

Paradigms for using semantic actions and code execution with JavaCC

Marco Maniscalco
UNIVERSITY OF APPLIED SCIENCES STUTTGART

December 17, 2006

Abstract

This paper introduces compiler construction basics with JavaCC and focuses on paradigms of language translation with abstract syntax trees. In this context one-pass compilers, handwritten syntax trees as well as generated tree parses have been discussed.

Contents

1	Introduction	1
2	Compiler construction	3
2.1	Overview	3
2.2	Practical use of compilers	3
2.3	JavaCC and other modern generators	4
2.4	Current development	5
3	Grammars	6
3.1	BNF/EBNF (Backus Naur Form)	7
3.2	Choice positions	7
3.3	Lookahead	8
3.4	Conflicts	9
3.5	Choice determination algorithm	9
3.6	Syntax lookahead	10
3.7	Semantic lookahead	10
4	Semantics	11
4.1	Embedding semantic actions	11
4.2	One-pass compiler	12

5	Abstract syntax trees	12
5.1	Semantic classes	12
5.2	Paradigms of code execution	13
5.3	Manual syntax tree construction	13
5.4	Visitor pattern	13
5.5	Tree parser	14
5.6	JJTree and JTB	14
6	Conclusion	15

1 Introduction

JavaCC (Java compiler compiler¹) is a parser generator and lexical analyzer generator that generates parsers from grammars. Complex problems can be solved using generated parsers. Common tasks are interpreters, compilers as well as pretty printers and recognizers for dynamic and complex data structures. This paper first introduces some compiler construction basics leading to a comparison of common paradigms of code execution followed by promising visions in compiler construction for future use.

Plain language recognition is a more trivial task than language translation when it comes to semantic actions. However compiler construction deals with translation of a source language to a target language what als can be termed *code execution*.

There are three common techniques that actually execute code aided by recognizers and semantic actions.

- one-pass compilers
- manual syntax tree construction
- tree parsers and generated parse trees

These paradigms will be introduced and a comparison from a practical point of view will be given. First various compiler construction basics are refreshed like the role of lexical analyzers and what slicing a given word into tokens is used for. A parser parser uses these tokens to recognize the word that matches the BNF (Backus Naur Form) productions of a given grammar. Such grammars consists of regular expressions and of BNF productions, which may contain augmented semantic actions.

The output of JavaCC is a top-down LL(k) parser that consists of Java code. Furthermore it is possible to auto-create abstract syntax trees [1] with JJTree or JTB (Java Tree Builder).

The third chapter deals with grammar syntax and lookahead technique. The fourth chapter shows how EBNF productions can be augmented with semantic actions followed by the fifth chapter where examples of abstract syntax trees

¹<https://javacc.dev.java.net/>

and different paradigms of translation will be treated. The last chapter gives a perspective of modern compiler construction and its meaning today.

2 Compiler construction

2.1 Overview

Translators take an input language and apply transformation rules in order to generate corresponding output in another language. A lexer determines the tokens which represent the *vocabulary* of the language. These tokens are used in the *symbol table*. The parser picks these tokens in the read direction and validates against grammar rules. If a rule is violated, this is called a syntax error. As a result, the parser constructs a tree structure of tokens which in turn gets used by the semantic code. The following figure shows the principle.

