Spring 5-1-2017

Enhancing an Extensible Interpreter with a Syntax Macro Facility

Xin Wan
Arkansas Tech University

ENHANCING AN EXTENSIBLE INTERPRETER WITH A SYNTAX MACRO FACILITY

By

XIN WAN

Submitted to the Faculty of the Graduate College of
Arkansas Tech University
in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE IN MS INFORMATION TECHNOLOGY
May & 2017

Permission

Title: Enhancing an Extensible Interpreter with a Syntax Macro Facility

Program: Information Technology

Degree: Master of Science

Signature

Date

Acknowledgements

I would first like to thank my thesis advisor Dr. Larry Morell of Computer and Information Science at Arkansas Tech University. He gave me lots of guidance whenever I ran into a trouble spot or had a question about my research or writing. He definitely provided me with the tools that I needed and steered me in the right direction.

I would also like to acknowledge Dr. David Middleton of the Computer and Information Science Department at Arkansas Tech University. He taught me lots of knowledge related to programming language and compiler design so that I can finish my thesis.

I would also like to thank the expert Dr. Jerry Wood of the Computer and Information Science Department at Arkansas Tech University as my thesis committee member. Without his passionate participation and support, the whole thesis could not have been successfully conducted.

Finally, I must express my very profound gratitude to my husband and my children for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Author

Xin Wan

Abstract

This thesis builds a syntax macro facility for an extensible interpreter, which enables the developer to extend the base language at run time, without knowing the details of the interpreter. In the enhanced extensible interpreter, each grammar rule is represented by a distinct class that inherits from `Instruction` class, which contains all the information necessary for scanning, parsing, and interpreting the corresponding construct. The macro facility is implemented by a special class `Macro`, which also inherits from `Instruction` class. Each macro rule is associated with a unique `Macro` instance. With the new extensible interpreter strategy, the syntax macro facility does not require special format for macro invocation. The macro grammar rule and the base grammar rule share similar syntax, which means that there is no difference between extended language and base language from the user's view. The process of macro invocation is almost the same as a base instruction invocation, except that expansion occurs before evaluation.

Key Words: syntax macro; interpreter design; extensible interpreter

Table of Contents

List of Tables

List of Figures

1. Introduction

This thesis introduces an extensible interpreter with a syntax macro facility, which enables the developer to extend the base of a programming language at run time, without knowing the details of the interpreter.

A traditional design of an interpreter consists of three main components: a scanner, a parser and an evaluator. Each grammar rule of the language must be split into three parts and be added into different places, which makes it hard to maintain. Moreover, as all the grammar rules have to be built and integrated into an interpreter, it is difficult to change the interpreter once it's built.

A new design [3] has been introduced to solve the problem caused by splitting the implementation of an interpreter into three components, by creating a distinct class for each grammar rule and building the grammar module at parsing time with a rule registration strategy. The new design makes the implementation extensible by adding a single class for each additional grammar rule. However, this requires the person doing the extension to understand the underlying structure of the interpreter and the language used to implement the interpreter.

To solve this problem, this thesis introduces a syntax macro to extend a language without knowing the details of the interpreter.

When invoking the macro statements, most languages require a special syntactic format and an additional strategy for macro invocation [6]. For instance, in Dylan [14], each macro rule has two basic parts: a pattern that is matched against a fragment of code, and a template that is substituted for the matched fragment. When a macro is invoked, each macro rule in the macro rule set is tried in order until a matching pattern is found.

As the time complexity to match the pattern is O(n), where n is the number of macro rules, if the macro rule set is really large, macro invocation will be slow.
With the new extensible interpreter strategy, the syntax macro facility introduced in this thesis does not require a special format for macro invocation. The macro grammar rule and the base grammar rule share the same syntax.

In the following sections, we will first review the related backgrounds of interpreters, the improved interpreter with extensible capacity, and syntax macros in programming languages. Second, we present our design of a syntax macro facility in an extensible interpreter. Then, we test our design by building a new language, adding macro rules and executing the macro statements. The conclusion reviews the features of this design and point to further work.

2. Literature Review

When developing a programming language, there are two important aspects to consider:

Programming Language Specification – The programming language specification defines the language, which is typically detailed and formal with some notations like syntax (structure) and semantics (meaning) of the language, and is primarily used in programming language implementation.

Programming Language Implementation – The programming language implementation uses the specifications to execute the programming language on one or more configurations of hardware and software.

The first two parts of this section provide an overview of the implementation approaches, some key components, and general techniques of programming language specification. Then, an improved interpreter with extensible capacity is presented. Lastly, syntax macros in programming languages are discussed.

2.1 Compiler vs. Interpreter

There are two main approaches to programming language implementation: compiler and interpreter.

2.1.1 Concepts of compiler and interpreter

A compiler, shown in Figure 2.1, is a program that transforms source code written in a human-oriented programming language (the source language) into a computer-oriented machine language (the target language). It translates the source code from a high-level language to a lower-level language.

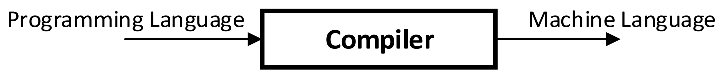


Figure 2.1 The definition of compiler

The input of a compiler is always a specific high-level programming language. Compared with a low-level language, a high-level programming language is machine-independent and easier to use, therefore, allowing users to ignore the details of the hardware and thereby making the software development simpler.

The output of a compiler is generally machine code. The code is executed by the Central Processing Unit (CPU) of a specific machine in a process called the *fetch-decode-execute cycle*. The machine code for one CPU may be different from a high-level language, depending on the amount of abstraction between target language and machine language [1]:

Pure Machine Code – The instruction set of target code belongs to a particular machine's instruction set. This type of target code could be executed on bare hardware independently from any other software. It is sometimes used to define operating systems or embedded applications like a programmable controller.

Augmented Machine Code – This form of target code is generated for a machine augmented with operating system (OS) routines and language support routines (I/O, storage allocation, mathematical functions, etc.). Machine instructions together with OS and run-time routines form a virtual machine.

Virtual Machine Code – The target code only includes virtual instructions without any machine's native instructions. It allows the code to run on different operating systems without having to be rewritten or recompiled.

Java is one of the best-known examples that uses virtual machine approach. A Java source program is first compiled into bytecode for Java virtual machine. As virtual machine code, the bytecode is platform-independent and can then be sent to any platform and run on that platform by using Java Virtual Machine. To execute a compiled Java program, the Java virtual machine interprets the bytecode into instructions understandable by the particular processor.

An interpreter, described in Figure 2.2, is a program that directly executes the instructions written in a programming language.

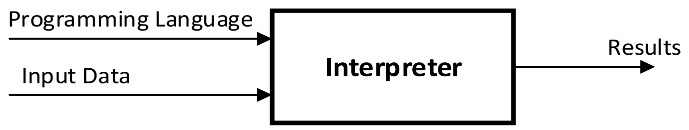


Figure 2.2 The definition of interpreter

Unlike a compiler which translates the source code into target code for later execution on the corresponding machine (specified by the target code), the interpreter translates the high-level program into an intermediate form and then simulates the fetch-decode-process cycle in software and directly executes the high-level language instead of machine code.

2.1.2 Differences between compiler and interpreter

The main difference between a compiler and interpreter is that once the program has been compiled, it could run anytime, whereas an interpreted program needs to be interpreted line-by-line every time it runs. The difference seems minimal, but leads to many different features.

A compiled program runs faster and is more efficient. One reason is that it gets converted into machine code once before execution. Another reason is that an interpreter may analyze the same statements within a loop over and over again, whereas a compiler only needs to translate them once.

Debugging is easier for an interpreted language. As a compiler takes the entire program as input, it reports the errors after checking the whole program. One error might produce many spurious errors. On the other hand, when an interpreter encounters an error, it immediately reports it to the user so that the programmer can locate the error easily.

2.1.3 Structures of compiler and interpreter

Figure 2.3 shows the structure of a typical compiler.

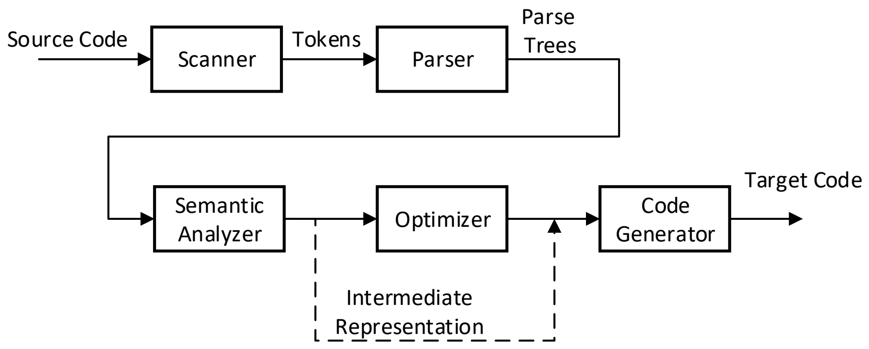


Figure 2.3 The structure of a typical compiler

There are several phases/components in a typical compiler:

Scanner – A scanner is a lexical analyzer. It scans the source code as a stream of characters and converts the letters into meaningful words (tokens).

Parser – A parser is a syntax analyzer. It takes the tokens generated by scanner as input, and groups them into syntactic structure which is often in the form of syntax tree (or parse tree).

Semantic Analyzer – A semantic analyzer checks for the static semantics of each syntactic structure. If the structure is semantically correct, it produces an intermediate level between source code and target code, usually called an Intermediate Representation (IR).

Code Generator – A code generator maps the IR into the target machine code by code generator.

A simple interpreter parses the source code and performs its behavior directly, which is called pure interpretation. Some low-level languages use this approach for program execution.

More often, for high-level languages, an interpreter (Figure 2.4) first translates the source code into a parse tree or some efficient IR code and then executes afterward.

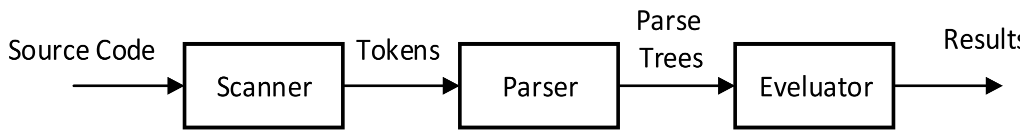


Figure 2.4 The structure of a typical interpreter

Thus, both compilers and interpreters share some of the same components as a scanner and parser. In a sense, an interpreter implicitly includes a compiler that translates the source code into a certain type of code (IR code or parse tree). Moreover, an interpreter needs one more phase to perform the actions described by the source code, which is called an evaluator.

Evaluator – An evaluator interprets the syntax structures (the IR) into their meanings. Usually each statement will be associated with one or more semantic routines that produce the meaning of the statement.

2.2 Implementation Methods of the Typical Interpreter

A typical interpreter includes three main phases: lexical analysis, syntax analysis and semantic evaluation. In this part, we will explain each phase in detail and discuss some common methods to specify the language in those phases.

2.2.1 Lexical Analysis

There are two primary approaches to implementing a scanner (lexical analyzer). The first is to write a scanner by hand. However, the hand-coded scanner may be complex and difficult to maintain, extend or reuse.

The second is to use regular expressions to specify tokens and finite automata to recognize tokens from input characters. Moreover, the second method is useful to program a scanner generator.

Regular Expressions

A regular expression is a sequence of characters that defines a pattern that a set of strings satisfy (regular set). Regular expressions specify the tokens that make up an input program. The set of regular expressions are defined as follows:

- $\emptyset$ is a regular expression, denoting the empty set.
- λ is a regular expression, denoting the set that only consists of an empty string.
- If A and B are regular expressions, then AB, $A|B$ and A^* are also regular expressions, denoting three standard regular operators of *catenation*, *alternation* and *Kleene closure* (A repeats 0 or more times).
- If A is a regular expression, then
  - A^+ denotes that A repeats 1 or more times, which can be obtained by using standard regular operators: $A^+ = AA^*$.

Regular expressions can be used to define the tokens of a programming language. Here are some examples.

Let:

$D = [0-9], L = [a-zA-Z]$(These are the shorthand to represent continuous character sets)

Then

- An *identifier*, that consists of letters, digits and the underscores, begins with a letter or an underscore, can be defined as:

  $ID = (L|\_)(L|D|\_)^*$

- A *number* can be defined as:

  $NUM = (+|-|\lambda)D^+$

Finite Automaton (FA)

A Finite Automaton is a simple idealized machine used to recognize strings that belong to regular sets. When a finite automaton processes a string of symbols, it changes the state based on each symbol. If the input string reaches the final state, it is accepted, which means a valid token has been found/verified. There are two types of FA:

Deterministic Finite Automaton (DFA) – For each transition (with a given state and character), only one state can be reached. We can define a DFA in as follow:

- A finite set of *states* (Q)
- A finite set of input characters, or *vocabulary* (V)
- A special *start state* ($q_0 \in Q$)
- A set of *final*, or *accepting*, *states* ($F \subseteq Q$)

- A *transition function* ($\delta: Q \times V \rightarrow Q$), that maps one state to another, labeled with input characters in *vocabulary* (V)

A DFA begins with the start state. For every input character in the sequence, it computes the next state with the transition function associated with the current character and current state. Each of the transitions in a DFA is uniquely determined by its current state and input symbol. Therefore, it jumps deterministically from one state to another until reach the final state or there are no available transitions.

Nondeterministic finite automaton (NFA) – For some state and input character, the next state may be nothing or one or more possible states. An NFA consists of:

- A finite set of *states* (Q)
- A finite set of input characters, or *vocabulary* (V)
- A special *start state* ($q_0 \in Q$)
- A set of *final*, or *accepting*, *states* ($F \subseteq Q$)
- A *transition function* ($\Delta: Q \times V \rightarrow P(Q)$), $P(Q)$ denotes the power set of Q

In a NFA, for each state there can be zero, one or more transitions corresponding to a particular character. So, if a NFA gets to a state with more than one possible transitions corresponding to the particular input character, it will be hard to determine the next state.

A DFA is more efficiently executable than NFA because each input maps to a unique successor state and is therefore often used to drive a scanner. Compared to DFA, a NFA is easier to construct but less efficient to execute. There is a standard method for converting an NFA into an equivalent DFA. Hence, we will only discuss DFA here.

As noted above, a DFA can be represented graphically using *state transition diagrams*, with nodes for states, and arcs for transitions that are labeled with input characters. Another way to specify a DFA is to use *transition table* T, which can be conveniently used by the computer program.

For example, given a regular expression that defines the *identifier*

$$ID = (L|\_)(L|D|\_)^*$$

The corresponding DFA state transition diagram lies:

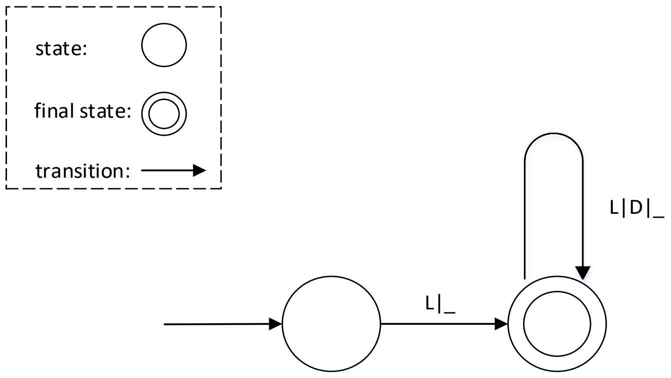


Figure 2.5 Example of state transition diagram

The corresponding transition table T is shown in Table 2.1:

Table 2.1 Transition table

State	Character Sets		
	L	D	_
1	2		2
2	2	2	2

Then, the scanner works in the following way:

- Starting from the start state 1.

