A
TRANSLATOR WRITING SYSTEM
FOR A
JAVA ORIENTED COMPILER COURSE

By

HANS-GEORG LERDO

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

This document is dedicated to the students of the University of Florida.

ACKNOWLEDGMENTS

I thank my wife and my two sons for their continuous support and patience with my work schedule and amount. Without them this project could not have been completed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A TRANSLATOR WRITING SYSTEM FOR A
JAVA ORIENTED COMPILERS COURSE

by

Hans-Georg Lerdo

May 2003

Chair: Manuel E. Bermudez
Major Department: Computer and Information Science and Engineering

This thesis is not about how to build a parser. It is not about how to build a lexical analyzer. It is not about building a “thing” that revolutionizes the compiler area of computer engineering in some form or another. Many good tools have been developed for that. Although sometimes not necessarily brand-new, they still perform well and produce good results.

While covering different fields (e.g., lexical analysis) and taking different approaches to a problem (e.g., LL(1) vs. LR(1) and LALR(1) parsing) these tools have one thing in common: They all require a specialized input depending on the respective tool. Suppose one wants to change the grammar for RPAL to be written in German as opposed to English. The grammar is probably at hand in some standardized form. Maybe it even exists in BNF or EBNF notation. The chances are this form has to be modified to comply with the input formats required by the respective tools.

The majority of tools exist for the C/C++ segment of programming languages. This thesis will deal with the Java side, specifically the design of a TWS for use in the academic environment.

The goal of this thesis is taking existing programs used in compiler generation and combining them to form a new tool. This tool will be a program that takes a uniform, standardized language grammar as its input and produces output in standardized form for further use.

The presence of the key tools will be transparent to the user. The program forms a “black box” that takes standardized lexical and grammatical information as its input and produces native Java source code. This code can be compiled to form a compiler for program source code that was written in compliance with the new grammar.

This paper is not about dry and sterile presentation of research results. I tried to make this rather technical material more approachable and fun to read. It is my opinion that something that is fun to read is easier to comprehend and to learn.

CHAPTER 1
MATHEMATICAL PRELIMINARIES

Before we present the actual translator construction I will “warm up” by covering some of the important basics that will come up all the time. The following material is not intended to explain the respective topics in every detail. They are merely an overview of what needs to be in the readers’ memory when dealing with construction details of a writing system. A reader familiar with the basics in lexical analysis and parsing procedures may skip through to the actual description of the writing system.

Context Free Grammars

Syntax Analysis of a language happens in two steps: scanning and parsing. Scanning breaks up the text source of a language into individual ***tokens***. These tokens constitute the basic elements of the respective source. Scanning is also called ***Lexical Analysis*** and a scanner is sometimes referred to as ***lexer.*** Its main purpose is to make the “life” of the parser easier: it generates meaningful textural elements out of individual characters. The parser is now presented with the tokens, verifies that these tokens constitute a grammar-compliant sentence by processing them according to a set of given ***production rules***. These production rules are used to generate a parse tree, which used later on for further processing.

The set of rules is called a Context Free Grammar, or CFG [1].

BNF Notation

In the late 50s Philadelphia born mathematician John Backus [2] and Danish computer scientist Peter Naur [3] were the first to present a formal notation to describe the syntax of a given language. This notation was consequently called the ***Backus Naur Form***, or ***BNF*** [4]. The following gives a brief overview of the BNF notation:

The meta-symbols of BNF are shown in Table 1-1:

Table 1-1. BNF meta symbols

::=	meaning "is defined as"
|	meaning "or"
< >	used to surround category names.

The angled brackets distinguish syntax rule names (also called nonterminal symbols) from terminal symbols, which are written exactly as they are to be represented. A BNF rule defining a nonterminal has the form:

nonterminal ::= sequence of alternatives consisting of strings of terminals or nonterminals separated by the meta-symbol “|”.

For example, the BNF production for a mini-language is shown in Figure 1-1:

```
<program> ::=      program
        <declaration_sequence>
        begin
        <statements_sequence>
        end ;
```

Figure 1-1. BNF notation for a mini language

This shows that a mini-language program consists of the keyword "program" followed by the declaration sequence, then the keyword "begin" and the statements sequence, finally the keyword "end" and a semicolon [5].

EBNF Notation

It often is the case that production rules repeat themselves with only slight variations. For example: In a grammar for a simple calculator there have to be rules for

the basic algebraic operations addition, subtraction, multiplication, and division. In BNF these would be described like shown in Table 1-2:

Table 1-2. BNF example

```
operator ::=   +
operator ::=   -
operator ::=   *
operator ::=   /
```

In this case only four rules are sufficient to cover algebraic operations. In case of a string in a programming language--it can cover ALL possible ASCII characters--this would be quite inconvenient.

Over time a few “shorthand” terms have come into use. These vary slightly from author to author, but are essentially the same in their meaning. The most common one is the vertical bar “|”, which represents “or”. In case a nonterminal symbol is part of several production rules the “|” simplifies their description. The above mentioned example could be expressed as the following using the “|”:

```
operator ::=   + | - | * | /
```

This representation is called the ***Extended Backus Naur Form***, or ***EBNF***. Unlike BNF the notation for EBNF is not clearly defined. On other occasions a symbol can appear in multiple instances. The use of “{}” and “[ ]” varies widely. Some authors use [ ] to indicate one ore more instances, some to indicate optionality, and {} to indicate zero or more instances of whatever symbols are inside.

In the course of this paper we will use the notation common at the University of Florida (Note: “::=“ is equivalent to “→“ and “ε“ denotes the empty string) as shown in Table 1-3:

Table 1-3. EBNF notation used by the University of Florida

A →	a?	⇔	optional instance of a (i.e. A -> a | ε)
A →	a+	⇔	one ore more instances of a (i.e. a, aa, aaa, ...)
A →	a*	⇔	zero ore more instances of a (i.e. ε, a, aa, aaa, ...)
A →	a list b	⇔	a (ba)*

The vertical bar will be used as indicated above.

Abstract Syntax Tree

Before we go into different parsing techniques there is one more thing to talk about. How is a source represented after it has been parsed? A tree structure has become the one of choice. This tree is referred to as the ***Abstract Syntax Tree***, or ***AST*** ([6], p86).

Suppose we have the following (very) simple grammar:

Sum → <integer> '+' <integer>

If the input "4 + 7" is parsed and placed into a tree structure it would look like shown in Figure 1-2:

```
Sum
 |
 +
/ \
4   7
```

Figure 1-2. A tree structure

The resulting tree is straight forward and the only one that can be produced by this grammar. The result of the calculation is the same in every case. If the grammar is now modified in a certain way the result is not clear-cut any more. For example:

Use "4 + 7 * 8" as input for the following grammar:

Sum → <integer> '+' <integer> '*' <integer>

If one starts parsing on the left the parse tree looks like in Figure 1-3:

```
      Sum
       |
       *
      / \
     +   8
    / \
   4   7
```

Figure 1-3. Another tree structure

On the other hand, if one starts parsing on the right, the tree looks like in Figure 1-4:

```
      Sum
       |
       +
      / \
     4   *
        / \
       7   8
```

Figure 1-4. Yet another tree structure

The results are "88" for left-start (left-to-right) and "60" for right start (right-to-left) parsing.

There is clearly a difference in the parse result depending in what direction the parsing takes place. But not only the direction is important. If the production rules contain recursion the direction in which the recursion is resolved changes the way a parser can operate. Recursion is usually re-written into iteration. The direction of this re-write is, as already mentioned, non-trivial. The fact that a single source sentence can be represented in more than one parse tree makes the respective grammar ambiguous. While they have some use, ambiguous grammars will not be further discussed here. The TWS will have mechanisms that try to solve ambiguity and produce a single, unique tree and consequentially, unambiguous and executable code.

