTagedType() returns it.
299          *  Otherwise it returns null. Iff T is not incomplete, it never
300          *  returns null. T.FindCompleteTagedType()'s value always is of
301          *  type T or null.
302          */
303         syn TagedType TagedType.FindCompleteTagedType();
304
305         /**
306          * Decide, whether a structured type is incomplete or not.
307          */
308         syn boolean TagedType.IsIncompleteType();
309
310         /**
311          * This interface is used to represent results of ordinary name look ups.
312          *
313          * CONSTRAINT: Each node type, which can represent an ordinary name
314          *  declaration must implement this interface.
315          *
316          * @author C. Brger
317          */
318         public interface IOrdinaryNameDeclaration {
319             boolean IsErrorNode();
320             boolean hasBindedName();
321             Identifier getBindedName();
322             String Name();
323             _r_Type getType();
324         }
325         public boolean EnumerationElement.hasBindedName() {return true;}
326     }
327
328     aspect TypeAnalysis {
```

```
329     // TODO
330 }
331
332 /**
333  * This aspect specifies the error handling concern's tasks and API
334  * (see chapter 5.2.1).
335  *
336  * Two main tasks have to be realized:
337  *  - The construction of error nodes
338  *  - The association of errors with AST nodes
339  *
340  * @author C. Brger
341  */
342 aspect ErrorHandling {
343     /**
344      * Support access to the error node builder:
345      */
346     syn Compilation ASTNode.CompilationRootNode() =
347         getParent().CompilationRootNode();
348     eq Compilation.CompilationRootNode() = this;
349     syn ErrorBuilder ASTNode.ErrorBuilder() =
350         CompilationRootNode().getErrorBuilder();
351
352     /**
353      * Check, whether an AST node is an error node or not.
354      */
355     inh boolean ASTNode.IsErrorNode();
356
357     /**
358      * Retrieve an error node of the given type.
359      *
360      * CONSTRAINT: Nodes returned must never become part of an AST.
361      */
362     syn lazy ASTNode ErrorBuilder.ConstructErrorNode(Class nodeType);
363
364     /**
365      * Add an error to the node's error list.
366      */
367     syn boolean ASTNode.addError(jastaddc.SourceError error);
368
369     /**
370      * Return an iterator above all the tree's errors.
371      */
372     syn java.util.Iterator<jastaddc.SourceError>
373         ASTNode.collectAllErrorsOfTree();
374
375     /**
376      * Return the node's errors.
377      */
378     syn java.util.Collection<jastaddc.SourceError>
379         ASTNode.collectAllErrorsOfNode();
380
381     /**
382      * Check the tree for the node, with which the given error is associated.
383      *
384      * CONSTRAINT: Iff the node cannot be found, null is returned.
385      */
386     syn ASTNode ASTNode.findNodeInSubtreeErrorIsAssociatedWith(
387             jastaddc.SourceError error);
388 }
389
390 /**
```

```
391  * This aspect specifies the constraint verification concern's tasks and API
392  * (see chapter 5.2.3). Its main task is to decide for any C program or
393  * fragment, whether it satisfies all semantic constraints or not. Thus, it
394  * must support methods to do a semantic evaluation of the complete AST,
395  * for a subtree or even for a single AST node.
396  *
397  * @author C. Brger
398  */
399 aspect ConstraintVerification {
400     /**
401      * Check all semantic constraints of the node and its subtree.
402      */
403     syn boolean ASTNode.IsCorrect();
404
405     /**
406      * Check all the node's semantic constraints, without
407      * checking its subtree.
408      */
409     syn boolean ASTNode.IsCorrectLocal();
410 }
411
412 /**
413  * This aspect specifies support methods, classes and interfaces.