- With the current state s and the next character c in the input string, then T[s][c] specifies the next state or an error (if empty, which can be implemented by dump state).
- When the scanner reaches the final state, then a token has been found successfully.

2.2.2 Syntax Analysis

Syntax analysis or parsing is the second phase of an interpreter to check that if we have a valid sequence of tokens and then generates a certain hierarchical structure, such as parse tree, abstract syntax tree, or the like. In this phase, a context-free grammar (CFG) is used to specify the grammar rule.

Context-free Grammar (CFG)

A context-free grammar is defined by 4-tuple ($G = (V_t, V_n, S, P)$) described as follow:

- A finite set of terminals V_t. This is the token sets produced by the scanner.
- A finite set of non-terminals V_n. They are symbols that define the language generated by the grammar.
- A start symbol or goal symbol $S \in V_n$ that starts all derivations (defined below).
- A set of productions P in the form of $A \rightarrow X_1 \cdots X_m$, where $A \in V_n$, $X_i \in V_t \cup V_n$, $1 \leq i \leq m$, $m \geq 0$.

Having the definitions above, there are a few more notations:

Derivation – It is a sequence of rewriting steps to generate the string that only consists of terminals defined by the CFG from the start symbol:

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w,$$

where

- S is the start symbol of the CFG.

- $w \in V_t^*$ is a sentence of the CFG.
- γ_i $(0 \leq i \leq n)$ is a sentential form of the CFG.
- $\Rightarrow$ denotes a one-step derivation, which means only one non-terminal is rewritten to its corresponding right-hand side from the previous sentential form to next sentential form.

At each step, during parsing, we need to decide which non-terminal is replaced and what production is applied. There are two standard rules to choose the non-terminal:

Leftmost Derivation – The leftmost non-terminal is always replaced first.

Rightmost Derivation – The rightmost non-terminal is always replaced first.

Here is an example to show the process of leftmost derivation and rightmost derivation. Consider the grammar of a simple arithmetic expression shown in Figure 2.6.

```
<exp> → <exp> + <term> | <exp> - <term> | <term>
<term> → <term> * <factor> | <term> / <factor> | <factor>
<factor> → ( <exp> ) | <no>
```

Figure **Error! No text of specified style in document.**.6 A grammar of a simple arithmetic express

The leftmost derivation and rightmost derivation of sentence “(1+2)/3” are described in Figure 2.7.

Leftmost derivation	Rightmost derivation
<exp>⇒<term>	<exp>⇒<term>
⇒<term>/<factor>	⇒<term>/<factor>
⇒<factor>/<factor>	⇒<term>/<no>
⇒(<exp>)/<factor>	⇒<term>/3
⇒(<exp> + <term>)/<factor>	⇒<factor>/3
⇒(<term> + <term>)/<factor>	⇒(<exp>)/3
⇒(<factor> + <term>)/<factor>	⇒(<exp> + <term>)/3
⇒(<no> + <term>)/<factor>	⇒(<exp> + <factor>)/3
⇒(1 + <term>)/<factor>	⇒(<exp> + <no>)/3
⇒(1 + <factor>)/<factor>	⇒(<exp> + 2)/3
⇒(1 + <no>)/<factor>	⇒(<term> + 2)/3
⇒(1 + 2)/<factor>	⇒(<factor> + 2)/3
⇒(1 + 2)/<no>	⇒(<no> + 2)/3
⇒(1 + 2)/3	⇒(1 + 2)/3

Figure 2.7 Leftmost derivation and rightmost derivation

Leftmost derivation and rightmost derivation lead to two different parsing techniques (top-down and bottom-up) as discussed below.

Top-down Parsing

Top-Down Parsing starts at a start symbol (the root of the parse tree) and expands toward leaves (input sequence of tokens), attempting to find the left-most derivation to rewrite the input by hypothesizing general parse tree structures and trying to match the input with the known fundamental structures. It generates preorder traversal of parse tree.

Recursive-descent parsing - Recursive-descent parsing is a well-known top-down parsing technique. Each non-terminal is associated with a parsing procedure which can recognize token sequences generated by that non-terminal. As the productions of the grammar could be recursive, those associated procedures may be called recursively, and that's why it's called "recursive". "Descent" indicates that the parsing proceeds is in a top-down fashion.

The key problem when parsing is to determine which production to match for each step. Backtracking is a general way to find out the correct production by trying each possible production in turn. However, it might be quite time consuming. To avoid backtracking, a class of grammar known as LL(k) grammars use *k* tokens of look-ahead to predict the correct production.

LL(1) Grammars – LL(1) parses the input from **L**eft to right, and constructs a **L**eftmost derivation of the sentence by looking ahead **one** token to determine the current production. To make the prediction available, one constraint on LL(1) grammars is that the predict sets (e.g., the set of possible tokens that may begin a sentential form derivable from a non-terminal) for productions sharing the same left-hand side must be disjointed.

For a production $A \to X_1 \cdots X_m$, where $A \in V_n,\ X_i \in V_t \cup V_n,\ 1 \le i \le m, m \ge 0$, the predict set could be described as:

$$\begin{aligned}\text{Predict}(A \to X_1 \cdots X_m) = {} & if\ \lambda \in \text{First}(X_1 \cdots X_m) \\ & then\ (\text{First}(X_1 \cdots X_m) - \lambda) \cup \text{Follow(A)} \\ & else\ \text{First}(X_1 \cdots X_m).\end{aligned}$$

where,

$$\text{First}(X_1 \cdots X_m) = \{a \in V_t | X_1 \cdots X_m \Rightarrow^* a\beta\}$$

$(if\ X_1 \cdots X_m \Rightarrow^* \lambda\ then\ \{\lambda\}\ else\ \phi)$, denoting the set of all the terminals that can begin a sentential form derived from *A*.

$\text{Follow}(A) = \{a \in V_t | \text{S} \Rightarrow^* \cdots Aa \cdots\} \cup (if\ \text{S} \Rightarrow^* \alpha A\ then\ \{\lambda\}\ else\ \phi)$, denoting the set of all the terminals that may follow A in some sentential form.

The predict sets can be easily represented in the form of *parse table* T as follows:

T: $V_n \times V_t \rightarrow P \cup \{Error\}$

$if\ t \in$ Predict$(A \rightarrow X_1 \cdots X_m)$ $then$ T$[A][t] = A \rightarrow X_1 \cdots X_m$

else T$[A][t] = Error$.

Then, we could build the parser based on LL(1) parse table. For each step, the method for deciding the production is:

- Find the leftmost non-terminal A from the sentential form.
- Look at the next unmatched token t from the input token sequence.
- Choose the production indicated by T$[A][t]$.

Bottom-Up Parsing

Bottom-Up Parsing, in reverse, begins at the bottom (the leaves of the tree which represent the input stream of tokens) and attempts to work back to the starting symbol, by finding a right-most derivation backwards.

Shift-Reduce Parsing – Shift-Reduce Parsing is a commonly used bottom-up parsing method, without guessing or backtracking. The shift-reduce parser uses a parse stack, and the two main actions, *shift* and *reduce*. The shift action pushes the next token from input token sequence to the stack. The reduce action replaces the right-hand side on top of the stack by its corresponding left-hand side. The key issue is to find the specific right-hand side, the handle, so that the rightmost derivation could be traced to the root in reverse.

LR Parsers –These are also known as LR(k) parsers, where L stands for left-to-right scanning of the input; R stands for the construction of right-most derivation, and k denotes the number of lookahead symbols to make decisions. They are the most commonly used shift-reduce parsing approaches. The operation of LR parsers is controlled by two parse tables:

- ACTION Table: a table with rows indexed by states and columns indexed by terminals. For a shift process, if the parser is in certain state *s* (specified by the ACTION table) and the current lookahead terminal is *t*, then the action is specified by ACTION[s][t]. This table can contain four different kinds of entries:
    - Shift *s*: Shift the next token and next state *s* onto the parse stack.
    - Reduce r: replace right-hand side on top of the stack by its corresponding left-hand side of rule r.
    - Accept: the input has been parsed successfully.
    - Error: announce a parse error.
- GOTO Table: a table with rows indexed by states and columns indexed by nonterminals. For a reduce process, after right-hand sides on top of the stack is replaced by its corresponding left-hand side N, then the next state to enter is given by GOTO[s][N].

Parse Tree – A parse tree is a graphical description of a derivation. It shows how a sentence is derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Its leaves are terminals and interior nodes and non-terminals.

2.2.3 Semantic Evaluation

Unlike the scanner and the parser that have some general specifications and methods to describe and implement them, it is difficult to describe the meaning, also called dynamic semantics, of the language. Generally speaking, there are three major ways to specify the formal semantics [2].

Operational Semantics

The basic idea of the operational semantics is to describe the meaning of a program with a collection of effects obtained by running it on a hypothetical machine. Those effects are viewed as a set of transition relations (transitions between states, where a state is a collection of the values in the storage). In other words, the meaning of a statement of a language is specified by a set of statements of another language [2].

The issue for operational semantics is to find the appropriate level of the language to specify the meaning. As machine language implementations have irrelevant details and real computer implementations are too complicated, an intermediate language is suitable to define the operational semantics.

Different levels of uses of operational semantics can be categorized into two types. One type is called structural operational semantics (or small-step semantics), which describes how the individual steps of a computation take place in a computer-based system. Another type is named natural semantics (or big-step semantics), which describes how the overall results of the executions are obtained [2].

Denotational Semantics

Denotational semantics are concerned with constructing mathematical objects called denotations to describe the meanings of the language. It’s based on the recognition that each language entity can be mapped to an appropriate mathematical object, such as a number, or a function. For example, strings of digits are associated with numbers [2].

For example, given the expression 1+2, its denotational definition could be described as

$$[\![1+2]\!]=3,$$

where $[\![\quad]\!]$ represents meaning of the inside phrase, the value followed by = is the object associated with the phrase in $[\![\quad]\!]$.

One important principle of denotational semantics is that semantics should be compositional, which means the denotation of a language is determined by the denotations of its subphrases and the rules used to combine them. Therefore, the meaning of the phrase only depends on the meaning of its constituents:

$$[\![1+2]\!]=[\![1]\!]+[\![2]\!]$$

As a consequence of compositionality, a denotational specification parallels the syntactic structure of the parse tree.

Axiomatic Semantics

Axiomatic semantics use logical expressions called assertions (a statement that a predicate is expected to always be true at that point in the code) to describe the behavior of constructs in the language [2]. Unlike operational semantics and denotational semantics that correspond to the state of the machine, axiomatic semantics specify what can be proven about the program.

Each statement has the assertions before and after it, respectively called pre-condition and post-condition. And the meaning of the statement is specified by its pre-condition and post-condition, which describe what is true before a statement and after a statement.

2.3 An improved Interpreter with Extensible Capacity

As mentioned above, a traditional interpreter at least has three key components: a scanner, a parser and an evaluator. Generally speaking, those components are relatively independent and deal with different tasks.

However, the problems are when tailoring, extending or modifying the implementation of the language. For example, to add a new grammar rule (construct) to the current language, we may need to modify almost all the components, adding tokens to scanner, setting up parsing procedure in parser, and creating evaluation routine for the evaluator. Obviously, this traditional structure makes the system difficult to extend.

A new design in [3] is introduced to solve the problem, by means of creating a distinct class for each grammar rule. Each class contains the following:

- A definition of the grammar rule being represented.
- A registration routine that registers the rule along with its parser and an associated translator to be applied when instance of the rule is parsed.
- Declaration of the state needed to record the results of the translation.
- A routine for evaluating (interpreting) the construct.

Each class implements an Instruction class, containing its own grammar rule, parser and evaluator. The information and related methods for each class is in one place and the implementations of scanning, parsing and evaluating are associated with the specific global structures (represented by a trie) that link all the grammars and parsers respectively.

Instead of hardcoding all the grammar rules to the system, it begins with an empty grammar. The developer adds the needed grammar rules to the grammar by calling the registration routine `initialize()` for each desired grammar rule. During registration, each grammar rule in the grammar is associated with its own parser and evaluator.

When parsing, the main program invokes a parser associated with the start symbol, but does not need to know how the parser works. A parser for a rule may invoke different parsers for each member of the right-hand side without knowing how they work. When the rule is parsed, it collects a list of `Instruction` from its right-hand side one for each terminal and non-terminal. This list is passed to an object of the class associated with this rule, which stores the relevant parsed `Instruction` list, and returns a single Instruction with specified state (class). This `Instruction` then serves higher-level parsers, stored in higher-level Instruction lists, and so on.

For example, the grammar rules of a language that supports the arithmetic operations are listed as below:

```
  <Expression> ::= <Term>
  <Expression> ::= <Term> <AddOp> <Expression>
  <Term> ::= <Factor>
  <Term> ::= <Factor> <MultOp> <Term>
  <AddOp> ::= +
  <AddOp> ::= -
  <MultOp> ::= *
  <MultOp> ::= /
  <Factor> ::= ( <Expression> )
  <Factor> ::= <id>
  <Factor> ::= <no>
```

Each rule is associated with a class that represents the Instruction class. Following is an Instruction module for grammar rule "`<Expression> ::= <Term> <AddOp> <Expression>`".

```
public class ExpTermAddOpExp extends ELSEInstruction {

  // grammar rule defined as a String
  private static String rule = "        <Expression> ::= <Term> <AddOp>
<Expression>";
```

```
  // grammar rule encoded in SyntaxRule
  private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

  // static initialization that registers the syntax rule
  // with the grammar, creates an instance of this class
  // and creates the parser to parse any arithmetic
  // expression that fits this rule.
  public static void initialize() {
            GRAMMAR.addRule(syntaxRule,
                           new ExpTermAddOpExp(),
                           new GenericParser(syntaxRule.lhs()));
      }

  public ExpTermAddOpExp(){ }

  // default to building a full parse tree that records
  // the result list il and current state (class) ExpTermAddOpExp
  public Instruction createInstruction(InstructionList il) {
      ExpTermAddOpExp result = new ExpTermAddOpExp();
        result.il = il;
            return result;
  }

  // routine for evaluating (interpreting) the construct
  public Value eval(EnvironmentVal env) {
            Value a = il.get(0).eval(env);
       Value b = il.get(2).eval(env);
       if (a instanceof DoubleVal && b instanceof DoubleVal){
                  if (il.get(1) instanceof AddOpAdd)
                        return new DoubleVal(((DoubleVal)a).getVal() +
  ((DoubleVal)b).getVal());
            if (il.get(1) instanceof AddOpSub)
                        return new DoubleVal(((DoubleVal)a).getVal() -
  ((DoubleVal)b).getVal());
            }
            return null;
      }
}
```

The main program registers all the grammar rules by calling each associated `initialize`() routine, which registers the syntax rule with the grammar, and associates with it two objects: an instance of this class that represents the grammar rule, and the parser to be used to parse any statement that fits this rule.

After registration, the grammar holds all the rules and associated parsers, and it is ready to begin recursive descent parsing. The main program invokes the parser associated with the start symbol of the grammar, which, in turn invokes appropriate parsers registered with right-hand sides of rules as they are discovered. It recursively invokes the appropriate parsers until deriving the whole token sequence is derived. As a parser recognizes a maximal-length rule, it builds a list of `Instruction` created by calling lower-level parsers. This list is passed to the `createInstruction()` of the registered Instruction.