LL vs. LR Parsing

The first L in LL(1) stands for Left-to-Right parsing. The second L stands for Left-most-Derivation. This means that in a nonterminal's production rule that contains other nonterminals on its right side of the rule the left-most nonterminal is re-written into its respective rule's right side. If that rule also contains nonterminals the process repeats itself. ASTs by LL() parsers have a tendency to grow to the left more than to the right due to this derivation behavior. LL() parsing is also known as “top-down-parsing”. The parsing proceeds from the "root of the tree" (i.e. the start of the program) into each of its leaf nodes and then builds the final tree out of the several sub-trees on the way back up. Take the example given in Figure 1-5, which produces a sequence of a's (i.e. a*) followed by a sequence of b's (i.e. b*):

S	→	A B
A	→	A a
	→	ε
B	→	B b
	→	ε

Figure 1-5. Sample grammar

Assume a sentence “aabbb”. The LL(1) parser would proceed the following way:

- Take the first “a” as a token, take the second “a” as the look-ahead token
- Start with the production rule for S: it states that “A” is next. So the tree root “S” is created with “A” as its left child. Then the production rule for “A” is called.
- Check A’s production rule: Either “A” is next or the empty set. The look-ahead token resolves the ambiguity: since “a” is the next token in line “A” is next and within this production rule the same production rule is called again; new look-ahead token is the first “b” now. A new tree node “A” is created and the left child is set up following the mentioned production rule.
- This time the production rule for “A” states the empty set as the valid one. A sub-tree with “A” as root and “ε“ as its left child is created the production rule returns to the calling production rule.

- There the sub-tree is added as the left child of “A” with “a” as right child and passed to its calling production rule
- The same is done again, this time for “S” as the root. Now the first part (i.e. left sub-tree) of S’s production rule is done and the rules for “B” are entered. They proceed in the same way as the rules for “A” and result in the final tree as shown in Figure 1-6:

```
                S
            ___/ \___
           /         \
          A           B
         / \         / \
        A   a       B   b
       / \         / \
      A   a       B   b
     /           / \
    ε           B   b
               /
              ε
```

Figure 1-6. Top-down-parsed and top-down-built parse tree

Note that the lowest-level “A” and “B” are not really necessary to describe the tree and can be ignored. The way to do this is to build the tree bottom up. The tree would then look like in Figure 1-7:

```
                S
            ___/ \___
           /         \
          A           B
         / \         / \
        A   a       B   b
       / \         / \
      ε   a       B   b
                 / \
                ε   b
```

Figure 1-7. Top-down-parsed and bottom-up-built parse tree

In short: the LL(1) parser decides on the right production rule depending on what is ahead and so far unparsed. It keeps no information on where it has been so far in the parsing process. If an ambiguity arises despite the look-ahead, the parser has no means of

resolving it and throws an error. One could set the look-ahead to a higher number, but the problem remains: other than the look-ahead there is no method to recover from ambiguities.

LR() parsers still use a certain number of tokens as a look-ahead, but also keep track of where they are in the parsing process. They do by means of a stack onto which they push information about the state that they just left. In order to have such information LR() parsers are structured differently than LL() parsers. They are built as ***Deterministic Finite-State Automata***, or ***DFA*** ([6],pp66). Each set of production rule is considered a state and each production rule that is going into a nonterminal is a transition into another state. Consider the grammar Figure1-4. “S” is always followed by “A”. So let’s call this state (i.e. “S”) state (1). “A” either moves into “A” again or into “ε“. Let’s call “A” state (2). And similarly “B” is state (3). So far we have identified all the states. In order for the grammar to be unambiguous each state must have a unique transition into another state. In this case (3) can only transit into (2). Note that although “B” is present in the production rule it is not considered in the transition listing! Now for “A”: (2) can transit into (2). What happens with the rule where “A” produces the empty set? Now the rule for “S” needs to be considered again. After “A” follows “B” in this rule. Hence the transition into the empty set for “A” is equivalent with the transition into the rules for “B”, hence (2) goes into (3). After that (3) goes into (3) or completes the parsing process (i.e. “B” produces the empty set), since there are no further elements after the “B” in the production rules for “S”. In summary we get the state transitions as given in Figure 1-8:

(1) → (2)
(2) → (2),(3)
(3) → (3), done

Figure 1-8. State transitions for grammar

With 1 token look-ahead this grammar is unambiguous. If we were to change the production rule for "S" into "S → ABA" the whole picture changes. Now the sentence "aa" is ambiguous. Now there are several possibilities:

- a non-empty "A" followed by an empty "B" and an empty "A"
- a single "A" followed by an empty "B" and another "A"
- an empty "A" followed by an empty "B" followed by a non-empty "A"

This case is ambiguous even with 1 (or more) token look-ahead.

Now consider the sentence "aaaba". With 1 token look-ahead the first two a's still have the same ambiguity as stated above. But now the parser will encounter the b in the middle. This b clarifies the previous a's as being part of the first "A" in the production rule for "S". If the parser had considered these a's as being part of the second "A" in the production rule for "S" it could trace back the state transitions by popping the transitions off the stack, back to the point where the first "A" starts, and start the parsing of the sentence from that point on. One can picture this as something like having 3 streets to choose from. One is the "through-street," and the other two are dead ends. Choose one, proceed until we run into a wall, go back, and take the next one. If the next one is also a dead end, go back again and take the next (i.e. last possible) one. In short: just try all the transitions until you find one that leads to the acceptance of the production.

It is easy to see that LR() parsing is much more resistant to ambiguities then LL() parsing. Yet it is not foolproof (as seen in the "aa" sentence above) and might take much longer then an LL() parser, since several possibilities have to be probed. LR() parsers can

process more languages than LALR() parsers. Then why are there so many LALR() parsers around? The reason is that for LALR() vs. pure LR() parsing the "roads to choose from" are limited in LALR() parsing. The transition tables in LR() parsing grow beyond any practicality[1].

Which parsing approach is the better one? The all have their advantages and disadvantages, obviously. And as in many other cases the decision-making comes down to weighting performance against requirements. All parsers require tables with parsing results in order to check the grammar for ambiguities. LL(1) tables are smaller than LALR(1), by a ratio of about two-to-one. LR(1) tables are too large to be practical. Time wise, both LL(1) and LR-family parsers are linear for the average case (in the number of tokens processed). […] Most language designers produce a LALR(1) grammar to describe their language. The LR-family grammars can also handle a wider range of language constructs; in fact the language constructs generated by LL(1) grammars are a proper subset of the LR(1) constructs. For the LR-family the language constructs recognized are:

> LR (0) << SLR(1) < LALR(1) < LR(1)
> LL(1) is almost a subset of LALR(1)
> where << means much smaller and < means smaller" [7].

In order to maximize possible languages while still keeping an eye on acceptable memory requirements and performance I chose LALR() parsers for this TWS.

[1] This is presently being debated. 'Large' parse tables by 1970's standards need not be 'large' by today's hardware and software standards.

CHAPTER 2
EXISTING TRANSLATOR WRITING SYSTEM

Now we are approaching the heart of this project. First, we will present the original TWS with its components and lines of thought. Taking this as the basis we will design the new system, shown in Chapter 4. The basic idea will be the same, yet it will be as close to OOP as possible. A certain “script feel” can not be avoided, however, since we are dealing with a sequence of some sort:. This “script” will take place as one final main method that takes the file names as parameters and then goes through the process of creating the compiler, which in turn takes the source as an input in order to compile it and run it on the interpreter.

What is a TWS?