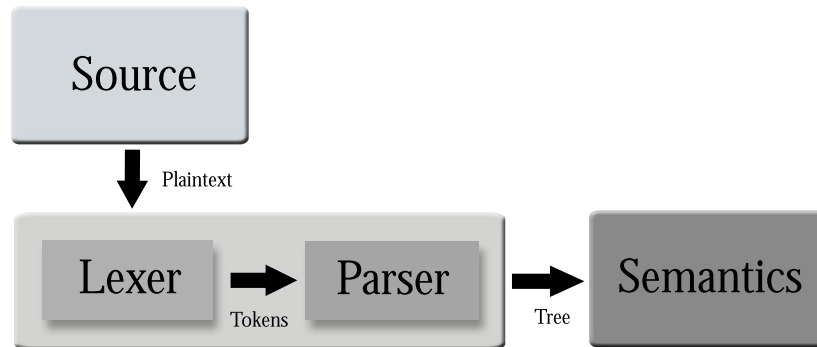


Figure 1: Principle of a translator

2.2 Practical use of compilers

Nowadays, there is a broad spectrum of use for compilers that focus on high-level languages as well as the recognition of complex data structures. As the development of Java6 (Mustang) demonstrates, the area in which highly defined languages already exist, is moving in the direction of scripting. The dynamic extension of languages through flexible features is achieved with scripting languages. Regular expressions is a typical usecase within Java. Ruby on Rails, Groovy, Python and other languages can be embedded as well.

Another area deals with the underlying data structure of highlevel languages in modern IDEs. (Integrated Development Environments) Syntax highlighting and powerful refactoring methods as used in Eclipse, would not be possible without the obligatory parser. In order to use modern refactoring methods accessing sourcecode in text form is not sufficient. It is necessary to use context and information regardless of what language syntax is applied.

For this reason the kind of abstract syntax tree has to be applied, who can provide the necessary object model. The mentioned refactoring-example needs parsers as a vital part in order to work reliable and scalable. In order to gain a full roundtrip engineering it is necessary to have the textual representation of

the source code in form of a syntax tree. This syntax tree has to be transformed into the textual version. One obvious advantage of syntax trees is the fact that operations, like restructuring, compression, optimization and other transformations can be implemented more easily. Furthermore, these operations have proven to be more reliable than those implemented with plain text format. The recognition of complex data structures requires complex state machines, that need a lot of effort for being developed in the first place, and furthermore need a lot of maintenance to be adjusted in order to meet different requirements.

Another example is *date string recognition* what can be done with different approaches. Goal is to gain atomic access to a string like "29.Feb.1981" and validate against rules. The implementation of a deterministic state machine for recognizing this string can grow to a size that is not manageable without a great amount of effort. Intention of this paper is to show an sophisticated way to implement universal recognizers and translators. Since such a recognizer can be generated by the grammar of the language, reduced complexity has to be expected because it is not necessary to touch generated code.

All recognizers can be created by hand as well. For a more sophisticated solution, generated parsers can be used without the amount of effort put into the handcrafted parsers. As a result, more time and effort can be allocated to the actual problem solving activity. As a general rule handcrafted parsers seem to be more performant at functioning and easier to debug. One disadvantage is the fact that extension is a more time consuming factor than with generated parsers. However, this criterion hardly counts for modern computers, which are in general equipped with powerful computing capability.

2.3 JavaCC and other modern generators

Asides from JavaCC, there are other modern generators like ANTLR, developed by Terence Parr, or Jikespg created by IBM. Alternative generators like Lex [2] and Yacc [3] or Flex [4] and Bison [5] are 15 years old and some parts of them are even older. JavaCC and ANTLR are LL(k) parser generators whereas Yacc is a LALR(1) generator. In this paper all statements are related to LL(k) parsers.

JavaCC creates a top-down parser (recursive descending) [6] unlike to tools like Yacc, they produce a bottom-up parser what is also not subjected here. The use of top-down parses allows more natural grammars whereat more natural means better readability for humans. Also left recursion is not allowed. As showed in the book compiler construction [7] bottom-up parsers can provide more powerful language recognition than top-down parsers. Top-down parsers are also easier to debug and passing attributes into the recursive descent parser is more easy. As shown in the next chapters, providing attributes forward and backward in the parsing tree is an essential feature when it comes to semantics.

In JavaCC the lexical specification like regular expressions and strings as well as the grammar productions are written down in the same file. This keeps things simple because while developing a compiler, the grammar and lexical specification often gets modified simultaneously. Also an important feature of

JavaCC is the use of EBNF (Extended Backus Naur Form) which enriches the BNF (Backus Naur Form) with regular expressions.