Here is an example to parse expression "(1+2)/3". The scanner first analyzes the input string and returns a token sequence: [(] [1] [+] [2] [)] [/] [3]. Then, the main program invokes the parser associated with "`<Expression>`", which is the start symbol. The whole parsing process is shown in Figure 2.8.

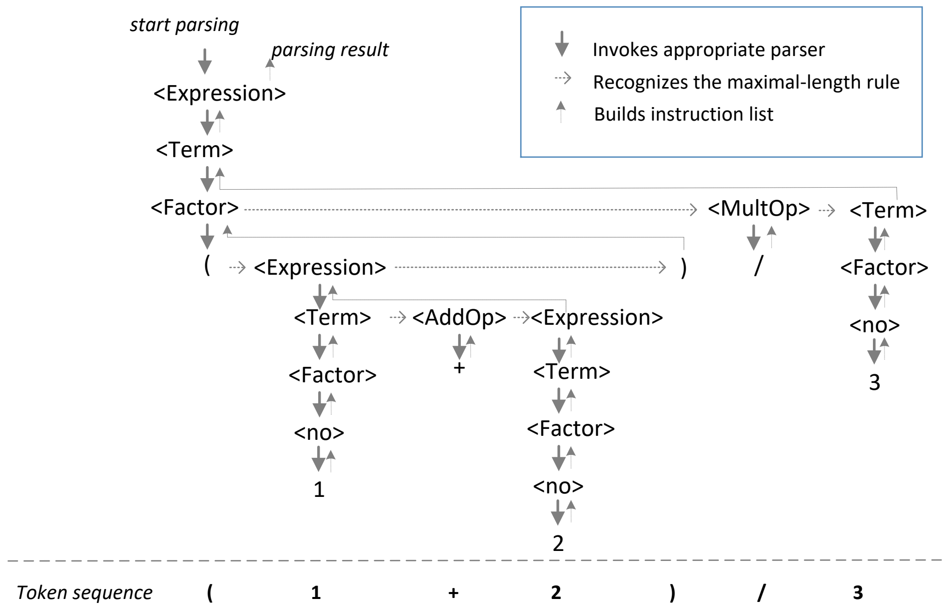


Figure 2.8 Parsing process of expression "1+2/3"

During parsing, there are three kinds of operations: invoking appropriate parser, recognizing the maximal-length rule and building an `Instruction` list. For instance, when matching token "(", the main program invokes the parser associated with "`<Expression>`", which in turn invokes the parser associated with "`<Term>`", which, in turn invokes the parser associated with "`<Factor>`". At this point, the token is matched with the rule "`<Factor> ::= ( <Expression> )`" , then the rule is explored to the second element "`<Expression>`" on the right-hand side and the parser associated with "`<Expression>`" is invoked to parse the next token "1". The parser associated with "`<Term>`" is invoked, which invokes the parser associated with "`<no>`", which then parses the "1" and return an `Instruction` list of length 1. Parser then continues with "`<Factor>`" and then with "`<Term>`". Then the next token "+" is parsed in the same way. When a token is matched and a complete rule with maximal-length is found, it builds a list of `Instruction` created by calling lower-level parsers and returns the current `Instruction` to the higher-level parser. This process continues until the whole input has been parsed and a parse tree constructed.

The result of parsing is a parse tree shown in Figure 2.9. Each tree node is an `Instruction` which holds its state (an object of rule class) and parsed data (subtree). After parsing expression "(1+2)/3", the parse tree has the following structure:

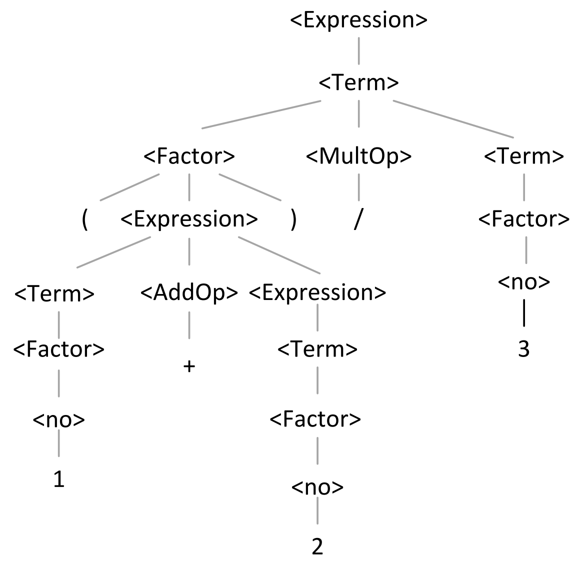


Figure 2.9 Parsing tree of expression "1+2/3"

The main program invokes the root of the graph, traversing the parse tree in pre-order by calling the associated `eval()` routine. If the invoked `Instruction` consists of `Instruction` list, each Instruction in the list will be evaluated first by calling associated `eval()` routine, then the value(s) will be used to compute the value of current `Instruction`.

Therefore, given the parse tree above, the evaluation process will be (uses `{grammar rule}` to represent the associated `Instruction` class):

#	process
1	`{<Expression> ::= <Term>}.eval()`
2	`={<Term> ::= <Factor> <MultOp> <Term>}.eval()`
3	`={<Factor> ::= (<Expression>)}.eval()`
	`/ {<Term> ::= <Factor>}.eval()`
4	`={<Expression> ::= <Term> <AddOp> <Expression>}.eval()`
	`/ {<Term> ::= <Factor>}.eval()`
5	`=({<Term> ::= <Factor>}.eval() + {<Expression> ::= <Term>}.eval())`
	`/ {<Term> ::= <Factor>}.eval()`
6	`=({<Factor > ::= <no>}.eval() + {<Expression> ::= <Term>}.eval())`
	`/ {<Term> ::= <Factor>}.eval()`
7	`=({<Factor > ::= <no>}.eval() + {<Expression> ::= <Term>}.eval())`
	`/ {<Term> ::= <Factor>}.eval()`
8	`=(1  + {<Term> ::= <Factor>}.eval()) / {<Term> ::= <Factor>}.eval()`
9	`=(1  + {<Factor > ::= <no>}.eval()) / {<Term> ::= <Factor>}.eval()`
10	`=(1 + 2) /  {<Term> ::= <Factor>}.eval()`
11	`=3 / {<Factor > ::= <no>}.eval()`
12	`=3 / 3`
13	`=1`

The new design encapsulates each grammar rule into unique class that holds all the information and operations associated to its rule. The classes of different rules inherit from the same super class named `Instruction`, which means they are nearly identical and templates can be used to create the basic structure of the classes. Therefore, it is much faster to build a language with the new extensible interpreter than using traditional interpreter implementation. Moreover, with the registration strategy, modifying or tailoring the language just requires modifying the registration list. As the rules are added to the grammar at runtime, it is also available to add or tailor any existing rule when parsing and bring some special features to the language.

2.4 Syntax Macros

A macro is a rule or pattern that specifies how an input sequence can be expanded to a replacement output sequence. A macro definition generally consists of two parts:

- *Macro rule,* often in the format of a structural description like a grammar rule, which gives a name and a set of formal parameters.
- *Macro body* that specifies how the corresponding macro rule are to be expanded.

The use of a macro name with a set of actual parameters is mapped to some code generated from its body, which is called **macro expansion**.

2.4.1 Macros in Assembly Languages

Macros were first introduced in assembly languages dates to autocoders of the 1950s [4]. Macro assemblers/processors became popular when the assembly languages were commonly used before 1980s. At that time, macros were widely used for the purpose of reducing the code lines, reusing the same code, and adding higher levels of structure to low-level languages.

2.4.2 Macros in High-level Languages

The macro concept to high-level languages were independently introduced by Cheatham [5] and Leavenworth [6] in 1966. They proposed the methods to implement the macro features in the high-level languages. Depending on the phase that macro expansion works at, two main kinds of macros can be identified.

Lexical Macros

Lexical macros were mentioned as one approach to add macro facilities to the high-level language by Cheatham [5]. They work at the level of lexical tokens, allowing input tokens to be substituted by another token sequence at lexical level. Language like C [7] and Unix M4 macro preprocessor [8] are the well-known lexical macro languages. These macros are implemented as lexical preprocessors, which only requires lexical analysis. They are quite similar to the assembly language macros, the only difference is that the macro body is written in the host language.

As lexical macros processing ignores syntax analysis, it cannot provide standard guarantees on the syntax correctness of language. As shown next, C macros can lead some unexpected result without syntax analysis.

Suppose we define a macro SUM(x, y) that will implement x + y, the format is written in the following:

```
#define SUM(x, y) x+y
```

In a code block, we write a following statement to compute (1 + 2)*3:

```
int i = SUM(1, 2) * 3;
```

The expansion will be

```
    int i = 1 + 2 * 3;
```

Apparently, it is not what we expect.

Syntax Macros

Syntax macros, both proposed and recommended by Cheatham [5] and Leavenworth [6] independently, work at the level of syntax trees. The macro body is often parsed into a parse-tree-like structure, called macro skeleton, with placeholders for later macro expansion. During parsing, macro variables are bound to subtrees in the abstract syntax tree and placeholders on macro skeleton are substituted into the corresponding subtrees, returning the actual parsing result with the syntactic structure abstract syntax tree. In contrast to lexical macros, syntax macros are based on syntactic element, and can guarantee syntactic safety [9]. More advantages include encapsulation and syntactic abstraction.

Lisp-like languages, the second-oldest high-level programming language, are the most famous languages with macro facilities. A variety of Lisp dialects have existed over the past 50 years. Nowadays, one of the best known general-purpose Lisp dialects is Common Lisp [11]. It is extensible through standard features such as Lisp macros and reader macros.

As an expression-oriented language, Lisp-like languages represent all code and data as Symbolic expressions (s-expressions) with parenthesized syntax, which can be defined as below:

- an atom which includes a number or string of contiguous characters, or
- a list which includes atoms and/or other lists enclosed in parentheses.

Here, the parenthesis is the feature of Lisp syntax, which helps to represent the nested structure of their code and data.

Lisp syntax uses prefix notation: the first element of an S-expression is commonly an operator or function name and the remaining elements are treated as arguments. For example, the expression “(1+2)/3” will be in the following form in Common Lisp program: (/ (+ 1 2) 3), with the tree structure in the following graph shown in Figure 2.10:

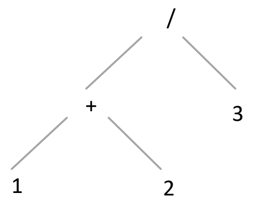


Figure 2.10 Prefix tree of expression “(1+2)/3” in Common Lisp

Lisp syntax lends itself naturally to recursion, in its evaluation process. When an expression is evaluated, it produces a value for the expression, which can then be embedded into other expressions. For example, when evaluating “(/ (+ 1 2) 3)”, it first evaluates the argument list “(+ 1 2)” and “3”, yielding 3 (applying an addition function to the argument list “(1 2))” and “3”). Then the values of argument list are used to evaluate the expression “(/ (+ 1 2) 3)”, by applying division function to the argument list “(3 3)” and producing 1.

Lisp-like languages are ideally suited to be an extensible language, because the uniform syntax with prefix notation makes it easier to determine the invocations of macros [10]. Lisp macro functions serve as powerful tools for the extension of language syntax, allowing programmers to add new syntax constructs or even create a new language.

In Lisp-like languages, a macro definition includes the name of the macro, a parameter list, an optional documentation string, and a body of Lisp expressions that defines the new form to be represented by the macro.

Suppose we want to write a simple macro named plusPlus2, which will take a number and increase its value by 2. The macro definition is:

```
(defmacro plusPlus2 (num)
(setq num (+ num 2)))
```

Then, we excute the following expressions:

```
(defvar x 1)
(print x)
(plusPlus2 x)
(print x)
```

The result returned is:

```
1
3
```

When evaluating the expression "`(plusPlus2 x)`", the main program first looks for the first element of the expression and finds that this expression is a macro expression. Then the expansion function is called with the entire macro call as its first argument (the second argument is a lexical environment), returning some new Lisp form, called the expansion of the macro call. After macro expansion, "`(plusPlus2 x)`" is expanded to "(`setq` x (+ x 2)))", which is then evaluated in place of the original form, producing "x = x+2".

Macros may also lead to a problem called variable capture, when macro expansion causes a name clash, which could happen in a Common Lisp system. To solve this problem, the notion of hygienic macro was introduced in the mid-1980s [12]. Languages with hygienic macros automatically rename variables to prevent subtle but common bugs arising from unintentional variable capture. It was first implemented in Scheme [13], which is now another one of the best known general-purpose Lisp dialects.

Dylan is a general-purpose, high-level programming language, designed for use in application and systems programming [14]. It was inspired by the Scheme and Common Lisp, inheriting much of their semantics. But unlike the Lisp-like languages of fully-parenthesized prefix syntax, Dylan uses the algebraic infix syntax that is similar to C and Pascal.

2.4.3 Existing limitation of Syntax Macros

Unlike the uniform syntax of Lisp-like language, which makes it easier to determine the invocations of macros, most non-Lisp-like languages require that the macro rule starts with a macro name for invocation detection. Most languages require a special syntactic format and additional strategy for macro invocation. For instance, in Dylan, each macro rule has two basic parts: a pattern that is matched against a fragment of code, and a template that is substituted for the matched fragment. When a macro is invoked, each macro rule in the macro rule set is tried in order until a matching pattern is found. As the time complexity to match the pattern is O(n), whereas n is the number of macro rules, if the macro rule set is really large, macro invocation will be slow.

3. Enhancing the Extensible Interpreter with Macro Facility

As mentioned in chapter 2.3, each grammar rule requires a distinct class. Creating a new rule means building a new class that can represent this rule. As the classes of different rules are nearly identical and templates can be used to create the basic structure of the class, it is much faster to build a language with the new extensible interpreter than using the traditional interpreter.

The language developed from the new interpreter strategy aims at teaching introductory students how to write algorithms. The new interpreter with such extensibility ensures students and instructors to add and tailor the language easily, which is the main purpose to the design of the new interpreter. Entry-level students cannot learn the structure of the interpreter and the language used to implement the interpreter. Is there any way to extend the language without knowing anything behind the language?

Yes. To achieve this goal, we enhance the extensible interpreter with a macro facility, which enables the developer to extend the base language without knowing the details of the interpreter.

Here, we call the new extensible interpreter with macro facility as the enhanced extensible interpreter.

In the following sections, we will introduce the basic syntax and structure of macros in the enhanced extensible interpreter. We then show how the macro works in a new language with some examples.

3.1 Syntax of Macro

In a language developed by the enhanced extensible interpreter, a macro definition has the following structure:

```
#MACRO_RULE
Grammar rule
#MACRO_BODY
Macro expansion
#MACRO_END
```

Consider a language with the following base grammar rules that supports the assignment and arithmetic operations:

```
<StatementList> ::= <Statement>
<StatementList> ::= <Statement> <StatementList>
<Statement> ::= <Assignment>
<Assignment> ::=  <id> := <Expression>
<Expression> ::= <Term>
<Expression> ::= <Term> <AddOp> <Expression>
<Term> ::= <Factor>
<Term> ::=  <Factor> <MultOp> <Term>
<AddOp> ::= +
<AddOp> ::= -
<MultOp> ::= *
<MultOp> ::= /
<Factor> ::= ( <Expression> )
<Factor> ::= <id>
<Factor> ::= <no>
```

All the symbols in the left hand side are non-terminals, and <id> and <no> represent terminals.