In general, a TWS is a system that takes as its input lexical and grammatical information of a language and creates a program that translates source code into object code. This object code can be compiled into an executable or run on an interpreter for the respective language. There are different areas of use for every language. A “open file” statement in needs to check for the existence of the chosen tables while for example a for-loop in needs to keep track of the loop variable. This information is contained neither in the lexical information nor the grammar. Hence the TWS also needs execution instructions for the respective language in order to be able to build the object code. This code is used in the code generator. Any errors in the source code with respect to scoping and variable declarations are caught in the constrainer. Figure 2-1 shows the general TWS setup [6]:

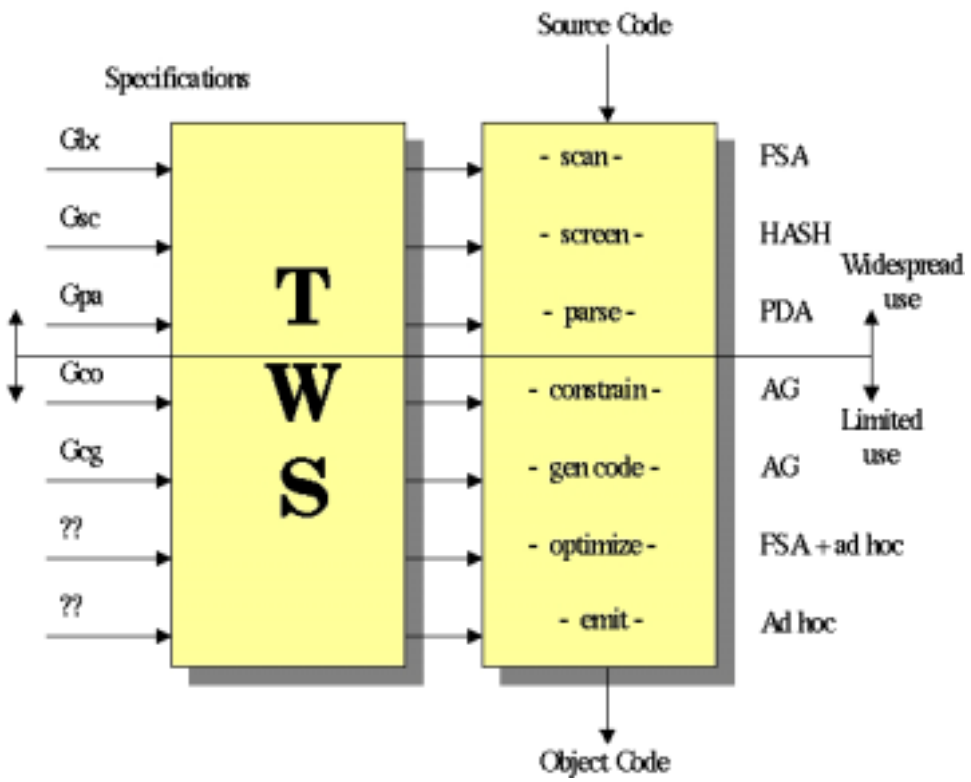

Figure 2-1. Overview of a TWS

Note: “Glx” stands for “Grammar Lexer”, “FSA” for “Finite State Automaton”, and so forth. The old setup had all the components except for the optimizer. The new setup will be the same in its basic general setup as shown above, yet the specifics will differ slightly from the old system.

Previous System Setup

For the presentation of the previous system we will only show its structure and intended functionality. The code of the system will not be discussed here. The system was limited to the C dialect ***Tiny***. Although it was possible to implement other languages the process of doing so would have been very inefficient. All of ***pgen*** and the ***lex/yacc*** input files would have had to be redesigned. The system was completely written in C in

1991. No OOP principles were used. Figure 2-2 shows the flow of information in a simplified manner [6]:

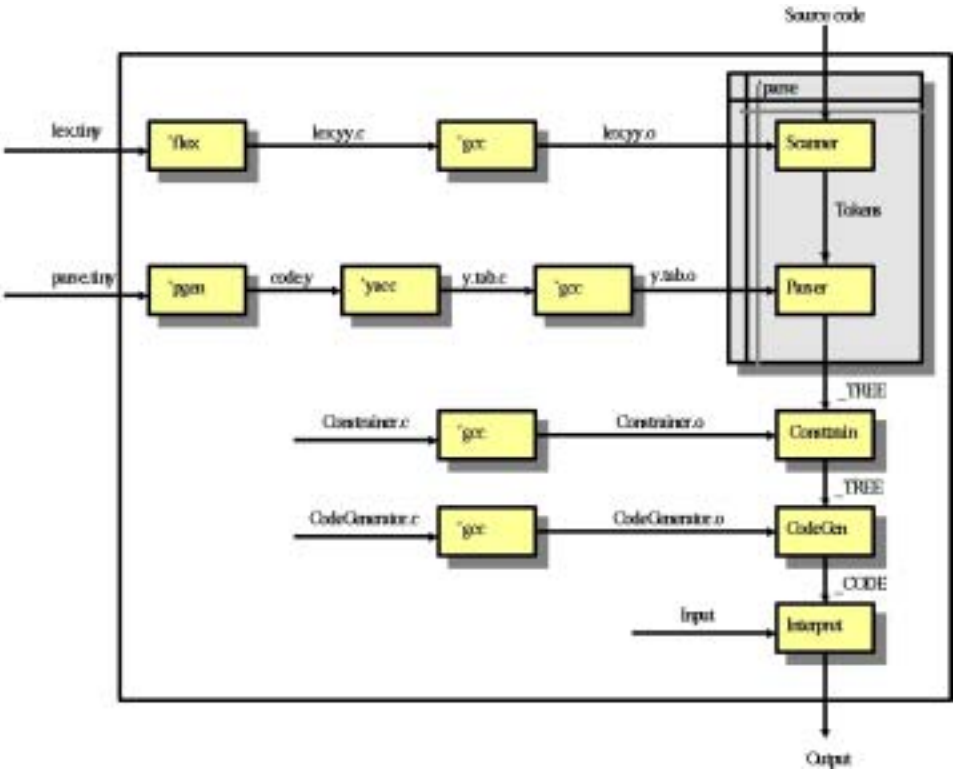

Figure 2-2. The Tiny Compiler/Interpreter

The parser is created by feeding Flex with a source file containing semantic rules (***lex.tiny***). After compiling Flex's output file ***lex.cc.y*** we have the scanner part of our parser.

The component ***pgen*** is the one building the actual parser. The shown data is strongly simplified to make the overall sequence of events clearer. Figure 2-3 shows the detailed data flow through ***pgen***. Note that ***pgen*** is a parser in itself! Since the file ***parse.tiny*** is a source file just like any source code for the final TWS it has to be parsed.

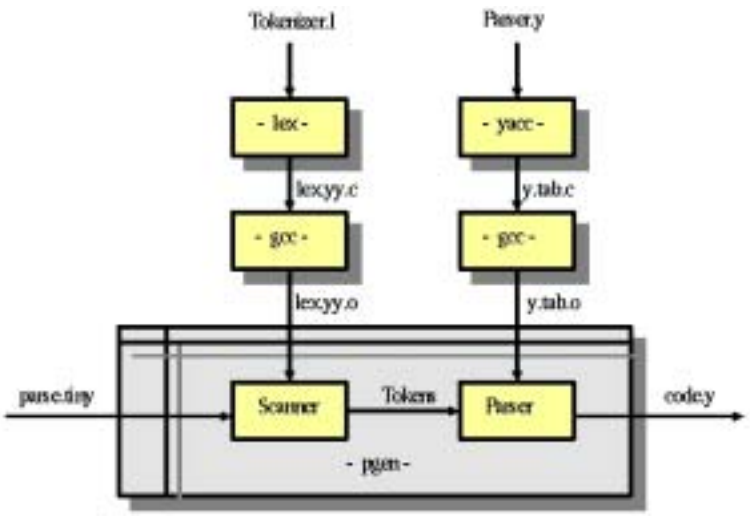

Figure 2-3. Detailed data flow through ***pgen***

Now referring back to Figure 2-2: the source code parser took ***Tiny*** source code as its input, scanned and parsed it and produced a tree as output. This tree not only contains the source code tokens in their respective positions depending on the grammar, but also was accompanied by TWS support modules, i.e. declaration tables. These tables contain variable values with respect to the different scopes created by the source code. The constrainer then checks if there are no scope violations. Scopes take into consideration if a variable is known within a certain method of a program. A very close description of scopes can be done referencing global and local variables. Although a variable declared globally is know locally as well, it can be re-declared locally to represent ***something completely different***. Within that scope the new declaration has precedence. When the scope is closed (e.g., at the end of a method of loop of some sort), the global declaration takes effect again. Consider for example the following pseudo code segment with explanations as comments [8]:

```
proc A;                  // Open a scope.
   var x: integer;       // Enter x (integer) in the current scope.
   y: boolean;           // Enter y (boolean) in the current scope.
   proc B:               // Open another scope.
      var x: boolean;    // Enter x (boolean) in the current scope.
      y: integer;        // Enter y (integer) in the current scope.
      begin {B}
         x := true;      // Lookup x (boolean). The outer x is invisible.
         Y := 1;         // Lookup x (integer). The outer y is invisible.
   endproc {B}           // Close current scope. Restore all variables
   begin {A}             // to what they were IN THE PREVIOUS SCOPE.
      X := 1;            // Lookup x (integer).
      y := true;         // Lookup y (boolean).
endproc {A}
```