JavaCC produces LL(1) parsers by default but is not limited to $k=1$ so $k>1$ is possible at any choice position [1]. This functionality is achieved with syntactic and semantic lookahead, what is described in the 3rd chapter. The "shift-shift" ambiguity can be solved locally to gain a lookahead of $k>1$. A JavaCC-generated parser can thus be LL(k) at the points where it is needed. "Shift-reduce" and "reduce-reduce" are not subject to top-down parsers and therefore not handled here.

2.4 Current development

There is table-driven parsing and recursive descent parsing (like JavaCC uses). A recursive descent parser implements every production as a procedure. The call stack holds active procedures corresponding to pending non-terminals. With a table-driven parser the recursion is simulated with an explicit stack. Usually this can be solved using a finite state machine with a stack, namely a PDA (push down automata).

There are differing algorithms that get used when calculating lookahead information. A full strength LL(k) analysis of a grammar is known as a NP-complete problem. Many parser generators use either a table-driven parse, which is usually 1/3 in codesize of a recursive descent parser, which also limits this parser to a small k value. Many deterministic algorithms must either limit k to a very small value, usually two, or use approximate lookahead methods that are considerably weaker than the strong LL(k) method. Terence Parr [6] invented the approximate lookahead method. In theory, storing full lookahead information for one decision requires $O(|T|^k)$ space, where $|T|$ is the number of token types. In practice, it is expensive for a parser generator to compute full lookahead for $k>1$ and the resulting parsers are usually impractically large. Storing linear-approximate decisions requires $O(|T| \cdot k)$ space, a significant saving over the $O(|T|^k)$ required for full lookahead. Another technique is to limit generation to LL(1) and offer the author the possibility to self-decide where lookahead $k>1$ is needed to guide the generator.

Terence Parr [8] also showed that a lookahead with $k>1$ can be useful in real-world applications but also stated the need for efficient algorithms to prevent computing strong LL(k).

JavaCC follows the paradigm that the author of the grammar has to decide on which conflicting points a lookahead has to be made. Also the size or the semantic of a lookahead can be given by the author. This gives good performance for the LL(1) case and full lookahead at any other choice position. Besides lookahead technique JavaCC provides error-recovering, tree construction and augmented grammars.

3 Grammars

Grammar files are not separated by lexical specification and grammar productions in JavaCC. Both parts take place in the same file that makes the system arranged more clearly and compact [1]. Typically a file starts with the lexical specifications where all tokens and blanks are defined. Blanks are a special kind of terminal symbols such as whitespaces or carriage returns, which are ignored while lexical analyzation. These tokens can be defined using regular expressions, (`["0"- "9"]+` e.g. matches all numeric strings with lenght $n > 1$. In order to match integers of variable length the following statement shows one implementation.

Listing 1: Integer lexer

```
1 TOKEN : { < #DIGIT : [ "0"- "9" ] > }
2 TOKEN : { < INTEGER : (<DIGIT>+ ( " i" | " I" )? ) > }
3 TOKEN : { < MULT: " *" > }
```

This is enough to recognize a language and to internally construct the token stream. After this part the grammar productions are given. Since JavaCC generates a LL(k) recursive descent parser, all grammar productions are entered as recursive functions. A possibility for ignoring blanks is the following:

```
1 SKIP : { " \n" | " \r" | " \r\n" }.
```