Here is an example of how to define a macro to extend the base language. Suppose we want to add a macro rule `“++<id>”` to implement `“<id>+1”`, this macro rule will be defined as:

```
#MACRO_RULE
<Expression> ::= ++ <id>
#MACRO_BODY
$id1 + 1
#MACRO_END
```

The macro body specifies the meaning of the corresponding macro rule with the existing language. To correctly and clearly map the non-terminals from the macro rule (actual parameters) to the non-terminals (formal parameters) in macro body, we use `$<non-terminal>Index` to represent the relevant non-terminals in macro body. "$" means the token is a special token, called a placeholder, that needs to be expanded later. The index indicates the position (starts from 0) from the instruction list in the current grammar rule. Thus, the association of parameters between the rule and its body can be built. In this example, there are two symbols in the macro rule: `[++] [<id>]`, in which "++" is the first symbol in position 0, and "`<id>`" is the second one in position 1. Then the macro body will be "`$id1+1`".

In our system, the macro rule and normal rule have the same format, which means the extended language will still look the same as the base language.

3.2 Structure of Macro in Interpreter

In the enhanced extensible interpreter, we define a special Macro class to implement all the macros. All the macro rules belong to a Macro class, which is similar to the basic `Instruction` class. The main differences between the basic Instruction and macro Instruction lie below:

- Each grammar rule in the base language is represented by a distinct class, whereas each macro rule is associated with a unique instance of `Macro` class.
- Each macro rule holds the relevant information of macro body.

`Macro` class contains attributes and methods that are same in the normal `Instruction` class, including:

- A definition of the grammar rule being represented.
- A registration routine that registers the rule along with its parser and an associated translator to be applied when instance of the rule is parsed.

- Declaration of the state needed to record the results of the translation.
- A routine for evaluating (interpreting) the construct.

Besides, the Macro class contains more special information than an Instruction class as below:

- A definition of the macro body;
- A macro body parse routine that parses the macro body string into macro skeleton (a parse tree structure with placeholders);

Here is the `Macro` class definition in the extensible interpreter:

```
public class Macro extends ELSEInstruction{
    private String rule;
       private SyntaxRule syntaxRule;
       private String macroBody;
       private Instruction macroTree;
       public Macro (String macroRule, String macroBody)
    {
       // macro grammar rule defined as a String
              rule = macroRule;

         // macro grammar rule encoded in SyntaxRule
              syntaxRule = new SyntaxRule(GRAMMAR, rule);

        // registers the syntax rule with the grammar,
         // creates an instance of this class and creates the
         // parser to parse any statetment that fits this rule.
              GRAMMAR.addRule(syntaxRule, this,
                         new GenericParser(syntaxRule.lhs()));

              //macro body defined as a String
              this.macroBody = macroBody;

              //macro body parsed as a macro skeleton
              // Now create a scanner
              Grammar grammar = ELSEInstruction.getGrammar();
              GenericScanner scanner;
```

```
                scanner = new GenericScanner(grammar, macroBody);
                // Prime the pump with the first token
                scanner.get();
                macroTree = grammar.parse(
                        grammar.createSymbol(syntaxRule.lhs().toString()),
                        scanner);
        }

    // default to building a full parse tree that records
    // the result list il and current state (class)
        public Instruction createInstruction(InstructionList il) {
        Macro result = new Macro();
                result.macroBody = macroBody;
                result.macroTree = macroTree;
                result.rule = rule;
                result.syntaxRule = syntaxRule;
                result.il = il;
                return result;
    }

    // routine for evaluating (interpreting) the macro
    // construct
        public Value eval(EnvironmentVal env) {
        Instruction mt = copy(macroTree);

          // macro expansion
                expand(mt, il);

          // evaluate the expanded parse tree
          return mt.eval(env);
          }
}
```

The `Macro` class holds all the macros by means of creating distinct instances for each macro rule. There are two parts in a `Macro` instance: macro grammar rule and macro body. A macro grammar rule is defined as a string, encoded as a `SyntaxRule`. A macro body is defined as a string, parsed as a macro skeleton. The syntax macro rule will also be registered with grammar, but in constructor instead of in `initialize()`

routine. The parser builds a list of instructions created by calls to lower-level parsers. When macro evaluator is called, the macro skeleton will be expanded to a parse tree by calling `expand()` routine. Then, the new parse tree is evaluated by the normal evaluating process by calling `eval()` routine associated with the expanded parse tree. More details about macro evaluation will be discussed in 3.3.2.

3.3 Invocations of Macro

The main program begins with an empty grammar. It first registers all the desired grammar rules and macro rules to build the grammar. Then, the main program is able to parse and evaluate the input sentence(s).

3.3.1 Registering Rules

The main program first registers all grammar rules of the base language with grammar, and associates each grammar with its parser and evaluator. Then, it registers all macros with grammar, associates each macro instance to its own parser and evaluator, moreover, parses the macro body to a macro skeleton. The macro rule has to be registered after all the base language that macro body uses because macro skeleton generation needs to call parsing process.

Here we define two types of macros: Simple Macro and Nested Macro. Simple Macro is the macro that includes no macro rule in its macro body. And Nested Macro is the macro that contains other macro rule(s) in its macro body.

Having the macros defined as following:

```
#MACRO_RULE
<Expression> ::= ++ <id>
#MACRO_BODY
$id1 + 1
#MACRO_END
```

```
#MACRO_RULE
<Statement> ::= bump <id>
#MACRO_BODY
$id1 := ++ $id1;
#MACRO_END
```

There are two macro rules "`<Expression> ::= ++ <id>`" and "`<Statement> ::= bump <id>`". The second one is a nested macro rule in which the macro body uses other macro statement to explain the new macro rule. So, we call the first one is a Simple Macro, and the second one is a Nested Macro.

When parsing the macro body to generate the skeleton tree, we use a special token named placeholder to hold the place for further expansion. A placeholder is a special terminal that only exists in the skeleton tree. It works like a formal parameter, when parsing a macro statement, the value of the tokens that serve as the actual parameters can be mapped to the relevant placeholders.

After the macros have been registered to the main program, the macro body of "`$id1+1`" in Simple Macro is parsed as a macro skeleton as below in Figure 3.1:

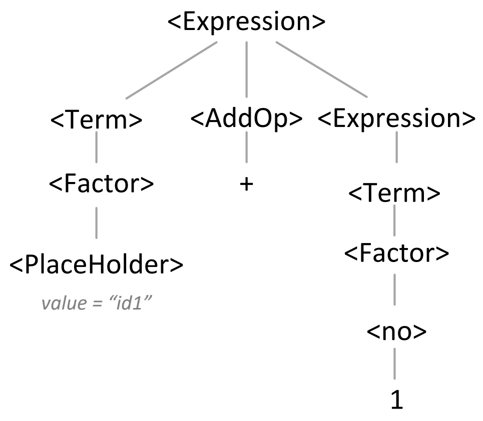


Figure 3.11 Skeleton tree of macro body "`$id1+1`"

For the Nested Macro, the macro body of " $id1 := ++ $id1" is parsed as a macro skeleton shown in Figure 3.2:

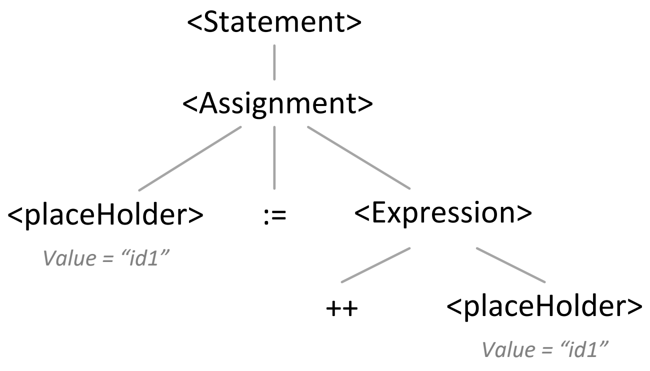


Figure 3.12 Skeleton tree of macro body "`$id1 := ++ $id1`"

The value of placeholder indicates the type and index of the instruction that is used to do parsing and expansion.

3.3.2 Interpreting Macro Statements

After registering the rules to the grammar, the system is ready to interpret the input statements with three steps: scanning, parsing and evaluating.

We will describe how each of them works in our enhanced extensible interpreters with some examples.

Scanning

Given the input statements:

```
a := 1
b := ++ a
bump b
```

The first statement is a basic statement, the second statement is a simple macro statement, and the third statement is a nested macro statement.

The scanner returns a token sequence: [a] [:=] [1] [b] [:=] [++] [a] [bump] [b]. The macro statement is treated as a base statement.

Parsing

The main program invokes the parser associated with the start symbol of the grammar, which, in turn invokes related parsers registered with each member on the right-hand sides. And so on and so forth. The parsing result is a parse tree. Each tree node represents either a non-terminal with an Instruction list, or a terminal that holds the associated token.

The parse tree has the following structure shown in Figure 3.3:

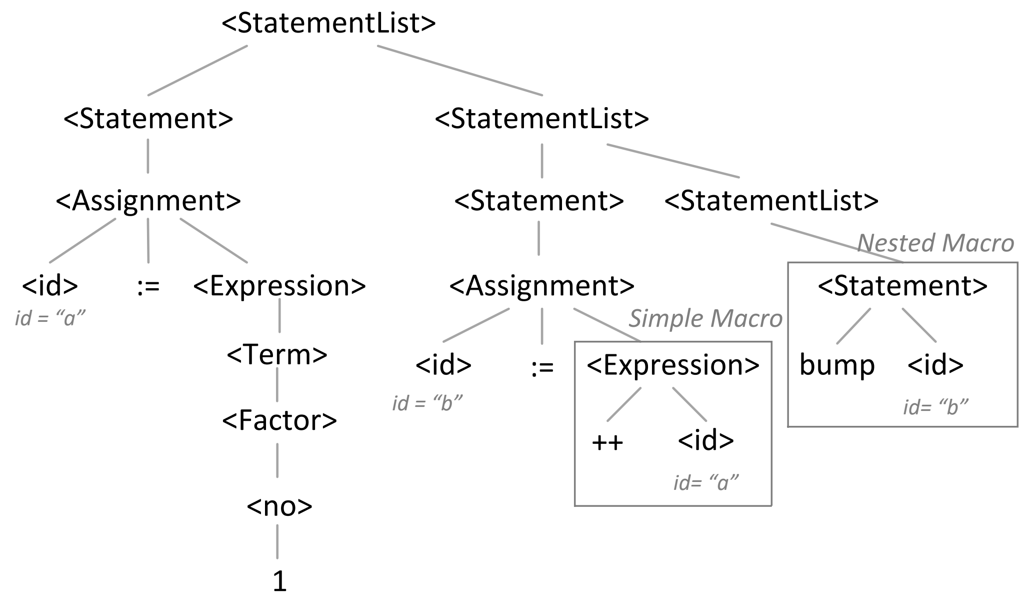


Figure 3.13 Parse tree of input statements

Evaluating

The main program invokes the root of the parse tree, by calling `eval()` routine that is associated with each Instruction in post order.

Recalling the `eval()` routine in Macro class,

```
// routine for evaluating (interpreting) the macro construct
public Value eval(EnvironmentVal env) {
    Instruction mt = copy(macroTree);
    // macro expansion
    expand(mt, il);
    // evaluate the expanded parse tree
          return mt.eval(env);
  }
```

The `eval()` routine of macro Instruction includes three steps:

- *Copy skeleton tree*: It copies a skeleton tree for expansion by calling `copy()` routine, and keeps the original skeleton tree for the macro rule.

The code inside `copy()` routine is:

```
        public Instruction copy (Instruction parseTree)
        {
                    // copy non-terminal
             if (parseTree instanceof ELSEInstruction)
              {
                    InstructionList il = parseTree.getInstructionList();
                    InstructionList newList = new InstructionList();
                    for(int i=0;i<il.size();i++)
                    {
                                 // reclusively call copy() routine to copy all
   the
                    // nodes in the tree
                    Instruction instr = copy(il.get(i));
                                 // build a tree which is same to parseTree
                    newList.add(instr);
                    }
                    return parseTree.createInstruction(newList);
            }
            else
            {
                          //copy terminal
                    return parseTree.copy(parseTree);
            }
        }
```

- *Macro expansion*: It expands the macro skeleton into an expanded tree, by calling `expand()` routine with `Instruction` list of Macro parse tree as the actual parameters for expansion.

Here is the `expand()` routine:

```
// il is Instruction list of Macro parse tree
public void expand(Instruction skeletonTree, InstructionList il)
{
     if (!(skeletonTree instanceof ELSEInstruction))
       return;
       InstructionList stList = skeletonTree.getInstructionList();
      InstructionList mpList = il;
      for(int i=0;i<stList.size();i++)
      {
             Instruction instr = stList.get(i);
           if (instr instanceof PlaceHolder)
             {
                         // compute the index of associated
            // Instruction in il list
             String str = ((PlaceHolder)instr).value;
                    int pos = 0;

                    while (pos < str.length())
              {
                           if (isDigit(str.charAt(pos)))
                           break;
                    pos++;
                    }

            if (pos == str.length())
                    continue;
                    int index = 0;
                    while (pos<str.length())
                    {
                    index = index*10 + str.charAt(pos)-'0';
                    pos++;
                }

            // replace the placeholder with the
            // associated Instruction
                stList.set(i, mpList.get(index));
           }
           else {
```

```
                                  // reclusively call expan() routine to check all
    the
                        // nodes in the tree
                             expand (instr, il);
                 }
               }
          }
```

The `expand()` routine uses depth-first search (DFS) algorithm to traverse the skeleton tree. When it encounters a placeholder, it replaces the placeholder with its associated `Instruction` from the `Instruction` list of macro parse tree. Because each placeholder holds the value that indicates the association with the `Instruction` list of macro parse tree, it is easy to expand the macro skeleton to the expanded tree.

- *Evaluation of expanded tree*. After expansion, the new parse tree is evaluated by the normal evaluating process by calling eval() routine that is associated with new expanded tree.

There are two examples to show how the macro evaluation works.