Figure 2-4. Scoping example

The following example shows the local knowledge of global variables and their local re-declaration. Note the “Dx” stands for “Declare x” and “Ux” for “Use x” [8]:

```
begin                    // OpenScope
      Dx                 // Enter (x,1).  The 1 means tree location 1.
      Dy                 // Enter (y,2).  This is tree location 2.
      begin              // OpenScope.
            Dx           // Enter (x,3).  This is tree location 3.
            Ux           // Lookup (x) should return 3.
            begin        // OpenScope.
                  Dx     // Enter (x,4).  This is tree location 4.
                  Dz     // Enter (z,5).  This is tree location 5.
                  Dx     // Enter (x,6).  This should be an ERROR!
                  Ux     // Lookup (x) should return 4.
                  Uy     // Lookup (y) should return 2.
            end          // CloseScope.
            Ux           // Lookup (x) should return 3.
      end                // CloseScope.
      Ux                 // Lookup (x) should return 1.
end                      // CloseScope.
```

Figure 2-5. Another scooping example

Observe the x can be declared only once within the same scope, but multiple times over multiple scopes!

The constrainer passed on a new tree that is free of semantic errors as input to the code generator. Here the tree was translated into a sequence executable by an abstract machine. The sequence was specially tailored to suit the interpreter. It can be re-constructed later to produce input for more sophisticated code generators that produce executable code that can be run on various processors and operating systems. The old

code generator was written particularly for ***tiny*** and the interpreter that was attached to the TWS. Any other language would have required a re-write of the constrainer and code generator. Modularized setup of the new system will make it easier and more flexible in that respect.

CHAPTER 3
TOOL OVERVIEW

The following section presents a few tools for the programming languages C/C++ and Java. The TWS will be constructed with Java as the language of choice for the system implementation. The presentation of the Java tools will be slightly more detailed to explain why a tool was chosen in the TWS.

C / C++

The number of tools for C/C++ is quite large; too large to mention them all. We will only provide a closer look at those that we consider the most important. The interested reader can research further tools by starting at the given web site with an extensive list of many compiler tools [9].

Bison

Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Once you are proficient with Bison, you may use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages [10].

Bison is Yacc compatible and was meant to be a replacement. While Yacc is more C oriented Bison implements the OO aspect by using C++ code and output.

BYacc

BYACC/Java is an extension of the Berkeley v 1.8 YACC-compatible parser generator. Standard YACC takes a YACC source file, and generates one or more C files from it, which if compiled properly, will produce a LALR-grammar parser. [...] This is the standard YACC tool that is in use every day to produce C/C++ parsers. I have added a "-J" flag that will cause BYACC to generate Java source code, instead [11].

This is Yacc with an extension for Java source code. Since this program is implemented in C/C++ it was not chosen.

Flex

Flex is a tool for generating scanners. It generates them as a C source file ***lex.yy.c***, which defines the important method yylex(). Compiling this file with the “-lfl” flag (linking it to the respective library) produces the executable. During its execution it analyzes its input for occurrences of the defined expressions. Whenever it finds one, it executes the corresponding C code, which was defined in the input file for Flex [12].

This tool was used along with Yacc to construct the parser in the original TWS. Figure 3-1 shows the Flex setup specialized to the C dialect “tiny”:

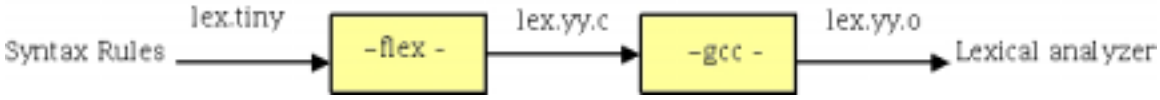

Figure 3-1. Data flow using Flex

Flex produces methods that provide a parser with tokens to parse.

Yacc

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process [13].

Figure 3-2 shows the flow of data through yacc in order to generate an executable parser:

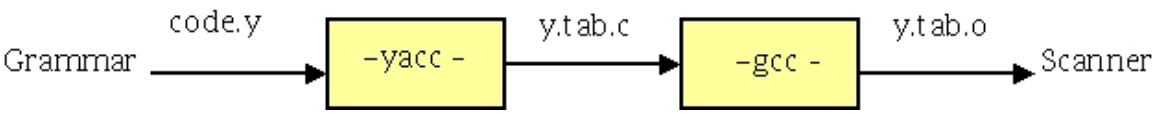

Figure 3-2. Data flow using Yacc

code.y contains not only grammar information, but also “reaction code”, specifying which methods have to executed for which grammar rule.

Java

For Java the number of available tools is quite limited compared to C/C++ due to the fact that Java is a relatively young language. Despite quite common in the internet and cross-platform community, Java has yet to prove itself as a language for serious applications. Yet there are tools out there. See reference: [9]

Jay

Jay takes a grammar, specified in BNF and augmented with semantic actions, and generates tables and an interpreter, which recognizes the language defined by the grammar, and executes the semantic actions as their corresponding phrases are recognized. The grammar should be LR(1), but there are disambiguating rules and techniques [14].

The fact that Jay is implemented in C/C++ and ***only produces*** Java code led me to dismiss this tool.

JavaCC

Java Compiler Compiler (JavaCC) is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc [15].

This parser has numerous ready-made grammars [16] available and is often used by a variety of users. Since Java CC is a LL(1) parser it was not chosen for this project despite its clear input syntax, tree building tools, flexibility, and easy of use.

Java Cup

The Java based Constructor of Useful Parsers (CUP) is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC, and in fact offers most of the features of YACC. However, CUP is written in Java, uses specifications including embedded Java code, and produces parsers which are implemented in Java [17].

This tool almost became the tool of choice for the parser section of the project. CUP has support built into the ***lexer*** of choice (JFlex). The philosophy for this project was to keep the system modularized. This means that all tools within the system must be exchangeable. If tools were chosen that are too closely linked to each other the overall design might suffer from this situation. It is my opinion that by using tools that are not developed for one another the overall design stays more flexible.

JFlex

JFlex is a lexical analyzer generator (also known as scanner generator) for Java, written in Java. It is also a rewrite of the very useful tool JLex which was developed by Elliot Berk at Princeton University. As Vern Paxon states for his C/C++ tool flex: They do not share any code [18].

With the same way of use as its C/C++ counterpart and complete implementation in Java this would have been the tool of choice for the lexical analysis part of the project. It would have been used as a standalone tool without emphasizing its support features for respective parsers (e.g., CUP), so that it can easily be exchanged with another tool later on should the need arise. Since SableCC has a built-in lexer we am going to use its own lexer for now. The implementation will be transparent to the user with respect to this fact. More explanations on this subject will be implemented in Chapter 4.