JavaCC also offers some special predefined tokens like `<EOF>` (end of file). This token represents the end of the input stream and should be consumed at the end of the input. For example the input `"24 * 99\n"` will be recognized as `"INTEGER MULT INTEGER EOF"`. The ordering of the tokens within the lexical specification is vital to the functionality of the parser. The lexical analyzer generator part of JavaCC reads the lexical definitions top down and tries to match tokens in that order. Various side-effects are possible when ambiguous specifications are given. This can happen when a token was consumed inspite of not being meant to be consumed. An example would be the *"identifier-problem"* that occurs when the ordering of the lexical productions is incorrect.

```
1 TOKEN : { < FUNCTION : "function" > }
2 TOKEN : { < IDENTIFIER : <CHAR>(<DIGIT>|<CHAR>)* }
3 TOKEN : { < #DIGIT : [ "0"- "9" ] > }
4 TOKEN : { < #CHAR : [ "a"- "z" , "A"- "Z" , " _" ] > }
```

A correct way is to match *FUNCTION* before trying to match *IDENTIFIER*. If *IDENTIFIER* is matched prior to *IDENTIFIER*, variable names like `"function"` were possible, but are sematically not allowed.

A lexical analyzer recognizes tokens within plain text, assigns a token type and disregards blanks. Potential problems are provoked when attempting to match null-strings. This frequently happens when using regular expressions, in particular the `0..*` operator. Conflicts like this can be solved by factorizing to separated productions. The resulting productions are in the form of `"<T1><T2>*" [1]`. This guarantees a minimum of one token being matched. An

expression like $\langle t \rangle +$ is essentially the same as $\langle T \rangle \langle T \rangle^*$. All matched tokens are appended to the token stream, which later interacts with a parser.

The parser distinguishes between lexical, syntactic and semantic errors. Lexical Errors occur when the lexical analyzer is not able to match a given input with known token types. The *Integer lexer* correctly recognizes "42 23 13" because "INTEGER INTEGER INTEGER" is lexically correct but syntactically incorrect. A parser throws a syntax error when the given input does not match the productions of the grammar. JavaCC provides good interaction between the lexical analyzer and the recursive descent parser which enables fairly good error recovering and reporting facilities. Many modern applications of parsers need enhanced error reporting, because humans need correctly reported error messages. Highlevel languages like Java do not stop parsing on an error occurrence immediately. Modern parsers try to recover as much as possible by injecting mock-tokens to isolate the actual error.

3.1 BNF/EBNF (Backus Naur Form)

Besides defining tokens, the grammar productions are specified as well. Productions use these defined tokens to make up the grammar. This is done by using a starting production which is the entrance to the recursive descent step. The lookahead defaults to $k=1$ so a default parse recognizes LL(1) unless a local lookahead[1] is specified.

Listing 2: Multiplication

```
1 void Start() : {}
2 {
3     Number() ( <MULT> Number() )* <EOF>
4 }
5
6 void Number() : {}
7 {
8     <INTEGER>
9 }
```

With JavaCC its possible to describe a grammar with EBNF instead of plain BNF. This means that useful constructs like the star-operator (regular expression) can be used. A recursive descent parser calls its (nonterminal) functions recursively ending up trying to match terminal tokens. If no production exists to match the input or not matchable terminal tokens occur, a syntax error is thrown. A lexical valid expression like "42 * * 13 55" will be erroneous in the parsing phase due to production violation. This is called a syntax error. The token-stream contains "INTEGER MULT MULT INTEGER INTEGER EOF" and fails when no production for the second *MULT* is found.

3.2 Choice positions

There are 4 different types of choice positions in JavaCC:

- An expansion of form: (exp1 |exp2 |...)
In this case the parser has to decide which non-terminal has to be chosen to continue parsing.
- An expansion of form: (exp)?
In this case the parser has to decide wheater or not to choose *exp* to continue parsing. Note: [exp] is an equivalent expression.
- An expansion of form: (exp)*
In this case the parser has to decide like in the case above wheater or not to choose *exp* to continue parsing. After succesful execution the parse has to decide again until desicion is negative.
- An expansion of form: (exp)+
This case is semantically equivalent to the case above except the fact that the parse has to choose *exp* at least one time. Note: This representation is essentially equivalent to (exp)(exp)*.