414  *
415  * @author C. Brger
416  */
417 aspect Support {
418     /**
419      * Support the construction of references to taged types:
420      */
421     syn Type TagedType.buildQualifiedReference();
422     eq Enumeration.buildQualifiedReference() =
423         new EnumTypeReference(buildTypeQualifierCopy(),
424                 new Identifier(Name()));
425     eq Structure.buildQualifiedReference() =
426         new StructureTypeReference(buildTypeQualifierCopy(),
427                 new Identifier(Name()));
428     eq Union.buildQualifiedReference() =
429         new UnionTypeReference(buildTypeQualifierCopy(),
430                 new Identifier(Name()));
431
432     syn List<TypeQualifier> TagedType.buildTypeQualifierCopy() {
433         List<TypeQualifier> result = new List<TypeQualifier>();
434         for (int i = 0; i < getNumTypeQualifier(); i++)
435             result.add((TypeQualifier)getTypeQualifier(i).copy());
436         return result;
437     }
438 }
```

Bibliography

[1] Monica S. Lam Alfred V. Aho and Ravi Sethi. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2 edition, October 2007. [cited on page 2, 11]

[2] Alexander Asteroth and Christel Baier. *Theoretische Informatik*. Pearson Studium, 1 edition, 2002. [cited on page 3, 8, 11, 39]

[3] Uwe Aßmann. *Invasive Software Composition*. Springer, 1 edition, 2003. [cited on page 31]

[4] Helmut Balzert. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, 1 edition, 1998. [cited on page 34, 35]

[5] M.G.J. van den Brand, S. Klusener, L. Moonen, and J.J. Vinju. Generalized Parsing and Term Rewriting - Semantics Directed Disambiguation. In Barret Bryant and João Saraiva, editors, *Third Workshop on Language Descriptions Tools and Applications*, volume 82 of *Electronic Notes in Theoretical Computer Science*, pages 575–591. Elsevier, December 2003. [cited on page 106]

[6] Alexander Demenchuk. Beaver project. web page, May 2006. http://beaver.sourceforge.net/, Version 0.9.6.1. [cited on page 13]

[7] Peter Dencker. *Generative attribuierte Grammatiken*. PhD thesis, Universität Karlsruhe, Karlsruhe, Germany, 1986. [cited on page 12, 93]

[8] Melvil Dewey. *A Classification and Subject Index for Cataloguing and Arranging the Books and Pamphlets of a Library*. Amherst, U. S. A., 1876. [cited on page 39, 73]

[9] Charles Donnelly and Richard Stallman. *Bison: The Yacc-compatible Parser Generator*, 2.3 edition, March 2006. [cited on page 10, 13, 14]

[10] Torbjörn Ekman and Görel Hedin. The jastadd system — modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, December 2007. [cited on page 12]

[11] Torbjörn Ekman. *Extensible Compiler Construction*. Tryckeriet i E-huset, Lund University, 1 edition, 2006. [cited on page 3, 11, 12, 121]

[12] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns*. Addison-Wesley, 1 edition, 1995. [cited on page 13]

[13] James Gosling, Bill Joy, and Guy L. Steele Jr. *The Java Language Specification*. Java. Addison-Wesley, 3 edition, Juni 2005. [cited on page 2, 12]

[14] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, February 1992. [cited on page 10]

[15] Apache Group. Apache ant project. web page, December 2006. http://ant.apache.org, Version 1.7.0. [cited on page 18, 19, 104]

[16] David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1 edition, February 1995. [cited on page 120]

[17] Görel Hedin. An object-oriented notation for attribute grammars. In S. Cook, editor, *Proceedings of the ECOOP '89 European Conference on Object-oriented Programming*, pages 329–345, Nottingham, July 1989. Cambridge University Press. [cited on page 14]

[18] Görel Hedin, Anders Nilsson, Anders Ive, and Torbjörn Ekman. Implementing java compilers using recrags. *Nordic Journal of Computing*, 2004. [cited on page 115, 121]