SableCC

SableCC is an object-oriented framework that generates compilers (and interpreters) in the Java programming language. This framework is based on two fundamental design decisions. Firstly, the framework uses object-oriented techniques to automatically build a strictly typed abstract syntax tree that matches the grammar of the compiled language and simplifies debugging. Secondly, the framework generates tree-walker classes using an extended version of the visitor design pattern that enables the implementation of actions on the nodes of the abstract syntax tree using inheritance. These two design decisions lead to a tool that supports a shorter development cycle for constructing compilers [19].

The reason why SableCC was chosen for this project lies in its general philosophy and setup as shown in Figure 3-3:

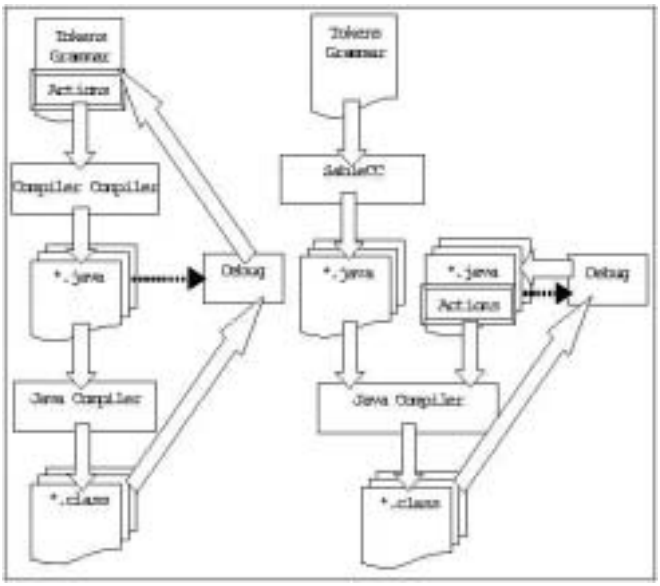

Figure 3-3. SableCC data processing flowchart

Unlike yacc and similar working tools SableCC separates the grammar from the code that needs to be executed if a rule takes place.

CHAPTER 4
TRANSLATOR WRITING SYSTEM

Now we are finally approaching the heart of this project. First, we will present the original TWS with its components and lines of thought. Taking this as the basis we will design the new system. The basic idea will be the same, yet it will be as close to OOP as possible. A certain “script feel” can not be avoided, however, since we are dealing with a sequence of some sort:. This “script” will take place as one final main method that takes the file names as parameters and then goes through the process of creating the compiler, which in turn takes the source as an input in order to compile it and run it on the interpreter.

New System Setup

What are the goals for the new design? The following list summarizes the overall design decisions that came up on several occasions in the previous chapters:

Table 4-1. New TWS design descicions

1. complete implementation in java
2. separate and standardized input files for
   - lexical definitions
   - grammar
   - execution rules
3. LALR(1) parsing
4. tool transparency
5. tool interchangeability

The motivation behind this was the idea of being able to use the same input files regardless of what lexer/parser tools were going to be used. This is pretty much the same idea as in cars and tires. If you want to use different tires you “just” change the wheels and not the entire car. If a new and improved tool came to be one had only to change the preprocessors in order to use all the previously constructed languages. This is not limited

to the idea of java programs. we can imagine a translator setup that translates native Visual C++ code for DOS into REXX code for IBM's OS/2. There are no limitations here. And with the TWS running on a OS independent platform its use is highly flexible.

The following subsections will deal with individual issues and also go into more detail about some of the specifications not visible in the overview.

Abstract Syntax Tree Processing

The code generator will be presented with an AST. Each lexer/parser tool has a more or less unique representation of the generated parse tree. Instead of changing the different AST for every single tool in order to correlate a tree node with its respective execution methods a connector needs to be defined that uniquely associates the tree node with that method. This connector will require a standardized input in which all the methods and variables that refer to tree nodes need to be named in a predefined matter. This predefinition will then be translated into the respective syntax for the used tool.

Input File Syntax

In order to comply with a standard we could make up my own. But that would defeat the purpose somewhat. To enhance structure and readability we will rely on the EBNF notation without special multiplicity rules for parenthesis and use of the metasymbol "|" to separate individual production rules for a single production. Specifics for the lexical rules and grammar files that will be shown in detail in the respective paragraph.

Lexical and grammar files

Lexical and grammar files basically have the same setup. Token names for the lexical file (lexicon) are tree node names for the grammar file. The basic structure of a token definition looks like this (the used notation is NOT conform to any standard this time):

```
<word name> -> <word description> (=> ‘<token name>’)? ;
```

Observe that the construct with the name of the token is optional! Not always are there only token names defined, but some words can help define tokens without being tokens themselves. For example:

```
Integer     ->    Digit+                  =>    ‘<INT>’ ;
Digit       ->    [0..9] ;
```

The basic structure for a grammar production rule looks like this:

```
<rule name>  -> <rule description> (=> ‘<node name>’)? ;
```

One can see the similarity. The rule description can consist of tokens (i.e. terminals) and other production rules (i.e. nonterminals). Yet with this notation there could only be one actual production rule for each rule name. Hence we need to amend this structure. Having said that we could re-write the production rule like this:

```
<rule> ( -> (<rule> | ‘<token>’)*  (=> ‘<node name>’)? )+ ;
```

With this structure a construct like this is possible:

```
foo    -> ‘(‘ foobar ‘<INT>’ foobar ‘)’    =>    ‘foo’
       -> ‘(‘ ‘<INT>’ ‘)’                  =>    ‘fooInt’
       -> ‘(‘ ‘)’ ;
foobar -> ‘<STRING>’ foo ‘<STRING>’
       ->    ;
```

The second production rule for ***foobar*** can lead to misunderstandings. In my project we will define this rule as the equivalent to the empty production:

```
foobar -> ε ;
```

This refinement is valid for the lexicon also. The re-write for the token structure looks like this:

```
<word> ( -> (<word> | '<char>' | <sys>)*  (=> '<token name>')? )+ ;
```

Here the <sys> represents any end-of-line, line feed, return, tabs, and the like. This was placed in the structure to distinguish between "ordinary" characters and return symbols and the like.

So a processor is needed that takes the two files as its input, digests them, and spits out the files that are conform to the syntax requirements of the used tools. Basically this means another compiler! Despite the fact that to a certain extend it is a re-write of the input file we will use a compiler generator tool for it. This way the whole system is more consistent in itself and maintenance will be easier. The specific processor for SableCC will only spit out ONE file since the lexical and grammar information is combined in one file for SableCC.

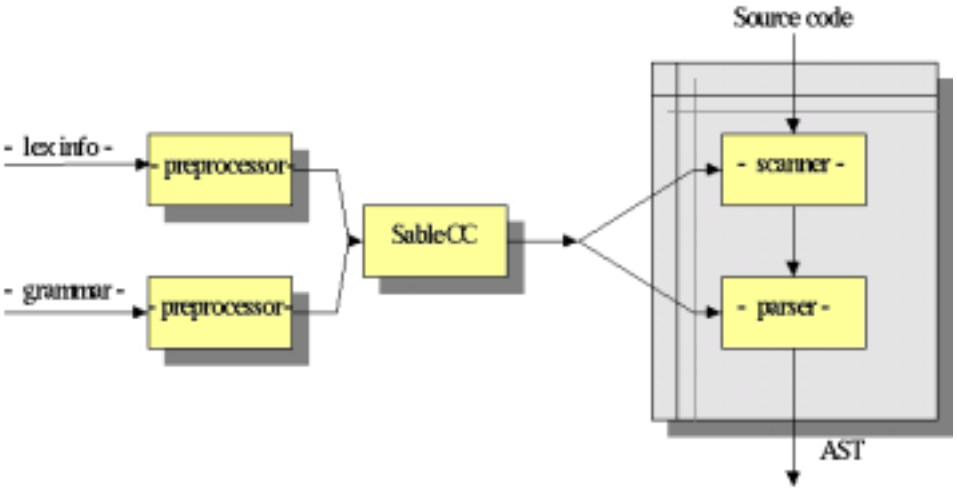

Figure 4-1. Preprocessor for lexical and grammar information

Figure 4-1 shows the pre-processor as two units. Why? The formats for both input files are so very similar that just one processor could have been enough. While the input files might be similar, the processing has its unique nuances. The first pre-processor reads in

the lexicon and writes the token information to a final SableCC config file. While the initial version of the TWS will not check errors to a very large extend. We have implemented a low-level error checker that verifies the correct input format as far as the standardized format is concerned.