At these positions JavaCC can decide on its own which production to choose and which terminals to consume. In case a bigger lookahead than $k=1$ is needed, JavaCC needs information on how to handle the lookahead situation like limiting k to a local maximum.

JavaCC offers the possibility to solve ambiguous positions in grammars by using variable lookahead size. The author basically guides the parser generator for creating lookahead information. A full calculation of lookahead information is known as a NP-complete problem, so linear approximation or small k is practically needed. A prominent problem is the "dangling else problem"[7] which occurs in highlevel languages like Java, C, C# and others. Responsible for this problem is the variable parenthesis of statement blocks in conjunction with *IF/ELSE*. Regarding input like "if E1 if E2 S1 else S2" makes clear that the *else* tree could be assigned to the first *IF* as well as the second *IF*. According to a convention developers agreed with always assigning an *ELSE* to the most inner *IF*.

```

1 void Statements() : {} {
2     ... | ... | IfElse()
3 }
4
5 void IfElse() : {} {
6     "if" Expression() Statements() [ "else" Statements() ]
7 }

```

3.3 Lookahead

A lookahead is the number of tokens that have to be seen until a desicion can be made which production has to be chosen.

In many cases, left factorization can be used to resolve choice conflicts by hand. A production gets splitted in 3 factored productions to reduce a lookahead of k to $k-1$. Another solution is prefixing with a terminal token, so the parser can identify which production to choose. Both ways can make a grammar unnatural. Unnatural grammars contain many left-factorized functions, so the actual meaning of the grammar is difficult to grasp. In this case a lookahead of $k > 1$ could be practicable. Since JavaCC offers selective lookahead, a grammar can have $k > 1$ at critical positions only without factorizing by hand. The rest of the grammar can remain LL(1). Obviously, many grammars match equivalent languages. So the performance of a parser essentially depends on how the grammar is implemented. A plain LL(1) parser usually shows faster performance than an LL(3) parser. Terence Parr [9] pointed out that even in 1996 the loss of speed with a lookahead with $k > 1$ does not necessarily impact the performance for practical useage. Furthermore, the readability of a grammar can be increased by using a bigger lookahead (e.g. $k \leq 4$).

3.4 Conflicts

- First conflict

Two alternatives in the same production with identical first-terminals can not be identified by its followed symbol.

- First/follow conflict

A production that derives to ϵ is neither allowed to have terminals in its follow-set, nor to have them occur in the first-set of another production. Otherwise, the parser can not decide if to derive to ϵ or not.

- Left-recursion

A left-recursive production $A ::= A\epsilon$ leads the parser in an infinite loop and is therefore not processable.

3.5 Choice determination algorithm

```

1 void IfElse () : {} {
2     "if" Expression() Statements()
3     [ LOOKAHEAD(1) "else" Statements() ]
4 }
```

The *choice determination algorithm* is designed to test all possibilities and choose the first fitting production.

```

1 void value () : {} {
2     "<" expr () ">" // Choice 1
3 |    "(" expr () ")" // Choice 2
4 |    "new" expr () // Choice 3
5 }
```

The *choice determination algorithm* applied on the sample above would work as follows (in pseudo code):

```
1 if (next token is "<") {
2   choose Choice 1
3 } else if (next token is "(") {
4   choose Choice 2
5 } else if (next token is "new") {
6   choose Choice 3
7 } else {
8   produce an error message
9 }
```

The algorithm mentioned above works for productions with lookahead information as well. Technically, the parser tries to match all tokens and then decides whether to take a production or to refuse it. Another prerequisite which lookahead sets have to fulfill is to differ within the choices of a choice location. If this requirement is not met, a choice conflict occurs. Thus, the *choice determination algorithm* is applicable on any grammar with lookahead.

3.6 Syntax lookahead

JavaCC offers a syntactic lookahead method that uses a *choice determination algorithm*. Lookahead information in form of non-terminals is given to the parser at specific locations. The parser tries to match these non-terminals in case of a choice position. If a match was successful, the production behind the lookahead statement will be chosen for further recursive descending. This equals a *LOOKAHEAD(INFINITY)* where *INFINITY* is a big integer. This technique can be used if it is not clear how many tokens are needed to look into the future of the token stream. A practical example is the parsing of class definitions in highlevel languages.