[19] Cay S. Horstmann and Gary Cornell. *Core Java: Fundamentals*, volume 1 of *Core*. Prentice Hall, 8 edition, September 2007. [cited on page 2]

[20] Cay S. Horstmann and Gary Cornell. *Core Java: Advanced Features*, volume 2 of *Core*. Prentice Hall, 8 edition, Mai 2008. [cited on page 2]

[21] Samuel P. Harbison III and Guy L. Steele Jr. *C: A Reference Manual*. Prentice Hall, 5 edition, 2002. [cited on page 3, 4]

[22] INRIA. Ocaml. web page. http://caml.inria.fr/. [cited on page 121]

[23] British Standards Institute. *The C Standard*. Wiley & Sons, 1 edition, September 2003. ISO/IEC 9899: 1999. [cited on page 4, 5, 41, 61, 95, 123]

[24] S.C. Johnson. Yacc: Yet another compiler-compiler. Computing Science 32, AT&T Bell Laboratories, Murray Hill, New Jersey, USA, 1975. [cited on page 10, 13, 14]

[25] Uwe Kastens. Attribute grammars in a compiler construction environment. In *Proceedings on Attribute Grammars, Applications and Systems*, pages 380–400, London, UK, 1991. Springer. [cited on page 10]

[26] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 2 edition, April 1988. [cited on page 4]

[27] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97: Proceedings of the 11'th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, Berlin, Heidelberg, and New York, June 1997. [cited on page 11]

[28] Gerwin Klein. Jflex project. web page, May 2008. http://jflex.de/, Version 1.4.2. [cited on page 13]

[29] Gerwin Klein. *JFlex User's Manual*, 1.4.2 edition, May 2008. [cited on page 13]

[30] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano. [cited on page 106, 108]

[31] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968. [cited on page 3]

[32] Donald E. Knuth. Semantics of context-free languages: Correction. *Theory of Computing Systems*, 5(2):95–96, June 1971. [cited on page 3]

[33] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley, 1 edition, Januar 1989. [cited on page 4]

[34] M. E. Lesk and E. Schmidt. Lex—a lexical analyzer generator. Computing Science 39, AT&T Bell Laboratories, Murray Hill, New Jersey, USA, 1975. [cited on page 10]

[35] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Java. Addison-Wesley, 2 edition, April 1999. [cited on page 2, 12, 98]

[36] Eva Magnusson. *Object-Oriented Declarative Program Analysis*. Tryckeriet i E-huset, Lund University, 1 edition, 2007. [cited on page 3, 11, 12]

[37] Scott McPeak and George C. Necula. Elkhound: A fast, practical glr parser generator. In E. Duesterwald, editor, *CC '04: Proceedings of the 13th International Conference on Compiler Construction*, volume 2985/2004 of *Lecture Notes in Computer Science*, pages 73–88, Berlin / Heidelberg, Germany, March 2004. Springer. [cited on page 3, 99]

[38] George C. Necula, Scott Mcpeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, volume 2304/2002, pages 213–228, Berlin / Heidelberg, Germany, April 2002. Springer. [cited on page 121]

[39] Tobias Nipkow and Franz Baader. *Term Rewriting and All That*. Springer, 2 edition, October 1999. [cited on page 93]

[40] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 3 edition, September 2004. [cited on page 115, 121]

[41] Vern Paxon. *Flex, A fast scanner generator*, 2.5 edition, March 1995. [cited on page 10, 13, 16]

[42] Lothar Schmitz. *Syntaxbasierte Programmierwerkzeuge*. Leitfäden der Informatik. B. G. Teubner, 1 edition, 1995. [cited on page 3, 11]

[43] S. Sippu and E. Soisalon-Soininen. *Parsing Theory - Vol.I: Languages and Parsing*, volume I of *EATCS Monographs on Theoretical Computer Science*. Springer, 1 edition, 1988. [cited on page 3, 8]

[44] S. Sippu and E. Soisalon-Soininen. *Parsing Theory - Vol.II: LR(k) and LL(k) Parsing*, volume II of *EATCS Monographs on Theoretical Computer Science*. Springer, 1 edition, 1990. [cited on page 3, 8, 11]

[45] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley, 2 edition, November 2002. [cited on page 31]