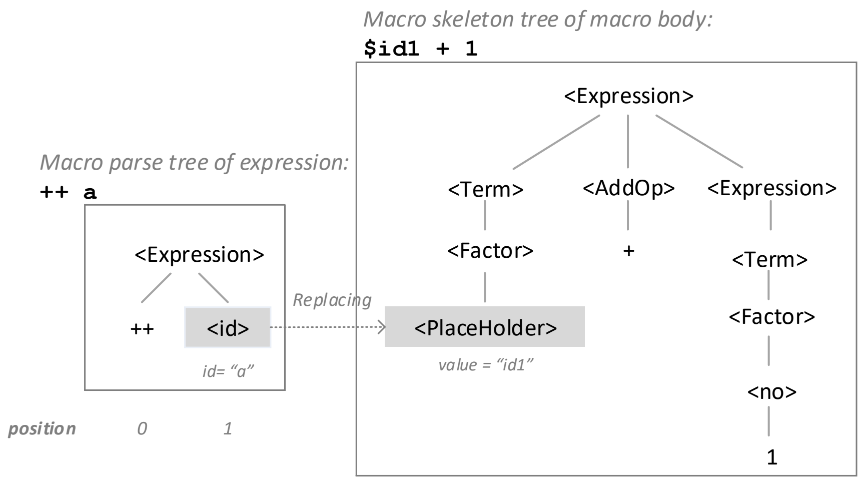
Simple Macro Evaluation

Simple Macro only needs one expansion. Now we want to evaluate the macro expression "`++ a`", where identifier "a" has the value "1". Having the macro parse tree of "`++ a`" from parsing process and macro skeleton tree of "`$id1 + 1`", the evaluation process (uses `{grammar rule}` to represent the associated Instruction class):

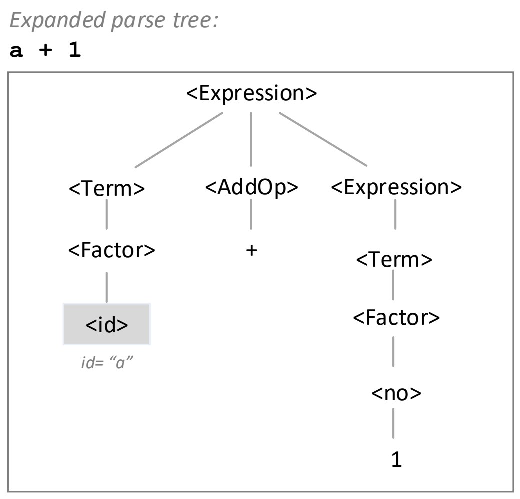
#	process
1	`{<Expression> ::= ++ <id>}.eval()`
2	`={expand(copyOfSkeletonTree, InstructionListOfMacroParseTree)}.eval();`
3	`={<Expression> ::= <Term> <AddOp> <Expression>}.eval();`
4	`={<Term> ::= <Factor>}.eval() + {<Expression> ::= <Term>}.eval();`
5	`={<Factor> ::= <id>}.eval() + {<Expression> ::= <Term>}.eval();`
6	`=1 + {<Expression> ::= <Term>}.eval();`

```
=1 + {<Term> ::= <Factor>}.eval();
=1 + {<Factor> ::= <no>}.eval();
=1 + 1;
=2;
```

#2 the `expand()` routine to expand the macro skeleton tree. Expansion detail is shown in Figure 3.4. After expansion, new expanded tree is evaluated instead of the macro parse tree.



(a)



(b)

Figure 3.14 Macro expansion of expression "`++ a`"

Nested Macro Evaluation

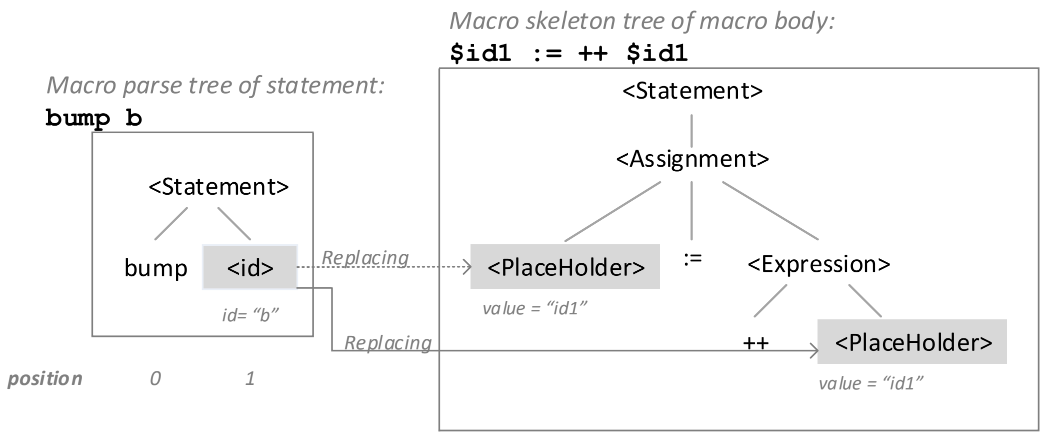
Nested Macro needs more than one expansion during evaluation process. Now we are going to evaluate the macro statement "`bump b`". The identifier "`b`" has the value "2" from previous statements "`a := 1 b := ++ a`". We also have the macro parse tree of "`bump b`" from parsing process and macro skeleton tree of "`$id1 := ++ $id1`", the evaluation process (uses `{grammar rule}` to represent the associated `Instruction` class):

#	process
1	`{<Statement> ::= bump <id>}`
2	`={expand(copyOfSkeletonTree, InstructionListOfMacroParseTree)}.eval();`
3	`={<Statement> ::= <Assignment>}.eval()`
4	`={<Assignment> ::=  <id> := <Expression>}.eval()`
5	`=b := {<Expression> ::= ++ <id>}.eval()`
6	`=b := {expand(copyOfSkeletonTree, InstructionListOfMacroParseTree)}.eval();`
7	`=b := {<Expression> ::= <Term> <AddOp> <Expression>}.eval();`
8	`=b := {<Term> ::= <Factor>}.eval() + {<Expression> ::= <Term>}.eval();`
9	`=b := {<Factor> ::= <id>}.eval() + {<Expression> ::= <Term>}.eval();`
10	`=b := 2 + {<Expression> ::= <Term>}.eval();`
11	`=b := 2 + {<Term> ::= <Factor>}.eval();`
12	`=b := 2 + {<Factor> ::= <no>}.eval();`
13	`=b := 2 + 1;`
14	`=b := 3;`

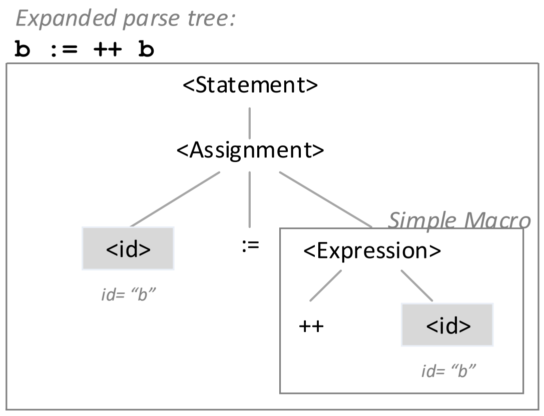
There are two expansions:

The first expansion in #2 expands "`bump b`" to be "`b := ++ b`", seeing expansion process in Figure 3.5, and then calls the `eval()` associated with "`b := ++ b`".

The second expansion is in #6, which expands "`++ b`" to be "`b + 1`" shown in Figure 3.6, and invokes the `eval()` routine associated with "`b + 1`".

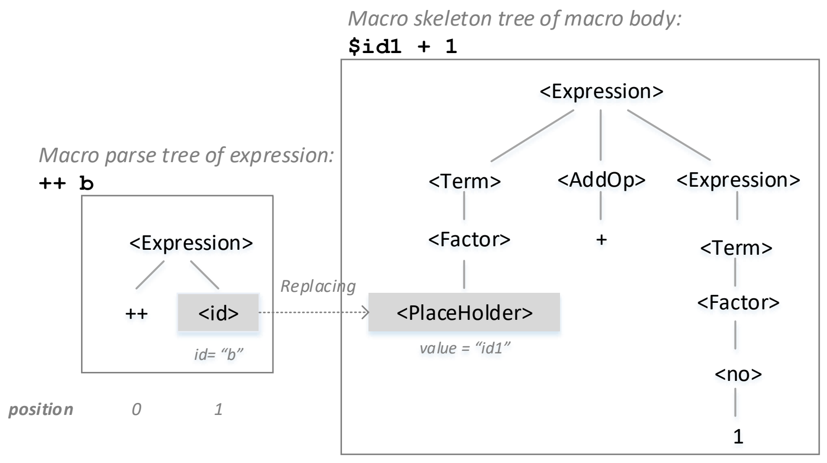


(a)

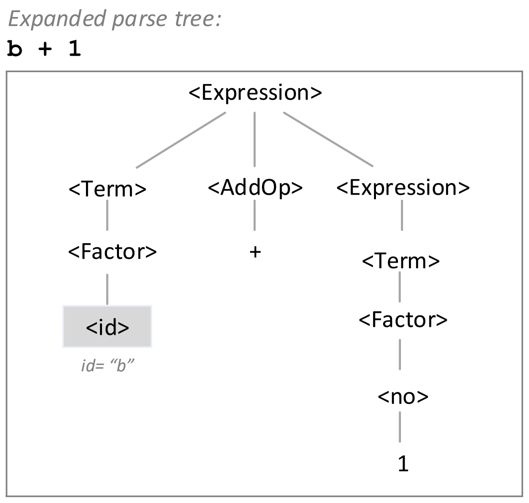


(b)

Figure 3.15 Macro expansion of expression "`bump b`"



(a)



(b)

Figure 3.16 Macro expansion of expression "`++ b`"

4. Implementation

With the new interpreter strategy, we first add macro facility to the extensible interpreter. Then, we build a new language by defining base language grammars and build the `Instruction` class for each grammar. Finally, we define some macro rules to extend the base language and execute some sentences to test the macro rules.

4.1 Adding Macro Facility to the Extensible Interpreter

To add the macro facility in the extensible interpreter, we need to add `Macro` class to the interpreter so that we can define macro rules. The structure of `Macro` class has been introduced in section 3.2.

Besides `Macro` class, we also need to proceed with a new placeholder token in scanner and parser, so that the macro body can be parsed into macro skeleton. Here are the details about the modifications in scanner and parser.

Recognizing Placeholder Token in Generic Scanner

Each token in scanner is represented in a distinct class. The new `PlaceholderToken` class has the following structure, which is similar to other tokens like `NumberToken` in the scanner.

```
public class PlaceholderToken extends Token {
// ------------ Instance variables -------------//
// intentional shadowing.  The inherited value
// must be "<PlaceHolder>"
String value;

// ------------- Constructors ------------------//
public PlaceholderToken (Symbol s, AbstractBuffer b,
                 int lineno, int charpos) {
    super(s,b,lineno,charpos);value = s.getValue();

    // set the inherited value
```

```
            symbol = Symbol.createSymbol(s.getGrammar(), "<PlaceHolder>");
        }

        // ------------- Getters/Setters ----------------//
          public String getId() {return value;}

        // ------------ Other member functions -----------//
               public String toString () {return "[" + value + "]"; }
    }
```

When parsing the input, the `get()` routine in generic scanner is called to load and recognize the next token. In `get()` routine, we add a code block to deal with placeholder shown as below:

```
public Token get() {
    …
    // placeholder for macro skeleton
    else if (ch == '$') {
         buffer.advance();
        value = buffer.getId();
        symbol = Symbol.createSymbol(grammar, value);
       t = new PlaceholderToken(symbol, buffer, lineNo, charPos);
    }
    …
}
```

Processing Placeholder Token in Generic Parser

As mentioned above, a placeholder is a special token that will be parsed as a terminal. We add a unique class `PlaceHolder` to represent the placeholder terminal in the parser:

```
public class PlaceHolder extends Instruction{
    public String value;
         PlaceHolder (String v) { value = v; }
         PlaceHolder (PlaceHolder ph) {
         value = ph.value;
    }

    // parser for PlaceHolder token
    public Instruction parse(Scanner scanner) {
```

```
                String id =(scanner.current().getValue());
                Instruction result;
            result = new PlaceHolder(id);
            scanner.get();
                return result;
            }

         @Override
         public Instruction createInstruction(InstructionList il) {
            return il.getFirst();
         }

        public Instruction copy (PlaceHolder a)
        {
           Instruction result = new PlaceHolder(a.value);
            return result;
        }

            // returning the value as its evaluation
            public Value eval() {
            return new StringVal(value);
            }

         // returning the value as its evaluation
         public Value eval(EnvironmentVal env) {
            return new StringVal(value);
        }
    }
```

We also add some process in the generic parser parse() to deal with placeholder token:

```
public Instruction parse(Scanner scanner) {
…
if (scanner.current().getSymbol().value.equals("<PlaceHolder>") &&
st.equals(s)){
            // parse placeholder
      PlaceHolder ph = new PlaceHolder(scanner.current().getValue());
     instr = ph.parse(scanner);
}
else
{
```

```
            // invoke appropriate parser
        instr = s.parse(scanner);
}
// add current Instruction to the Instruction list
// and search for the associate trie node for the next token
if (instr != null) {
        result.add(instr);
            parent = tn;
        s = scanner.current().getSymbol();
            if (s.value.equals("<PlaceHolder>")){
            String t1 = scanner.current().getValue();
            int pos = t1.length()-1;
                    while (pos >=0)
            {
                if (!isDigit(t1.charAt(pos)))
                        break;
                    pos--;
           }
            String v = "<"+t1.substring(0, pos+1)+">";
            st = s.getGrammar().symbols.get(v);

            if (!(st.isNonTerminal()))
                    tn = tn.getChild(st);
            else
            {
                Symbol ter = tn.getTermial(st);
            if (ter!=null)
                    tn = tn.getChild(ter);
                    else
                            tn = null;
             }
            }
            else
          tn = tn.getChild(s);
        }
  else {
       // parsing failed
            tn = null;
   }
…
}
```

4.2 Runtime Example

We build a little language named ELSE to verify our design. The process includes the three steps: building language, defining macro rules, and executing language with macro statements. The whole project is developed in java with NetBeans IDE.