Interfaces and classes in Java for example can only be separated if in case of a class the same has been parsed completely. In general limiting of a classes size is no good practical approach. Therefore a limit of *INFINITY* would fix the problem. The same effect can be achieved with using syntactic lookahead where at least one symbol is given to probe the lookahead.

```
1 LOOKAHEAD( ( "abstract" | "final" | "public" ) * "class" )
```

The calculation of the lookahead terminates as soon as the terminal *class* is found on the token stream.

3.7 Semantic lookahead

JavaCC offers a semantic lookahead what is especially used in conjunction with semantic actions. In this case a boolean expression is used as lookahead information which gets used by the parser in the particular context. This offers a

dynamic use of read tokens and its semantics so the decision of a production depends on a boolean expression.

General definition of lookahead in JavaCC:

```
1 LOOKAHEAD( amount, expansion, { boolean_expression } )
```

- "amount" defines the number of tokens lookahead.
- "expansion" specifies the production which gets probed until a decision is made.
- "boolean_expression" defines an expression in semantic lookahead which is responsible for choosing a production.

4 Semantics

Semantic actions get called to do semantic checks and produce intermediate code. These actions can be called directly from parser (syntaxdriven compiler) or executed in a separated pass while evaluating a syntax tree which gets generated (multi-pass compiler). Productions of a context-free grammar can be augmented with semantic code which are injected on right sides of the productions. This is called "translation scheme" [7];

A semantic action gets executed if the right side of the production has been resolved to the point where the semantic action resides. Semantic actions can be executed from every recognized symbol. Actions are embedded in the form of <Code> where this code consists of constructs of the compiler language (not the target language, the language the compiler is written).

4.1 Embedding semantic actions

Semantic actions can be embedded in JavaCC within grammar productions. This happens in the same language as the generated parser. In this case the semantic actions are written in Java code. These actions can be executed within a production and thus relates to the particular context. To access tokens within such semantic code the class *Token* is used. When a particular token gets consumed by the parser, this token can be assigned to a *Token* object. The reference of such a *Token* can be used within the semantic code. Semantic code can consist of arbitrary instructions.

```
1 Integer Number() : {  
2     Token t;  
3 } {  
4     t = <INTEGER> // grammar  
5     {return new Integer(t.image);} // semantic code  
6 }
```

4.2 One-pass compiler

A one-pass compiler executes all semantic actions while parsing the token stream. Actions can not be executed a second time without parsing the input again. This method has the advantage of being compact. There is no need for semantic classes, the calculation is done while parsing. It is clear to see that actions get executed even if a syntactic error occurs at the end of the token stream.

Listing 3: one-pass compiler

```
1 int Start() : {
2     Token t ;
3     int i ;
4     int value ;
5 }{
6     t = <NUMBER>
7     { i = Integer.parseInt( t.image ) ; // semantic code
8     value = i ; }
9     (<MULT>
10    t = <NUMBER>
11    { i = Integer.parseInt( t.image ) ; // semantic code
12    value += i ; }
13    )*
14    <EOF>
15    { return value ; }
16 }
```

The result of the non-terminal production *Start* with the given input "2*3*5" is 30. Another disadvantage besides ad-hoc execution of actions are context decisions. Since lexer, grammar and semantic actions reside in the same file, it can be confusing to select the type of action within the embedded semantic code and quickly leads to overloaded grammar files. If there is the need to do multiple passes of semantic actions, like analyze, transform, compile and interpret, this approach leads to complex grammars.

5 Abstract syntax trees

An abstract syntax tree is the abstract representation of the input in form of a tree. Syntax trees represent an acute granular form of intermediate code that contains syntactic and semantic information which can be used for analyzes and transformations.