[46] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized lr parsers. In R. Nigel Horspool, editor, *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, volume 2304/2002 of *Lecture Notes in Computer Science*, page 143–158, Berlin / Heidelberg, Germany, April 2002. Springer. [cited on page 103, 109]

[47] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. [cited on page 3, 99]

[48] Harald H. Vogt, Doaitse Swierstra, and Matthijs F. Kuiper. Higher order attribute grammars. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 131–145, New York, NY, USA, June 1989. ACM. [cited on page 12, 93]

[49] Reinhard Wilhelm and Helmut Seidl. *Übersetzerbau: Virtuelle Maschinen*. eXamen.press. Springer, 1 edition, 2007. [cited on page 115]

[50] Eric R. Van Wyk and August C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 63–72, New York, NY, USA, October 2007. ACM. [cited on page 100]

[51] Johan Åkesson. *Languages and Tools for Optimization of Large-Scale Systems*. Media-Tryck, 1 edition, November 2007. [cited on page 114]

List of Figures

2.1 Classical organization for the generation of compiler parts for different compilation phases. 10
2.2 Organization of *JastAdd* projects and the interaction between specifications, the *JastAdd* generator, generated semantic evaluator and parser. 13
2.3 Organization of *Beaver* projects and the interaction between specification, *Beaver* generator and generated parser. 15
2.4 Organization of *JFlex* projects and the interaction between specification, *JFlex* generator and generated lexer. 16
2.5 Complete compiler project structure for *JastAdd*, *Beaver* and *JFlex* and the interaction between specifications, generators and generated lexer, parser and semantic evaluator. 18
2.6 Calculator compiler: AST class hierarchy generated for the claculator's `*.ast` specification given in listing 2.4. 23

3.1 Modularization in phases and *passes*. 32
3.2 *JastAddC's* modularization: The *compiler-core* module, the modules implementing declared *core-concerns* and extension modules. 37

4.1 *JastAddC's lexer stack*. 48

5.1 *Error handling* module: `DefaultErrorBuilder`, *error node AST* and *non-null reference attributes* pointing to *error nodes*. 70
5.2 The standard's syntax tree for a complicated *C* declaration (part I). 80
5.3 The standard's syntax tree for a complicated *C* declaration (part II). 81
5.4 *Concrete to AST mapping* module: The AST *JastAddC's Beaver* parser constructs for a complicated *C* declaration (part I). 82
5.5 *Concrete to AST mapping* module: The AST *JastAddC's Beaver* parser constructs for a complicated *C* declaration (part II). . . . 83
5.6 *Concrete to AST mapping* module: The AST *JastAddC's* semantical computations expect for a complicated *C* declaration. . 84
5.7 Mapping technique: Imaginary example for the mapping of a *final AST* to its corresponding *complete final AST* in two steps. . . . 86

7.1 Complete compiler modularization across concerns with SGLR and ReCRAGs. 104

7.2 Breaking cyclic dependencies between *compilation phases*. 106
7.3 Imaginary AST with *ambiguous nodes*. 107
7.4 Automatically correct interaction between *ReCRAG AST mappings* and the disambiguation technique. 111
7.5 *JastAddC* AST with associated generated code, the abstract machine AST. 115

8.1 *Specification-Driven* Development of Extensible Compilers: Topics handled throughout the thesis and the chapters presenting them. 119

List of Tables

3.1 *JastAddC's compilation-phase – concern matrix* 35

5.1 Overview of *JastAddC's* concepts for semantical computations, internal error representation/handling and constraint checking/-construction of error messages. 62
5.2 Overview of *JastAddC's compiler-core* concerns and their tasks. . 65

6.1 *C* parts not implemented in *JastAddC*. 97