The pre-processor for the grammar will parse in the grammar input file. The final SableCC config file does not allow anything in its production rules but pre-defined tokens. Eventually we will not follow this example. The users can write the TWS input grammar in the fashion they are familiar with (i.e. use terminals of the form “xxx”). However, in this release the respective terminal needs to be defined as such in the lexicon (e.g., `XXX -> ‘xxx’ => ‘<XXX>’`). The reason for this that there is no functionality in the grammar preprocessor to define and add tokens to the SableCC config file yet. In a later release this functionality will be added. We are aware that the present setup is confusing: “Why would I write “xxx” in the grammar file when XXX is already defined and I can use that, since it has fewer characters?” It is true that the production rules could be written entirely in terms of tokens. Please keep in mind that this setup leaves “the door open” for the above-mentioned functionality and makes its implementation easier.

SableCC takes its production rules in LALR(1) form only. And here is the main difference between the two processors: the one for the grammar needs to transform the potential non-LALR(1) grammar into LALR(1). It does that by reading in the input grammar and the tokens that have been defined and tries to create LALR(1)-conforming production rules, and writes them into the same SableCC config file that the tokens had been written to earlier already. The biggest source for the dreaded shift/reduce conflict are production rules that contain both terminals and nonterminals within the same rule, but not within the same production rule. Another type of problem are the ones similar to

the so called ***dangling else*** problem, where a final else statement cannot be uniquely assigned to a particular production rule. This is shown in the sample production rules in Figure 4-2:

```
Stmt    → expr
        → stmt ;
if_stmt → IF expr THEN stmt
        → IF expr THEN stmt ELSE stmt ;
expr    → <INT>
        → <STRING> ;
```

Figure 4-2. Sample production rules

These rules are not LALR(1) due to their ambiguous content. SableCC will report an error. If we had two IF's and only one ELSE statement, what IF-statement would it be assigned to?

In the pre-processor we will break up rules like this and create intermediate rules to avoid shift/reduce errors like they occurred here.

SableCC requires another piece of information to process the production rules. Each rule, regardless of whether a AST node will be defined or not, needs to be uniquely identifiable. Therefore the pre-processor needs to add this info to each rule since the original input syntax did not contain this. In this release we will enumerate the rules and concatenate those rule names that do not have a node name defined for them with the rule's respective number. Figure 4-3 shows an example:

```
foo     →     ABC DEF
        →     CBA FED     ⇒     foobar
        →     EOF    ;
```

Figure 4-3. Another sample set of production rules

This would be transformed into LALR(1) style notation and adjusted to SableCC's needs as shown in Figure 4-4:

```
foo   →   {foo1}     ABC DEF
      →   {foobar}   CBA FED
      →   {foo3}     EOF    ;
```

Figure 4-4. Adjusted set of production rules

In this example it was assumed that the input grammar was indeed LALR(1) conform. Its purpose was to show the re-write aspect of the input grammar vs. SableCC grammar.

Execution code file

Now we get into the area of the code generator. That is basically what the application of execution rules depending on a respective tree node represents. The (very) basic structure of this processor is very similar to the previous one. The main difference is that the processor for the grammar does not need any feedback from the parser tool. The lexical and production rule names are defined in the input files and passed on the lexer/parser tools, which take those names and use them for further processing. What is different here is that the lexer/parser tool redefines names for the parsed tree nodes. These are unknown to the input execution rule file, yet must be made known to it so that these rules can be matched with the respective tree nodes.

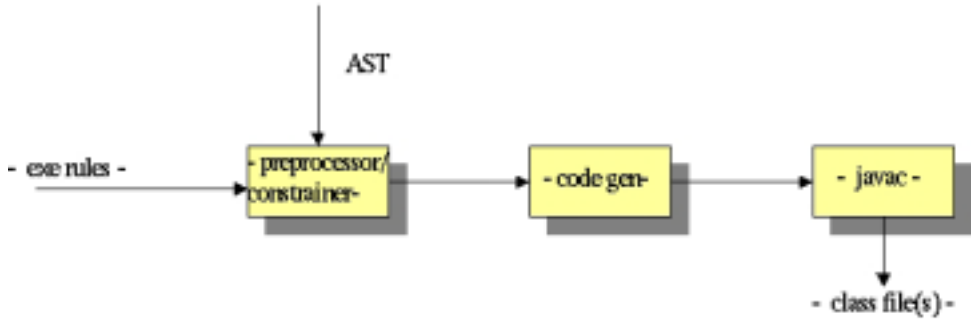

Figure 4-5. Preprocessor for execution rule information

Figure 4-5 shows the process of implementing the AST into the code generator. With a pre-defined naming scheme the pre-processor will correlate given user methods with ***codegen*** required method names. The code will be created in the following fashion: A class file needs to be written that contains the code that the user wants to be executed. Methods and variables relate to node names and leave names defined in the grammar in a standardized fashion. Each node will be represented by an object with a preset number of available methods.

The specifics of the naming and method structure (e.g., return types and such) will be refined in the actual program and can be looked up in its documentation later on. The user will be able to add any other desired methods/variables. They will be taken over into the TWS “as-is”.

Since the object code will be compiled with Sun’s java compiler ***javac*** the methods need to be written in java. The basic structure of the grammar file is a big switch/case statement. Upon the presence of a certain state within the AST a respective code segment need to be applied.

New system design

The above mentioned specifications can be shown as a whole in a general overview like shown in Figure 4-6:

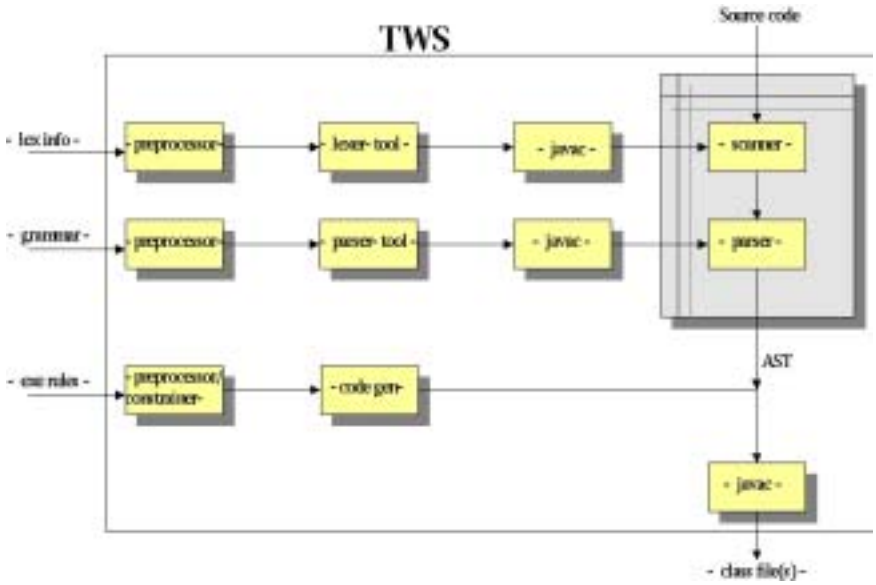

Figure 4-6. New system overview (general)

With SablaCC as the tool of choice the specific setup looks as shown in Figure 4-7:

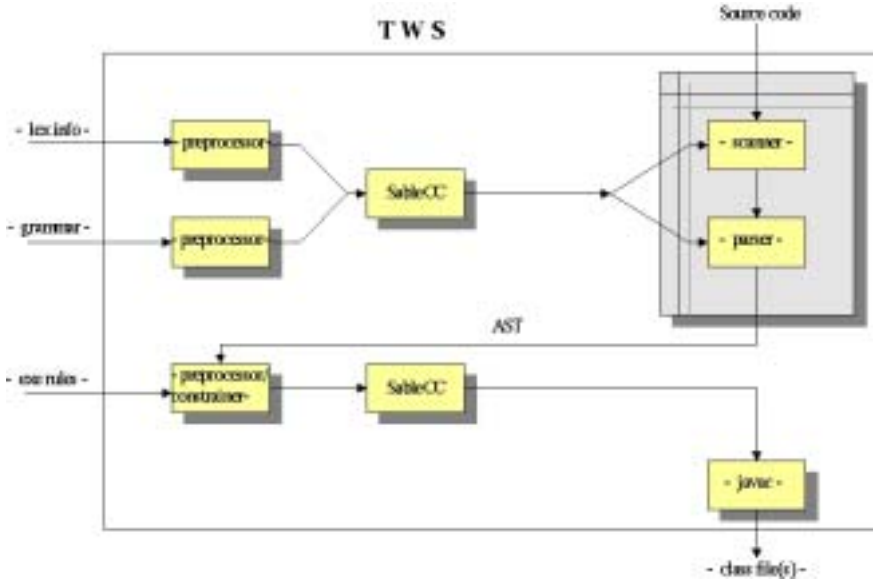

Figure 4-7. New System overview (tool specific)

Naturally this setup will change with every tool that is being used.

The future will show that certain design items will proof advantageous for the implementation of one tool and difficult with respect to the implementation of another tool. This is the nature of the beast. I tried to decide on certain design issues in a way that the user of the TWS has a structured input and ease of use when dealing with the TWS. This might result in sometimes not perfectly “slick” of “most efficient” coding. My primary concern was to find a good middle between total efficiency and maintainability. This program will for sure expand in its functionality over the years. Its place is in the academic environment and not the industry.

APPENDIX A
TWS INPUT FILES: RPAL

```
RPAL LEXICON (TWS.RPAL.LEXICON)
================================

//RPAL LEXICON:
//------------

ht         -> '\t' ;
eol        -> '\r' | '\n' ;

identifier -> letter (letter | digit | '_')*       => '<identifier>';

integer    -> digit+                               => '<integer>';

operator   -> operator_symbol+                     => '<operator>';

string     -> '"'
              ( '\' 't' | '\' 'n' | '\' '\' | '\' '"' |
                '('     | ')'     | ';'     | ','     |
                ' '     | letter | digit | operator_symbol )*
              '"'                                  => '<string>';

spaces     -> ( ' ' | ht | eol )+                  => '<delete>';

comment    -> '//'
            ( '"' | '(' | ')' | ';' | ',' | '\' | ' '
              | ht | letter | digit | operator_symbol
            )* eol                                 => '<delete>';

punction   -> '('                                  => '<lpar>'
           -> ')'                                  => '<rpar>'
           -> ';'                                  => '<semicolon>'
           -> ','                                  => '<comma>';

letter     -> 'A'..'Z' | 'a'..'z';

digit      -> '0'..'9';

operator_symbol
           -> '+' | '-' | '*' | '<' | '>' | '&' | '.'
            | '@' | '/' | ':' | '=' | '~' | '|' | '$'
            | '!' | '#' | '%' | '^' | '_' | '[' | ']'
            | '{' | '}' | '"' | '`' | '?';

let         -> 'let'                               => '<let>'      ;
in          -> 'in'                                => '<in>'       ;
fn          -> 'fn'                                => '<fn>'       ;
dot         -> '.'                                 => '<dot>'      ;
where       -> 'where'                             => '<where>'    ;
```

```
within      -> 'within'                          => '<within>'   ;
equals      -> '='                               => '<equals>'   ;
rec         -> 'rec'                             => '<rec>'      ;
aug         -> 'aug'                             => '<aug>'      ;
pipe        -> '->'                              => '<pipe>'     ;
or          -> '|' | 'or'                        => '<or>'       ;
and         -> '&' | 'and'                       => '<and>'      ;
not         -> '!' | 'not'                       => '<not>'      ;
gr          -> 'gr' | '>'                        => '<gr>'       ;
ge          -> 'ge' | '>='                       => '<ge>'       ;
ls          -> 'ls' | '<'                        => '<ls>'       ;
le          -> 'le' | '<='                       => '<le>'       ;
eq          -> 'eq'                              => '<eq>'       ;
ne          -> 'ne'                              => '<ne>'       ;
true        -> 'true'                            => '<true>'     ;
false       -> 'false'                           => '<false>'    ;
nil         -> 'nil'                             => '<nil>'      ;
dummy       -> 'dummy'                           => '<dummy>'    ;
plus        -> '+'                               => '<plus>'     ;
minus       -> '-' | 'neg'                       => '<minus>'    ;
times       -> '*'                               => '<times>'    ;
divideby    -> '/'                               => '<divideby>' ;
exp         -> '**'                              => '<exp>'      ;
att         -> '@'                               => '<att>'      ;
```

RPAL GRAMMAR (TWS.RPAL.GRAMMAR)
==========

```
//         RPAL Phrase Structure
//         ---------------------


rpal ->  e ;
e    -> 'let' d 'in' e                       => '<let>'
     -> 'fn'  vb+ 'dot' e                    => '<lambda>'
     ->  ew;

ew   -> t  'where' dr                        => '<where>'
     -> t;

//        # Tuple Expressions #######################################

t    -> ta ( 'comma' ta )+                   => '<tau>'
     -> ta ;

ta   -> ta 'aug' tc                          => '<aug>'
     -> tc ;

tc   -> b 'pipe' tc 'or' tc                  => '<pipe>'
     -> b ;

//        # Boolean Expressions #####################################

b    -> b 'or' bt                            => '<or>'
     -> bt ;
```

```
bt   -> bt 'and' bs                              => '<and>'
     -> bs ;

bs   -> 'not' bp                                 => '<not>'
     -> bp ;

bp   -> a rl a                                   => '<rl>'
     -> a ;

rl   -> 'gr'                                     => '<gr>'
     -> 'ge'                                     => '<ge>'
     -> 'ls'                                     => '<ls>'
     -> 'le'                                     => '<le>'
     -> 'eq'                                     => '<eq>'
     -> 'ne'                                     => '<ne>';

//        # Arithmetic Expressions ##################################

a    -> a 'plus' at                              => '<plus>'
     -> a 'minus' at                             => '<minus>'
     ->   'plus' at
     ->   'minus' at                             => '<neg>'
     -> at ;

at   -> at 'times' af                            => '<times>'
     -> at 'divideby' af                        => '<divideby>'
     -> af ;

af   -> ap 'exp' af                              => '<exp>'
     -> ap ;

ap   -> ap 'att' <identifier> r                  => '<at>'
     -> r ;

//        # Rators And Rands ########################################

r    -> r rn                                     => '<gamma>'
     -> rn ;

rn   -> <identifier>
     -> <integer>
     -> <string>
     -> 'true'                                   => '<true>'
     -> 'false'                                  => '<false>'
     -> 'nil'                                    => '<nil>'
     -> 'lpar' e 'rpar'
     -> 'dummy'                                  => '<dummy>' ;

//        # Definitions #############################################

d    -> da 'within' d                            => '<within>'
     -> da ;

da   -> dr ( 'and' dr )+                         => '<and>'
     -> dr ;
```

```
dr   -> 'rec' db                                => '<rec>'
     -> db ;

db   -> vl 'equals' e                           => '<equals>'
     -> <identifier> vb+ 'equals' e             => '<functionform>'
     -> 'lpar' d 'rpar' ;

//        # Variables ##############################################

vb   -> <identifier>
     -> 'lpar' vl 'rpar'
     -> 'lpar' 'rpar'                           => '<paren>';

vl   -> <identifier>
     -> <identifier> ('comma' <identifier>)*    => '<comma>';
```