5.1 Semantic classes

An abstract syntax tree consists of a tree of objects of which each represents a non-terminal. In real-world applications there is often no need to have atomic classes since a pragmatic approach reduces complexity. So not all non-terminals essentially need to be represented as a semantic class.

5.2 Paradigms of code execution

Abstract syntax trees can be produced in at least ways. Similar to the one-pass compiler all semantic classes can be instantiated (by hand) within the semantic code. This step also can be automated by generate a whole grammar file with semantic actions. This is called a tree parser.

5.3 Manual syntax tree construction

As mentioned above the syntax tree can be created by hand. A common recipe is to create semantic classes for all non-terminals, that inherit the same interface with a function e.g. *interpret()*. The grammar has to be augmented with semantic actions which instantiate these semantic classes at the particular non-terminal function in the recursive descent parser. It is common to pass a context object into the recursive functions to collect parsed data and construct the tree.

```
1 Number Number() : {
2     Number n;
3     Token t;
4 }
5 {
6     t = <INTEGER>
7     {return new Number(t.image);}
8 }
```

The non-terminal function which calls *Number()* will process the result of *Number()* what is an *Integer* object. On this way a implicit tree structure comes out. The manual tree construction has its strength in flexibility but also consistency checks regarding non-terminals and the interface for semantic classes have to be made by hand. As a matter of principle multiple passes could be implemented in the semantic classes to traverse the tree.

In real-world applications the interface provides functions like *analyze()*, *compile()* or *interpret()*. Compared to one-pass compilers the syntax tree is constructed one time while parsing but can be traversed multiple times. This is an advantage when reusing results of a previous pass is important.

The consequential disadvantage of this paradigm is to manually keep all semantic classes in sync regarding the interface. In addition the instantiation code is often very similar what leads to generating such code.

5.4 Visitor pattern

The visitor pattern is used in OOP (object oriented programming) and software engineering to separate object model from algorithm [10]. A practical aspect of separating is the possibility to extend or change the algorithm without the need of having to change the object model as well. The idea is to use structure classes which implement an *accept(..)* function with a special visitor object as argument. The visitor technically is an interface which provides a *visit(..)* function for each structure classes [10]. The *accept(..)* function subsequently

calls *visit(..)* of its own class. This enables the development of a particular visitor class which implements the visitor interface and thus provides access to all structure classes. This is separation of structure and algorithm and in this case the structure classes are the semantic classes and the algorithm is the actual pass of the compiler.

5.5 Tree parser

Another paradigm is code execution with generated tree parsers in conjunction with the visitor pattern. Here the grammar is used to generate another grammar with generated semantic actions and semantic classes. Also the visitor interface is generated. The generated semantic classes use the visitor pattern and implement the visitor interface. As mentioned above the structure classes and algorithms are separated. The implementation of the compiler resides in one single file, the visitor class. Therefore the semantic classes are compact and only contain the required visitor code and the compiler is not spread over dozens of semantic classes. It is common to implement an interpreter-visitor, compiler-visitor or even prettyprinter-visitor. Modern compilers like Jikes use this paradigm of writing visitors. This seems to be a sophisticated solution because the actual semantics are done in separate visitor files what gives a clear and scalable design.

5.6 JJTree and JTB

There are two tree parser generators for JavaCC, JJTree and JTB (Java Tree Builder). Both support the visitor pattern. Within the scope of this paper JJTree will be introduced since both tools provide almost the same functionality. Most of statements also apply to JTB.

JJTree generates semantic classes for all non-terminals in a given grammar which automatically get instantiated within the semantic actions. Against the manual tree construction the abstract syntax tree initially contains no semantic information [11]. JJTree generates such classes when providing the following options:

```
1 options {  
2     MULTI = true;  
3     VISITOR = true;  
4 }
```

JJTree creates classes of type "AST*.java" and when they are generated once JJTree does not overwrite them again on parser generation. This aids when implementing member variables in semantic classes. Furthermore the interface "*Visitor.java" which contains functions for all semantic classes is generated every time the grammar changes [11]. A visitor is created by constructing a class that implements this interface. It's clear to see that the auto-generated interface cares about visitor sync. The visitor will not compile until e.g. a newer version of the interface is correctly implemented.