Creating a New Little Language ELSE

The little language ELSE has the following grammar rules:

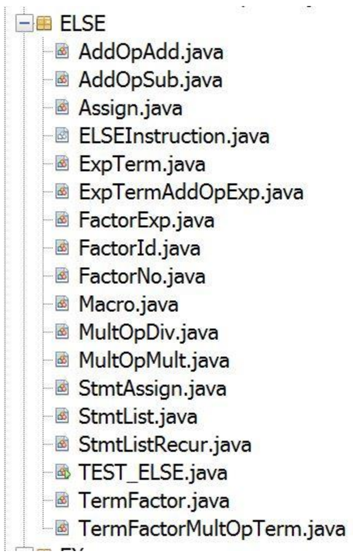
```
<StatementList> ::= <Statement>
<StatementList> ::= <Statement> <StatementList>
<Statement> ::= <Assignment>
<Assignment> ::=  <id> := <Expression>
<Expression> ::= <Term>
<Expression> ::= <Term> <AddOp> <Expression>
<Term> ::= <Factor>
<Term> ::=  <Factor> <MultOp> <Term>
<AddOp> ::= +
<AddOp> ::= -
<MultOp> ::= *
<MultOp> ::= /
<Factor> ::= ( <Expression> )
<Factor> ::= <id>
<Factor> ::= <no>
```

In our project, we put all the base grammar rules in a file named “ELSE.grm” with the following content:

```
Package ELSE
Author Wendy
Year 2017

StmtList               <StatementList> ::= <Statement>
StmtListRecur          <StatementList> ::= <Statement> <StatementList>
StmtAssign             <Statement> ::= <Assignment>
Assign                 <Assignment> ::=  <id> := <Expression>
ExpTerm                <Expression> ::= <Term>
ExpTermAddOpExp        <Expression> ::= <Term> <AddOp> <Expression>
TermFactor             <Term> ::= <Factor>
TermFactorMultOpTerm   <Term> ::=  <Factor> <MultOp> <Term>
AddOpAdd               <AddOp> ::= +
AddOpSub               <AddOp> ::= -
MultOpMult             <MultOp> ::= *
MultOpDiv              <MultOp> ::= /
FactorExp              <Factor> ::= ( <Expression> )
FactorId               <Factor> ::= <id>
FactorNo               <Factor> ::= <no>
```

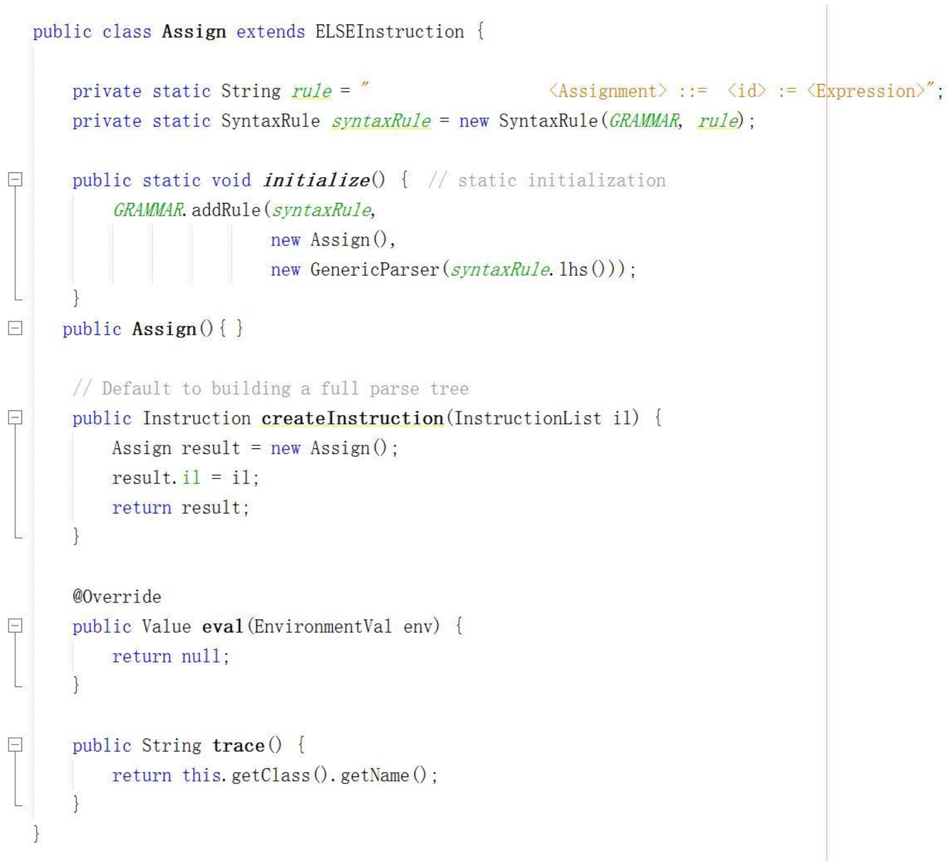
Notice that each grammar rule in the file starts with a name that will turn to the class name of the associated grammar rule.

There is a template routine to build the `Instruction` class for each grammar rule. After building the grammar rules class, a package named "`ELSE`" that holds all the base language `Instruction` classes (grammar rules) as well as `Macro` class (macro grammar rules) is built. Here is the file list in `ELSE` package:



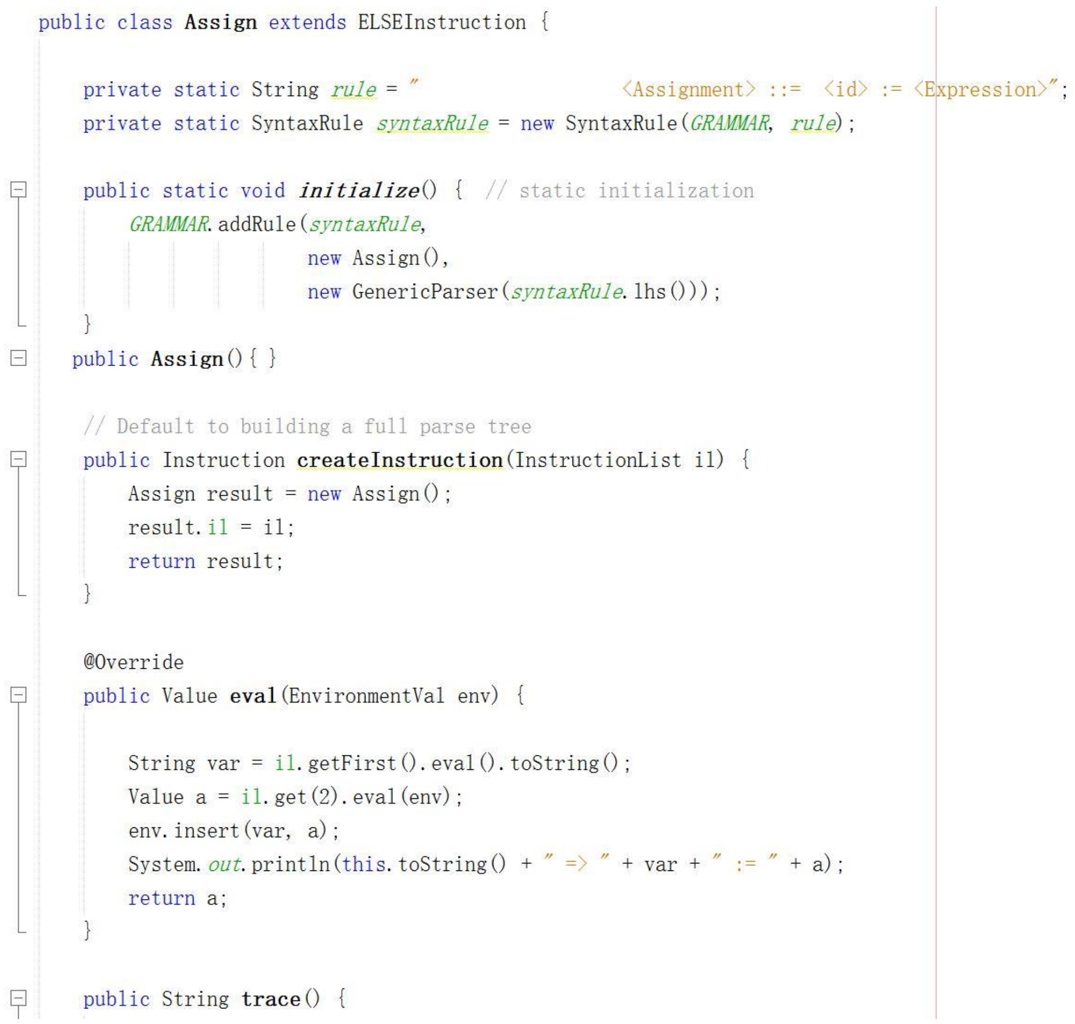
```
ELSE
    AddOpAdd.java
    AddOpSub.java
    Assign.java
    ELSEInstruction.java
    ExpTerm.java
    ExpTermAddOpExp.java
    FactorExp.java
    FactorId.java
    FactorNo.java
    Macro.java
    MultOpDiv.java
    MultOpMult.java
    StmtAssign.java
    StmtList.java
    StmtListRecur.java
    TEST_ELSE.java
    TermFactor.java
    TermFactorMultOpTerm.java
```

Take the grammar rule "`<Assignment> ::=  <id> := <Expression>`" as an example, the class has the following structure after applying the template on it:



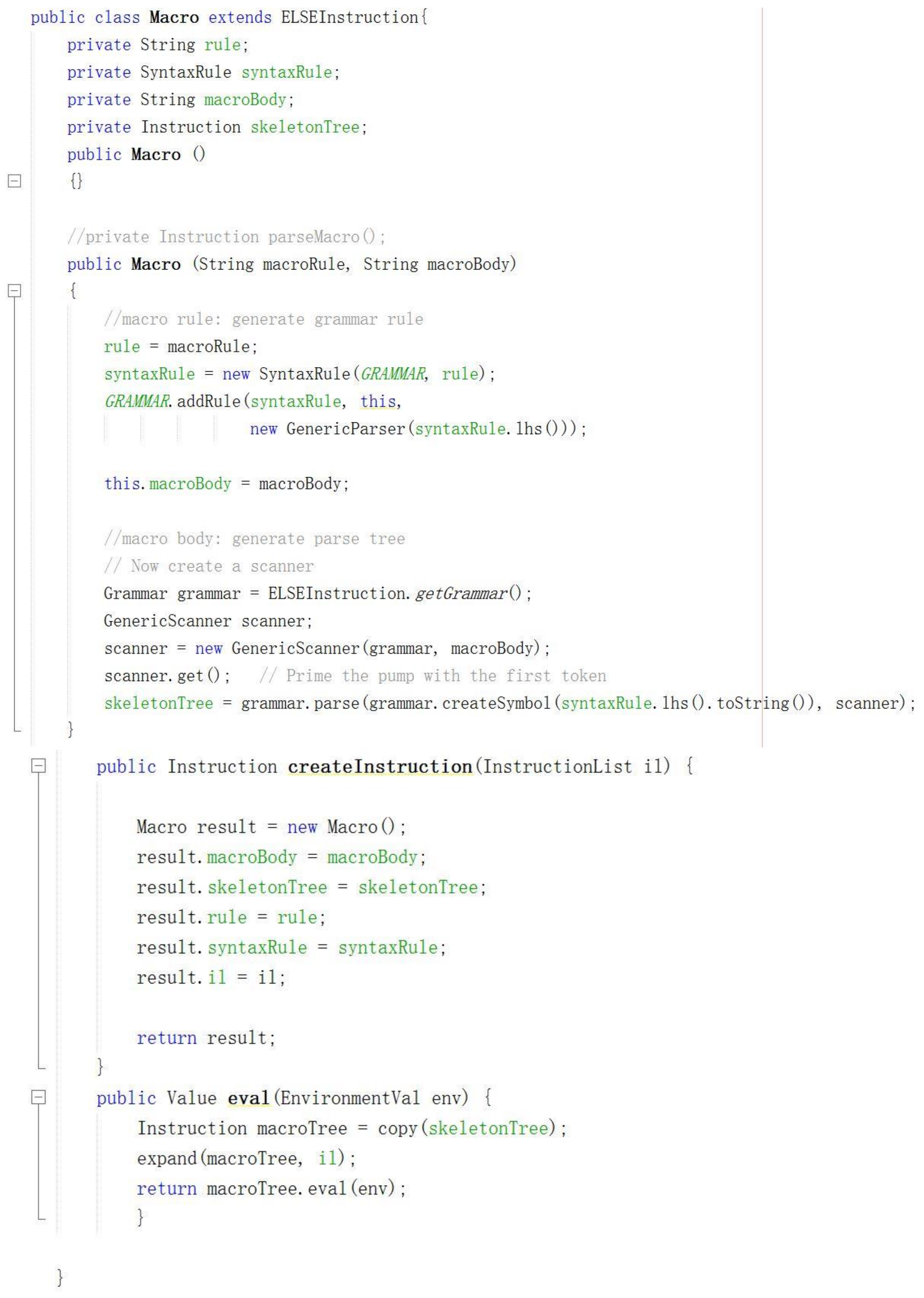
```
public class Assign extends ELSEInstruction {

    private static String rule = "                  <Assignment> ::=  <id> := <Expression>";
    private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

    public static void initialize() {  // static initialization
        GRAMMAR.addRule(syntaxRule,
                        new Assign(),
                        new GenericParser(syntaxRule.lhs()));
    }
   public Assign(){ }

    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
        Assign result = new Assign();
        result.il = il;
        return result;
    }

    @Override
    public Value eval(EnvironmentVal env) {
        return null;
    }

    public String trace() {
        return this.getClass().getName();
    }
}
```

Then, we manually modified the `eval()` routine for each class. The final class after modification is:



```
public class Assign extends ELSEInstruction {

    private static String rule = "                   <Assignment> ::=  <id> := <Expression>";
    private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

    public static void initialize() {  // static initialization
        GRAMMAR.addRule(syntaxRule,
                    new Assign(),
                    new GenericParser(syntaxRule.lhs()));
    }
   public Assign(){ }

    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
        Assign result = new Assign();
        result.il = il;
        return result;
    }

    @Override
    public Value eval(EnvironmentVal env) {

        String var = il.getFirst().eval().toString();
        Value a = il.get(2).eval(env);
        env.insert(var, a);
        System.out.println(this.toString() + " => " + var + " := " + a);
        return a;
    }

    public String trace() {
```

The `Macro` class in the project has the following structure, which has been introduced in the previous section:



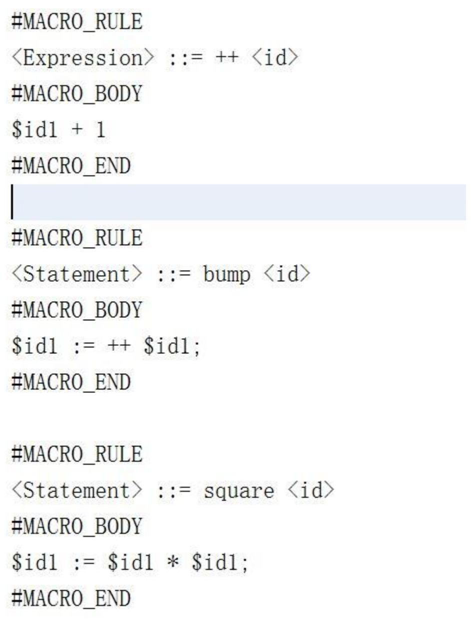
```
public class Macro extends ELSEInstruction{
    private String rule;
    private SyntaxRule syntaxRule;
    private String macroBody;
    private Instruction skeletonTree;
    public Macro ()
    {}

    //private Instruction parseMacro();
    public Macro (String macroRule, String macroBody)
    {
        //macro rule: generate grammar rule
        rule = macroRule;
        syntaxRule = new SyntaxRule(GRAMMAR, rule);
        GRAMMAR.addRule(syntaxRule, this,
                        new GenericParser(syntaxRule.lhs()));

        this.macroBody = macroBody;

        //macro body: generate parse tree
        // Now create a scanner
        Grammar grammar = ELSEInstruction.getGrammar();
        GenericScanner scanner;
        scanner = new GenericScanner(grammar, macroBody);
        scanner.get();   // Prime the pump with the first token
        skeletonTree = grammar.parse(grammar.createSymbol(syntaxRule.lhs().toString()), scanner);
    }
    public Instruction createInstruction(InstructionList il) {

        Macro result = new Macro();
        result.macroBody = macroBody;
        result.skeletonTree = skeletonTree;
        result.rule = rule;
        result.syntaxRule = syntaxRule;
        result.il = il;

        return result;
    }
    public Value eval(EnvironmentVal env) {
        Instruction macroTree = copy(skeletonTree);
        expand(macroTree, il);
        return macroTree.eval(env);
        }

}
```

Defining Macro Rules in ELSE

Now we can define macro rules to extend the base language. All the macro rules are written in a file named "ELSEMacro.grm". We define three macro rules:

```
<Expression> ::= ++ <id>
<Statement> ::= bump <id>
<Statement> ::= square <id>
```