7.1 Comparing the mapping technique of chapter 5.3 and the disambiguation technique of chapter 7.2.1. 112

Listings

2.1 Generic *glue-code main class* skeleton, which composes a final compiler based on a *JFlex* lexer, *Beaver* parser and *JastAdd* semantic evaluator. 18
2.2 Calculator compiler: Language syntax. 20
2.3 Calculator compiler: *Main class*. 21
2.4 Calculator compiler: `*.ast` specification. 22
2.5 Calculator compiler: *Java* code constructing an AST for the expression $1 + + + a/(7 - b)$. 23
2.6 Calculator compiler: `*.jrag` specification for the evaluation of expressions. 24
2.7 Calculator compiler: `*.jrag` specification for the name analysis. 24
2.8 Calculator compiler: *Beaver* specification. 25
2.9 Calculator compiler: *JFlex* specification. 27
3.1 *JastAddC*'s *compiler-core*: Example fragment of the *core-AST* (`C99.ast`) specification. 37
3.2 *JastAddC*'s *compiler-core*: Example fragment for the declaration of *core-concerns* (`CompilerAPI.jrag`). 38
4.1 Specification fragment of *JastAddC's core-AST*: AST nodes representing identifiers, constants and string literals. 42
4.2 Specification fragment of *JastAddC's* token recognition and construction *pass* (`CLexer`). 43
4.3 `CLexicalAnalyser` code fragment: Composition of the *lexer stack*. 48
4.4 `CCharacterMapper_` specification fragment: *Lexical rules* to map escape sequences in character constants and string literals. 50
4.5 `CParser` specification fragment: CFG rules for `primary_expression` (6.5.1), `argument_expression_list` (6.5.2) and `selection_statement` (6.8.4). 51
4.6 *Typedef problem* introduction example. 55
4.7 *Typedef problem* example: Identifier representing a type and a variable within the same declaration. 56
4.8 *Typedef problem* example: Scoping and shadowing of typedef-names. 57
4.9 *Typedef problem* example: Proof for the ambiguity of *C's* syntax. 58
5.1 *Error handling* module: *Compiler-core* declarations for the association of *error objects* with AST nodes. 67
5.2 *Error handling* module: Implementation realizing the association of *error objects* with AST nodes. 68
5.3 *Error handling* module: *Compiler-core* declarations for the recognition and supply of error nodes. 69

5.4 *Error handling* module: Implementation realizing the recognition and supply of error nodes. 70
5.5 *Error handling* module: Initialization of the `DefaultErrorBuilder's` *error node AST*. 72
5.6 *Access support* module: Implementation for the support of unique, comparable node addresses. 73
5.7 *Access support* module: Implementation handling the access of taged type names. 74
5.8 *Access support* module: *Compiler-core* declarations for the distribution of the switch statement currently in scope, the association of each switch label to a switch statement and the collection of a switch statement's switch labels. 74
5.9 *Access support* module: Implementation of the *access support* concern's switch statement functionalities. 75
5.10 *Constraint verification* module: Implementation of the `IsCorrect()` attribute. 76
5.11 *Constraint verification* module: Implementation of the `SwitchLabelStatement` node type's `IsCorrectLocal()` attribute. . . . 76
7.1 *C* example for SQL expressions and the `foreach` keyword. 100
7.2 SGLR specification for the `enum` specifier. Program input characters are embedded between `"`. 101
7.3 Simplified SGLR specification for *identifier*. 102
7.4 SGLR specification for the `foreach` extension module. 102
7.5 SGLR specification for the SQL extension module. 103
7.6 *C* code skeleton for a *compound statement* containing a *for statement*. 114
B.1 *Compiler-core*: *Complete final AST* specification. 141
B.2 *Compiler-core*: *Intermediate AST* specification. 149
B.3 *Compiler-core*: *Core-concerns* specification. 151

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, 25th September 2008