APPENDIX B
TWS SABLECC PRE-PROCESSOR FILES

```
LEXICON PRE-PROCESSOR GRAMMAR (TWS.LEX.PREPROC.SABLECC)
=======================================================

Package tws.sablecc.preproc.lexicon ;



Helpers

    letter          = [['a'..'z']+['A'..'Z']]           ;
    digit           = ['0'..'9']                        ;
    ascii           = [32..127]                         ;
    list            = ' list '                          ;
    cr              = 13                                ;
    lf              = 10                                ;
    ht              =  9                                ;
    name            = letter (letter | digit | '_')*    ;
    s_quote         = 39                                ;


Tokens

    blank           = ht* | ' '* | cr* | lf*                          ;
    comment         = ( '/*' (ascii | ht | |cr | lf)* '*/' ) | ( '//'
                      (ascii | ht)* (cr | lf) ) ;

    identifier      = name                   ;
    tokenname       = ''' '<' name '>' '''   ;

    or              = '|'                                             ;
    assign          = '->'                                            ;
    define_as       = '=>'                                            ;
    terminal        = s_quote ascii ascii? s_quote                    ;
    keyword         = s_quote name s_quote                            ;
    charlist        = s_quote ascii s_quote '..' s_quote ascii s_quote;

    lpar            = '('               ;
    rpar            = ')'               ;
    multiplicity    = '+' | '*' | '?'   ;
    semicolon       = ';'               ;


Ignored Tokens

    blank   ,
    comment ;
```

```
Productions

    lexicon             = {lexicon}     word+                        ;

    word                = {word}      identifier definition+ semicolon;

    definition          = {definition}  assign association+          ;

    association         = {association} entry token_definition?      ;

    token_definition    = {token}       define_as tokenname          ;

    entry               = {entry_id}    identifier       more_entry   |
                          {entry_term}  terminal         more_entry   |
                          {entry_key}   keyword          more_entry   |
                          {entry_list}  charlist         more_entry   |
                          {entry_par}   lpar entry+ rpar more_entry   ;

    more_entry          = {morefull}    multiplicity or entry        |
                          {moremult}    multiplicity                 |
                          {moreor}      or entry                     |
                          {morenull}                                 ;
```

GRAMMAR PRE-PROCESSOR GRAMMAR (TWS_GRAMMAR_PREPROC.SABLECC)
==========

```
Package tws.sablecc.preproc.grammar ;


Helpers

    letter          = [['a'..'z']+['A'..'Z']]                       ;
    digit           = ['0'..'9']                                    ;
    ascii           = [32..127]                                     ;
    list            = ' list '                                      ;
    cr              = 13                                            ;
    lf              = 10                                            ;
    ht              =  9                                            ;
    name            = letter (letter | digit | '_')*                ;
    s_quote         = 39                                            ;
    key             = [[33..47]-39] | [60..62] | 64 | 91 | 93 |
                      [123..125] | '->' | '>=' | '<=' | '==' | '++' |
                      '!=' | '--' | '**' ;   // freq used symbols


Tokens

    blank           = ht* | ' '* | cr* | lf*                        ;
    comment         = ( '/*' (ascii | ht | cr | lf)* '*/' ) | ( '//'
                      (ascii | ht)* (cr | lf) ) ;

    rulename        = name                                          ;
    nodename        = ''' '<' name '>' '''                          ;
    def_token       = '<' name '>'                                  ;
```

```
    keyword         = s_quote (name | key) s_quote                  ;

    or              = '|'                                           ;
    assign          = '->'                                          ;
    pipe            = '->'                                          ;
    define_as       = '=>'                                          ;

    lpar            = '('               ;
    rpar            = ')'               ;
    multiplicity    = '+' | '*' | '?'   ;
    optional        = '?'               ;
    semicolon       = ';'               ;


Ignored Tokens

    blank   ,
    comment ;


Productions

    grammar             = {grammar}     rule+                       ;

    rule                = {rule}        rulename definition+ semicolon;

    definition          = {definition}  assign entry+ node_definition?;

    node_definition     = {node}        define_as nodename          ;

    entry               = {entry_rul}   rulename         more_entry  |
                          {entry_key}   keyword          more_entry  |
                          {entry_tok}   def_token        more_entry  |
                          {entry_par}   lpar entry+ rpar more_entry  ;

    more_entry          = {morefull}    multiplicity or entry       |
                          {moremult}    multiplicity                |
                          {moreor}      or entry                    |
                          {morenull}                                ;
```

LIST OF REFERENCES

[1]: Scott M., Programming Language Pragmatics, Morgan Kaufmann Publishers, San Francisco, CA, 2000

[2]: O'Connor J., Robertson E., John Backus, http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Backus.html, 1996 Last access: April 10, 2003

[3]: Wikipedia, Peter Naur, http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Backus.html, 2002 Last access: April 10, 2003

[4]: Estier T., About BNF Notation, http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html, 1998 Last access: April 10, 2003

[5]: Marcotty M., Ledgard H., The World Of Programming Languages, Axel Springer Verlag, 1986

[6]: Bermudez M., COP4620 Translators, Unpublished Course Material, University of Florida, Gainesville, FL, Summer Semester 2002

[7]: Lemone K., Programming Language Translation, http://cs.wpi.edu/~kal/PLT/, 1997 Last access: April 10, 2003

[8]: Bermudez M., COP5555 Programming Languages Principles - Declaration Module, http://www.cise.ufl.edu/class/cop5555su02/Constrainer.htm, 2002 Last access: April 10, 2003

[9]: Fraunhofer Institute for Computer Architecture and Software Technology, Catalog of Compiler Construction Tools, German National Research Center for Information Technology, http://catalog.compilertools.net, 2002 Last access: April 10, 2003

[10]: Donnelly C., Stallman R., Bison--The YACC-compatible Parser Generator, http://dinosaur.compilertools.net/bison/index.html, 1995 Last access: April 10, 2003

[11]: Jamison B., BYACC/J--Java extension , http://troi.lincom-asg.com/~rjamison/byacc/, 2001 Last access: April 10, 2003

[12]: Paxson V., Flex--A fast scanner generator,
http://dinosaur.compilertools.net/flex/index.html, 1995
Last access: April 10, 2003

[13]: Johnson S., Yacc: Yet Another Compiler-Compiler,
http://dinosaur.compilertools.net/yacc/index.html, unknown
Last access: April 10, 2003

[14]: Schreiner A.T., Kuehl B., jay--A YACC For Java,
http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay, 1999
Last access: April 10, 2003

[15]: WebGain Inc., Java Compiler Compiler (JavaCC)--The Java Parser Generator,
http://www.webgain.com/products/java_cc, 2002
Last access: April 10, 2003

[16]: Won D., JavaCC Grammar Repository, http://www.cobase.cs.ucla.edu/pub/javacc, 2002
Last access: April 10, 2003

[17]: Hudson S., Cup--LALR Parser Generator For Java,
http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.redirect.html, 1996
Last access: April 10, 2003

[18]: Klein G., Jflex--The Fast Scanner Generator for Java, http://www.jflex.de/, 2003
Last access: April 10, 2003

[19]: Gagnon E., SableCC version 2.16.2, http://www.sablecc.org/, 2002
Last access: April 10, 2003

[20]: Gagnon E., SableCC--An Object Oriented Compiler Framework,
http://www.sablecc.org/thesis/thesis.html - PAGE21, 1998
Last access: April 10, 2003

BIOGRAPHICAL SKETCH

After graduating from the German high school Staatliches Martinus Gymnasium in Linz on the Rhine river in Germany in June 1985, Hans-Georg Lerdo joined the German Navy. Here he served as a Tactical Coordinator/Mission Commander on board maritime patrol aircraft. After 13 years of active duty his term ended and along with his family he moved to Gainesville, Florida, in August of 1998. There he enrolled in the computer engineering program in the CISE department at the University of Florida. He graduated with the degree of Bachelor of Science in computer engineering in August of 2002 and plans on graduating with the degree of Master of Science in computer engineering in May 2003.