Listing 4: An example of visitor functions.

```
1 public Object visit(ASTStart node, Object data) {
2     // code before recursion
3     data = node.childrenAccept(this, data);
4     // code after recursion
5     return data;
6 }
7
8 public Object visit(ASTMult node, Object data) {
9     // code before recursion
10    data = node.childrenAccept(this, data);
11    // code after recursion
12    return data;
13 }
```

It is obvious that a recognized non-terminal is passed into the visitor function. The `JJTree` function `childrenAccept()` calls all underlying child-objects in the parse tree. In relation to the example *one-pass compiler* the symbol *Start* may contain childrens of type *Number*.

6 Conclusion

This work showed three different paradigms of language translation while introducing JavaCC and JJTree. First, the usage of grammars and lookahead with `kl` from a practical point of view has been discussed. This leads to semantic actions and the relevance of semantic and syntactic lookahead in modern languages. As analyzed in several chapters of this paper, usage of lookaheads `kl` is practicable. Apart from that, the paradigms of language translation, like one-pass compilers and abstract syntax tree traversal, have been introduced. I showed in particular how the paradigms of translating one abstract data structure into another one have different areas of responsibility.

Pointing out where one-pass compilers might take place and where multiple traversal is an essential part is one of the most important aspects of this work. Compiler construction is present, visible or hidden in many relevant areas where automatic language recognition is used. Language recognition and translation is not only used for classical programming languages, furthermore it is used recognizing complex data structures in modern datamining applications or cross-compilers in highlevel languages.

Also, generic code invocation and generation becomes more and more important in real world applications. Cross compilation between highlevel languages is as well a challenging topic, delivering many possibilities, like automatically convert e.g. JDK sourcecode to e.g. native *C#*, for further investigation. Also interesting could be generation of Java intermediate code to execute PHP.

The conclusion is that language translation tasks can be solved with different approaches which address various real-world applications, so this seems to be a promising development for modern applications.

References

- [1] ANDREW W. APPEL, Jens P.: *Modern Compiler Implementation in Java, Second Edition*. Cambridge University Press, 2002
- [2] LESK, M. E. ; SCHMIDT, E.: Lex — A lexical analyzer generator. In: *Computing Science Technical Report No. 39, Bell Laboratories* (1975)
- [3] JOHNSON, S. C.: Yacc: Yet another compiler compiler. In: *Computer Science Technical Report #32, Bell Laboratories* (1975)
- [4] NICOL, G. T.: *Flex: The Lexical Scanner Generator*. 1993
- [5] DONNELLY, Charles ; STALLMAN, Richard: *Bison: the Yacc-compatible parser generator*. 1990
- [6] TERENCE PARR, Russell Q.: ANTLR: A Predicated-LL(k) Parser Generator. In: *SOFTWAREPRACTICE AND EXPERIENCE, VOL. 25(7)* (1995)
- [7] ALFRED V. AHO, Ravi S. ; ULLMAN, Jeffrey D.: *Compilers, Principles, Techniques, and Tools*. 1986
- [8] PARR, Terence J.: *Obtaining Practical Variants of LL(k) and LR(k) for k*, Diss.
- [9] TERENCE PARR, Russell Q.: LL and LR Translator Need k. In: *SIGPLAN Notices Volume 31* (1996)
- [10] STEVEN J. METSKER, William C. W.: *Design Patterns in Java*. Addison-Wesley Longman, 2006
- [11] SUN, Microsystems: *JJTree Reference Documentation*. Web: <https://javacc.dev.java.net/doc/JJTree.html>, 2006