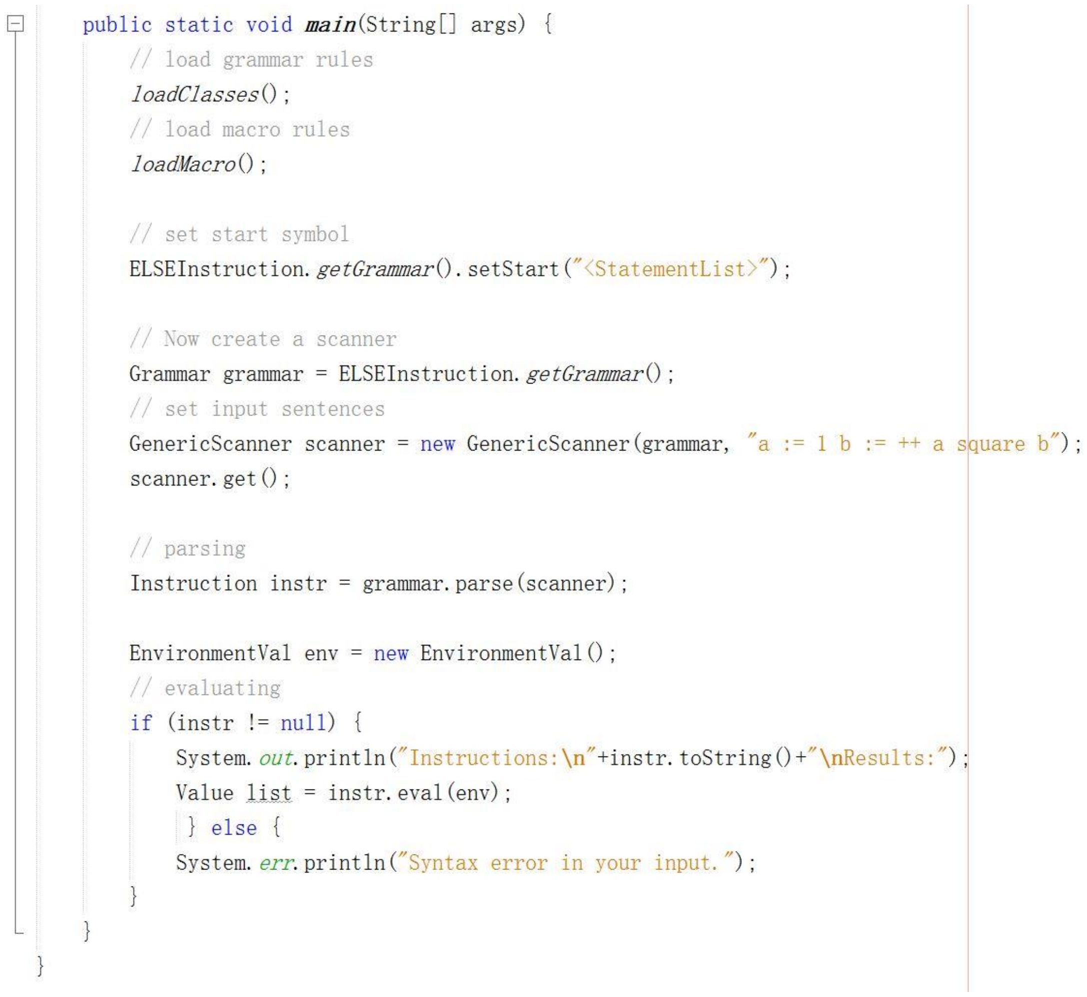
The corresponding macros defined in "ELSEMacro.grm" are:



```
#MACRO_RULE
<Expression> ::= ++ <id>
#MACRO_BODY
$id1 + 1
#MACRO_END

#MACRO_RULE
<Statement> ::= bump <id>
#MACRO_BODY
$id1 := ++ $id1;
#MACRO_END

#MACRO_RULE
<Statement> ::= square <id>
#MACRO_BODY
$id1 := $id1 * $id1;
#MACRO_END
```

Executing ELSE

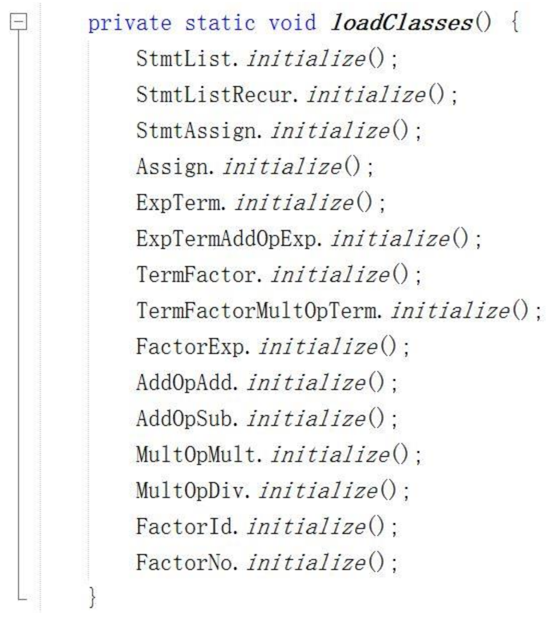
The main program to execute the ELSE is `main()` routine in "TEST_ELSE.java":



```
    public static void main(String[] args) {
        // load grammar rules
        loadClasses();
        // load macro rules
        loadMacro();

        // set start symbol
        ELSEInstruction.getGrammar().setStart("<StatementList>");

        // Now create a scanner
        Grammar grammar = ELSEInstruction.getGrammar();
        // set input sentences
        GenericScanner scanner = new GenericScanner(grammar, "a := 1 b := ++ a square b");
        scanner.get();

        // parsing
        Instruction instr = grammar.parse(scanner);

        EnvironmentVal env = new EnvironmentVal();
        // evaluating
        if (instr != null) {
            System.out.println("Instructions:\n"+instr.toString()+"\nResults:");
            Value list = instr.eval(env);
            } else {
            System.err.println("Syntax error in your input.");
        }
    }
}
```

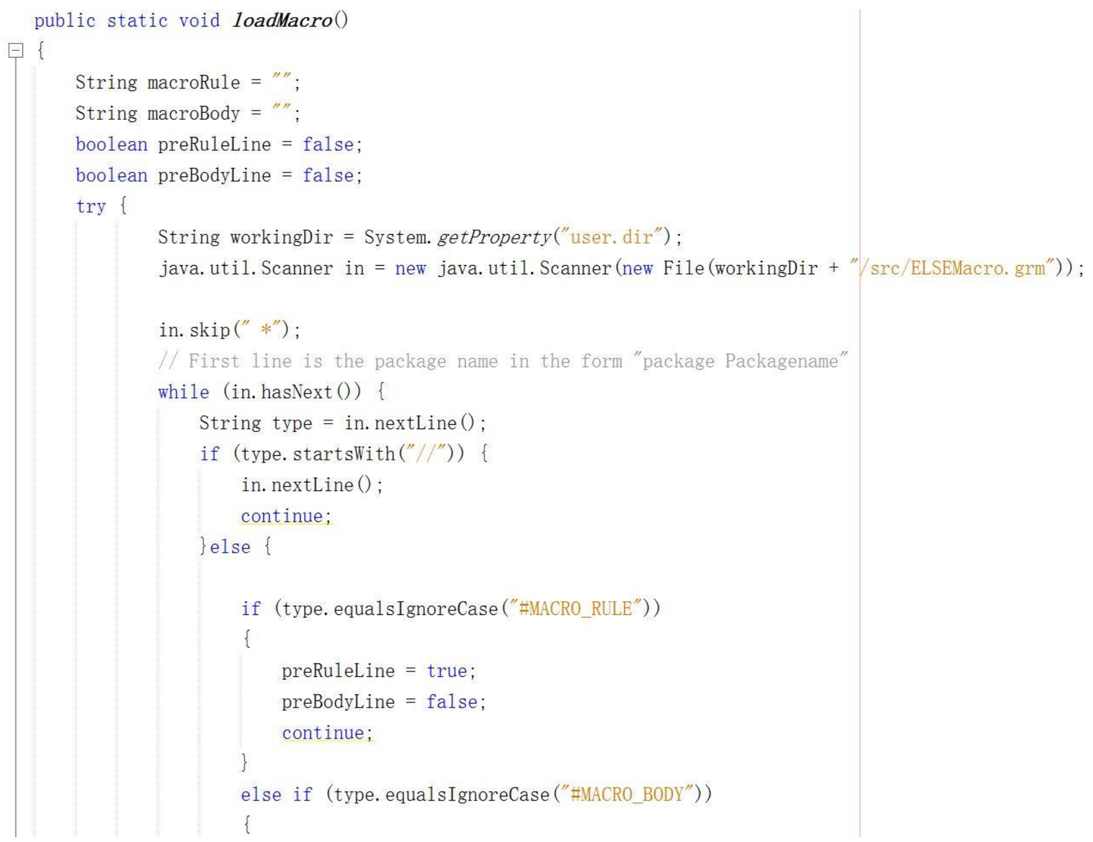
Having the Instruction classes of grammar rules and Macro class, we add these rules to the grammar at the beginning of the main program by calling `loadClasses()` routine and `loadMacro()` routine.

`loadClasses()` routine calls the registration routine `initialize()` for each desired grammar rule.

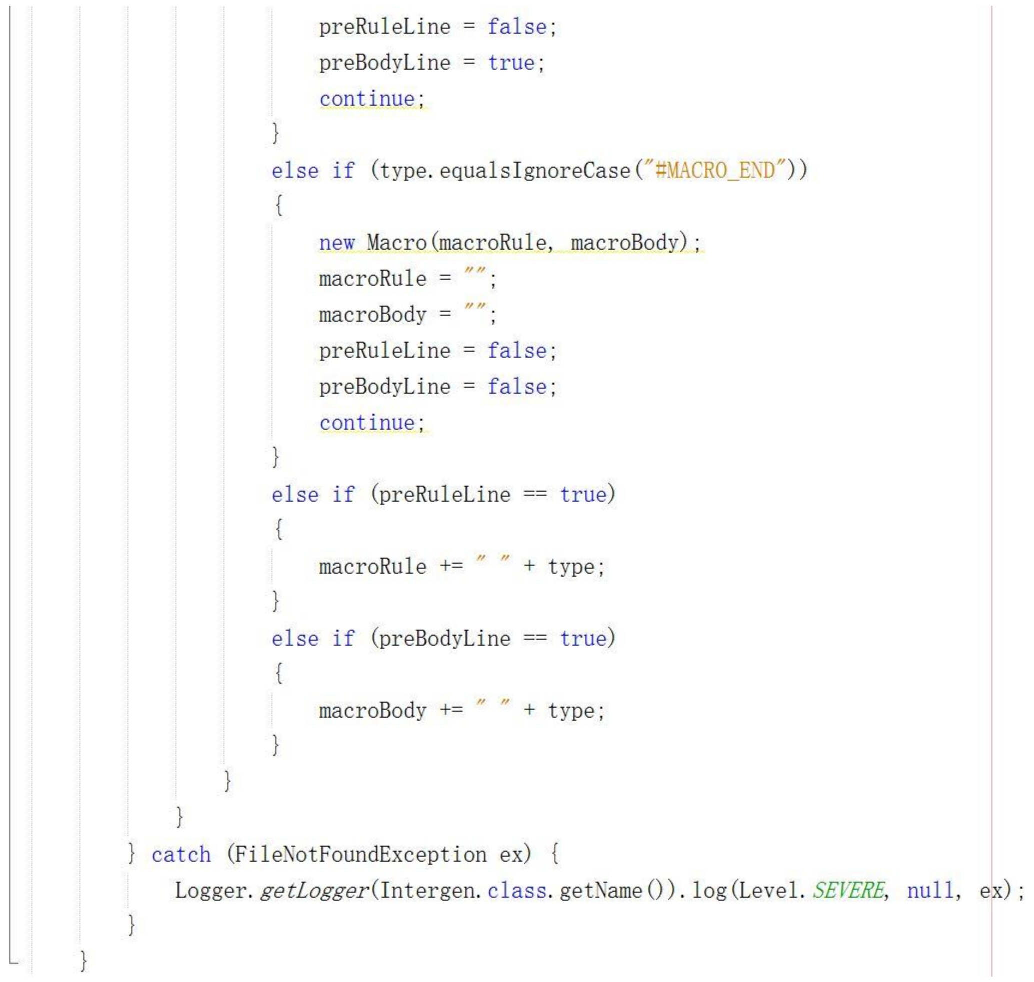


```
private static void loadClasses() {
    StmtList.initialize();
    StmtListRecur.initialize();
    StmtAssign.initialize();
    Assign.initialize();
    ExpTerm.initialize();
    ExpTermAddOpExp.initialize();
    TermFactor.initialize();
    TermFactorMultOpTerm.initialize();
    FactorExp.initialize();
    AddOpAdd.initialize();
    AddOpSub.initialize();
    MultOpMult.initialize();
    MultOpDiv.initialize();
    FactorId.initialize();
    FactorNo.initialize();
}
```

`loadMacro()` routine reads “ELSEMacro.grm” and builds instances for each macro rule:

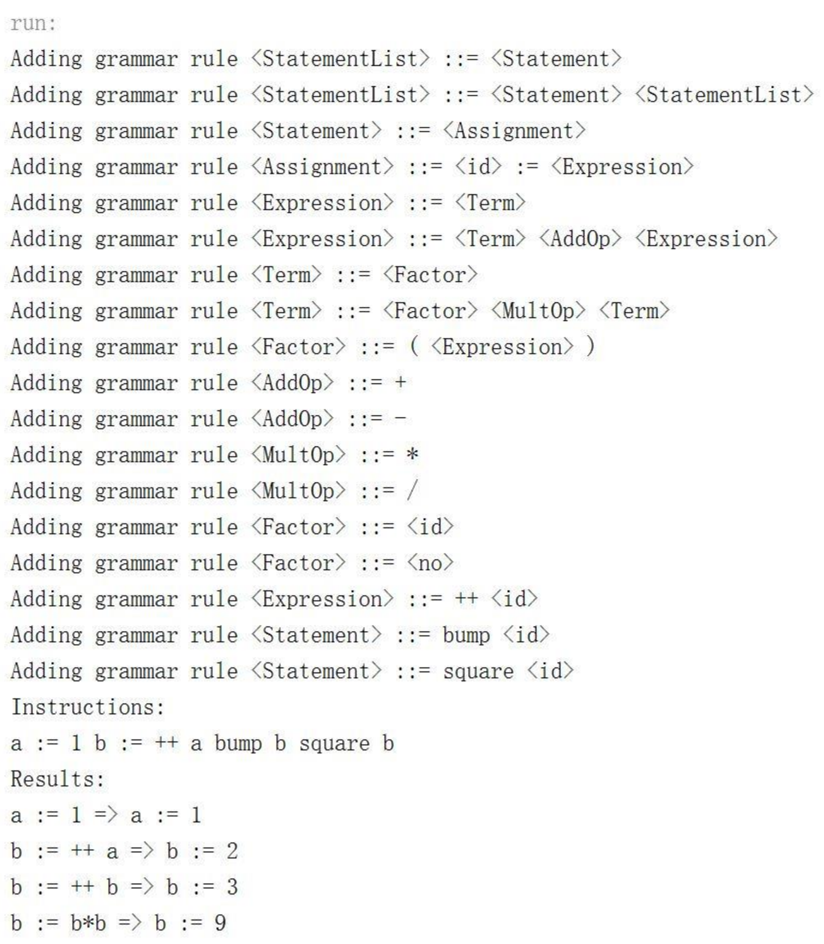


```
public static void loadMacro()
{
    String macroRule = "";
    String macroBody = "";
    boolean preRuleLine = false;
    boolean preBodyLine = false;
    try {
            String workingDir = System.getProperty("user.dir");
            java.util.Scanner in = new java.util.Scanner(new File(workingDir + "/src/ELSEMacro.grm"));

            in.skip(" *");
            // First line is the package name in the form "package Packagename"
            while (in.hasNext()) {
                String type = in.nextLine();
                if (type.startsWith("//")) {
                    in.nextLine();
                    continue;
                }else {

                    if (type.equalsIgnoreCase("#MACRO_RULE"))
                    {
                        preRuleLine = true;
                        preBodyLine = false;
                        continue;
                    }
                    else if (type.equalsIgnoreCase("#MACRO_BODY"))
                    {
```



```
                    preRuleLine = false;
                    preBodyLine = true;
                    continue;
                }
                else if (type.equalsIgnoreCase("#MACRO_END"))
                {
                    new Macro(macroRule, macroBody);
                    macroRule = "";
                    macroBody = "";
                    preRuleLine = false;
                    preBodyLine = false;
                    continue;
                }
                else if (preRuleLine == true)
                {
                    macroRule += " " + type;
                }
                else if (preBodyLine == true)
                {
                    macroBody += " " + type;
                }
            }
        }
    } catch (FileNotFoundException ex) {
        Logger.getLogger(Intergen.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

We input "`a := 1 b := ++ a bump b square b`" and execute the program, the results are:



```
run:
Adding grammar rule <StatementList> ::= <Statement>
Adding grammar rule <StatementList> ::= <Statement> <StatementList>
Adding grammar rule <Statement> ::= <Assignment>
Adding grammar rule <Assignment> ::= <id> := <Expression>
Adding grammar rule <Expression> ::= <Term>
Adding grammar rule <Expression> ::= <Term> <AddOp> <Expression>
Adding grammar rule <Term> ::= <Factor>
Adding grammar rule <Term> ::= <Factor> <MultOp> <Term>
Adding grammar rule <Factor> ::= ( <Expression> )
Adding grammar rule <AddOp> ::= +
Adding grammar rule <AddOp> ::= -
Adding grammar rule <MultOp> ::= *
Adding grammar rule <MultOp> ::= /
Adding grammar rule <Factor> ::= <id>
Adding grammar rule <Factor> ::= <no>
Adding grammar rule <Expression> ::= ++ <id>
Adding grammar rule <Statement> ::= bump <id>
Adding grammar rule <Statement> ::= square <id>
Instructions:
a := 1 b := ++ a bump b square b
Results:
a := 1 => a := 1
b := ++ a => b := 2
b := ++ b => b := 3
b := b*b => b := 9
```

5. Summary and Future Work

We enhanced an extensible interpreter with a syntax macro facility, which can only be developed under the new extensible interpreter strategy.

The macro feature we added to the interpreter inherits from `Instruction` class, which makes the development process fast and efficient. To add the macro facility to the extensible interpreter, we need to:

- Design and build a new syntax macro facility, named `Macro`, which allows the programmer to define the macro rules to extend the language.
- Enable the scanner to recognize placeholder tokens when parsing a macro body.
- Deal with the placeholder token in generic parser so that the macro body can be parsed into macro skeleton.

For the structure of `Macro` class, which inherits from `Instruction` class, we added two additional attributes "`macroBody`" and "`skeletonTree`" to hold the macro body, modify the constructor to parse the skeleton tree, and add `expand()` routine in `eval()` routine to do macro expansion before evaluation.

The syntax macro facility does not require special format for macro invocation. The macro grammar rule and the base grammar rule look the same. Because each macro rule is associated with a unique Macro instance, and each instance holds all the information necessary to scan, parse, expand and interpret a macro statement in a single `Instruction` class. The process of macro invocation is almost the same as a base instruction invocation, excepting the expansion before evaluation.

With the syntax macro facility, anyone can extend the base language without knowing anything about the interpreter. They only need to know one language – the base language, and extend the language based on current language. Similar to the base grammar rule, it allows macro rules to be created and added to the system as it is parsing.

For our next step, we will explore how small the base language can be to satisfy the extension without limitation. For instance, if we define a language that only consists of arithmetic expression, it impossible to add a loop macro to the language.

References

[1] C. N. Fischer et al., *Crafting a Compiler*, Addison-Wesley, 2009.

[2] R. W. Sebesta, *Concepts of Programming Languages,* 10th ed. Boston, MA: Pearson, 2012.

[3] L. Morell, “Design of an extensible interpreter using information hiding”, *Journal of Computing Sciences in Colleges*, vol. 26, no. 5, pp.130-136, May 2011.

[4] *Executive Guide to the IBM 1440 Data Processing System*, International Business Machines Corporation (IBM), 1962.

[5] T. E. Cheatham, Jr., “The introduction of definitional facilities into higher level programming languages”, *AFIPS '66 (Fall) Proc. November 7-10, 1966, fall joint computer conf.*, San Francisco, CA, 1966, pp. 623-637.

[6] B. M. Leavenworth, “Syntax macros and extended translation”, *Communications of the ACM*, vol. 9, no. 11, pp. 790-793, Nov. 1966.

[7] B. W. Kernighan, D. M. Ritchie, *The C programming language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.

[8] R. Adams, “Take command: The m4 macro package”, *Linux Journal*, vol. 2002, no. 96, pp. 6, April 2002.

[9] Daniel Weise, Roger Crew, “Programmable syntax macros”, *Proc. ACM SIGPLAN 1993 conf. on Programming language design and implementation*, Albuquerque, NM, 1993, pp. 156-165.

[10] D. K. Layer, C. Richardson, “Lisp systems in the 1990s”, *Communications of the ACM*, vol. 34, no. 9, pp. 48-57, Sept. 1991.

[11] G. L. Steele, *Common Lisp the Language*, 2nd ed, Newton, MA: Digital Press, 1990.

[12] E. Kohlbecker et al., “Hygienic macro expansion”, *Proc. 1986 ACM conference on LISP and functional programming*, Cambridge, MA, 1986, pp. 151-161.

[13] R. K. Dybvig et al., “Syntactic abstraction in Scheme”, *Lisp and Symbolic Computation*, vol. 5, no. 4, pp. 295-326, Dec. 1992.

[14] A. Shalit, *The Dylan Reference Manual*. Apple Press, 1998.

Appendix A.

Codes to Implement the Enhanced Extensible Interpreter

The project is developed in java with NetBeans IDE 8.1. There are three packages to implement the new language ELSE:

- ELSE: implement grammar rules and macro rules of new language ELSE
- scanner: scanner of the interpreter
- parsing: parser of the interpreter

Code in package "ELSE"

AddOpAdd.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */
public class AddOpAdd extends ELSEInstruction {

       private static String rule = "               <AddOp> ::= +";
       private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

       public static void initialize() {  // static initialization
              GRAMMAR.addRule(syntaxRule,
                              new AddOpAdd(),
                              new GenericParser(syntaxRule.lhs()));
       }
    public AddOpAdd(){ }

   @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
       AddOpAdd result = new AddOpAdd();
              result.il = il;
              return result;
    }
    @Override
    public Value eval(EnvironmentVal env) {
       return null;
       }
    public String trace() {
              return this.getClass().getName();
    }
}
```

AddOpSub.java

```
package ELSE;

import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class AddOpAdd extends ELSEInstruction {

	private static String rule = "		<AddOp> ::= +";
	private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

	public static void initialize() {  // static initialization
		GRAMMAR.addRule(syntaxRule,
				new AddOpAdd(),
				new GenericParser(syntaxRule.lhs()));
	}
	public AddOpAdd(){ }

	@Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
		AddOpAdd result = new AddOpAdd();
		result.il = il;
		return result;
	}

	@Override
	public Value eval(EnvironmentVal env) {
		return null;
	}
    public String trace() {
	return this.getClass().getName();
    }

}
```

Assign.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;
import scanner.GenericScanner;

/**
 * @author Wendy
 */

public class Assign extends ELSEInstruction {

    private static String rule = "			<Assignment> ::=  <id> :=
<Expression>";
    private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

    public static void initialize() {  // static initialization
```

```
            GRAMMAR.addRule(syntaxRule,
                  new Assign(),
                              new GenericParser(syntaxRule.lhs()));
    }
      public Assign(){ }

    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
        Assign result = new Assign();
        result.il = il;
        return result;
    }

    @Override
    public Value eval(EnvironmentVal env) {

        String var = il.getFirst().eval().toString();
        Value a = il.get(2).eval(env);
        env.insert(var, a);
        System.out.println(this.toString() + " => " + var + " := " + a);
        return a;
    }

    public String trace() {
        return this.getClass().getName();
    }
}
```

ELSEInstruction.java

```
package ELSE;
import static java.lang.Character.isDigit;
import parsing.*;
import runtime.*;
import java.util.Iterator;
import java.util.ArrayList;
import parsing.*;import runtime.*;
import java.util.Iterator;

public abstract class ELSEInstruction extends Instruction {
      // Define a possible runtime environment.  In this case we'll just
      // define it to be a stack of SN's.  All ELSEInstruction instructions can
      // therefore access this static stack.

      // one grammar for all ELSEInstruction instructions
      public final static Grammar GRAMMAR = new Grammar();

      public ELSEInstruction() { }
            public ELSEInstruction(ELSEInstruction a){}
            public ELSEInstruction(Instruction nextInstruction) {
                  this.nextInstruction = nextInstruction; // default
      }

      public ELSEInstruction(Source s, InstructionList parsedList) {
            super(s, parsedList);
      }

      public static Grammar getGrammar() {
            return GRAMMAR;
      }
      @Override
      public Instruction createInstruction(InstructionList il) {
```

```
     return il.getFirst();
     }
     public InstructionList exec() {
            throw new UnsupportedOperationException("Exec not supported.");
     }
     // implement required functions with defaults
     public Object value() {return null;}
     public Value eval() {return null;}
     public Value eval(EnvironmentVal env){return null;}

     public InstructionList getInstructionList() {
      return this.il;
     }
     public String toString() {
            String str="";
            for(int i=0;i<il.size();i++)
            {
            if (i > 0)
                   str += " ";
            str += il.get(i).toString();
            }
            return str;
  }
  public Instruction copy (Instruction parseTree)
  {
            if (parseTree instanceof ELSEInstruction)
            {
                   InstructionList il = parseTree.getInstructionList();
                   InstructionList newList = new InstructionList();
                   for(int i=0;i<il.size();i++)
                   {
                   Instruction instr = copy(il.get(i));
                   newList.add(instr);
                   }
                   return parseTree.createInstruction(newList);
            }
            else
            {
                   return parseTree.copy(parseTree);
            }
  }
  public void expand(Instruction skeletonTree, InstructionList il)
  {
      if (!(skeletonTree instanceof ELSEInstruction))
                   return;
      InstructionList stList = skeletonTree.getInstructionList();
            InstructionList mpList = il;
            for(int i=0;i<stList.size();i++)
            {
          Instruction instr = stList.get(i);
            if (instr instanceof PlaceHolder)
            {
                   String str = ((PlaceHolder)instr).value;
                   int pos = 0;
                   //id1->1
                   while (pos < str.length())
                   {
                          if (isDigit(str.charAt(pos)))
                          break;
                          pos++;
                   }
                   if (pos == str.length())
```

```
                                continue;
                        int index = 0;
                        while (pos<str.length())
                        {
                                index = index*10 + str.charAt(pos)-'0';
                                pos++;
                        }
                        stList.set(i, mpList.get(index));
                }
                else
                        expand (instr, il);
                }
     }
}
```

ExpTem.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class ExpTerm extends ELSEInstruction {

        private static String rule = "                  <Expression> ::= <Term>";
        private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

        public static void initialize() {  // static initialization
                GRAMMAR.addRule(syntaxRule,
                                new ExpTerm(),
                                new GenericParser(syntaxRule.lhs()));
        }
        public ExpTerm(){ }

        @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
                ExpTerm result = new ExpTerm();
                result.il = il;
                return result;
        }

        @Override
        public Value eval(EnvironmentVal env) {
              return il.get(0).eval(env);
        }
    public String trace() {
                return this.getClass().getName();
    }
    public String toString() {
                return il.getFirst().toString();
    }
}
```

ExpTermAddOpExp.java

```
package ELSE;
```

```
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class ExpTermAddOpExp extends ELSEInstruction {

       private static String rule = "         <Expression> ::= <Term> <AddOp>
<Expression>";
       private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

       public static void initialize() {  // static initialization
              GRAMMAR.addRule(syntaxRule,
                              new ExpTermAddOpExp(),
                              new GenericParser(syntaxRule.lhs()));
       }
       public ExpTermAddOpExp(){ }

       @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
              ExpTermAddOpExp result = new ExpTermAddOpExp();
              result.il = il;
              return result;
       }

       @Override
       public Value eval(EnvironmentVal env) {
              Value a = il.get(0).eval(env);
              Value b = il.get(2).eval(env);
              if (a instanceof DoubleVal && b instanceof DoubleVal)
              {
              if (il.get(1) instanceof AddOpAdd)
                            return new DoubleVal(((DoubleVal)a).getVal() +
((DoubleVal)b).getVal());
              if (il.get(1) instanceof AddOpSub)
                            return new DoubleVal(((DoubleVal)a).getVal() -
((DoubleVal)b).getVal());
              }
              return null;
       }
    public String trace() {
       return this.getClass().getName();
    }
}
```

FactorExp.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class FactorExp extends ELSEInstruction {
	private static String rule = "			<Factor> ::= ( <Expression> 
)";
	private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

	public static void initialize() {  // static initialization
		GRAMMAR.addRule(syntaxRule,
				new FactorExp(),
				new GenericParser(syntaxRule.lhs()));
	}
	public FactorExp(){ }

	@Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
		FactorExp result = new FactorExp();
		result.il = il;
		return result;
	}

	@Override
	public Value eval(EnvironmentVal env) {
		Value value = il.get(1).eval(env);
		return value;
	}
    public String trace() {
		return this.getClass().getName();
    }
}
```

FactorId.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class FactorId extends ELSEInstruction {

	private static String rule = "			<Factor> ::= <id>";
	private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

	public static void initialize() {  // static initialization
		GRAMMAR.addRule(syntaxRule,
				new FactorId(),
				new GenericParser(syntaxRule.lhs()));
	}
	public FactorId(){ }

	@Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
		FactorId result = new FactorId();
		result.il = il;
		return result;
	}

	@Override
	public Value eval(EnvironmentVal env) {
```

```
                String id = il.get(0).eval(env).toString();
            Value value = env.find(id);
                return value;
        }
    public String trace() {
                return this.getClass().getName();
    }
}
```

FactorNo.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class FactorNo extends ELSEInstruction {

        private static String rule = "                   <Factor> ::= <no>";
        private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

        public static void initialize() {  // static initialization
                GRAMMAR.addRule(syntaxRule,
                                new FactorNo(),
                                new GenericParser(syntaxRule.lhs()));
        }
        public FactorNo(){ }
        public FactorNo(FactorNo a)
        {
        this.il = new InstructionList();
                for (Instruction instruction: a.il) {
                String className = instruction.getClass().getName();
                this.il.add(instr);
                }
        }
        @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
                FactorNo result = new FactorNo();
                result.il = il;
                return result;
        }

        @Override
        public Value eval(EnvironmentVal env) {
                return il.getFirst().eval();
        }
    public String trace() {
                return this.getClass().getName();
    }
}
```

Macro.java

```
package ELSE;
import static ELSE.ELSEInstruction.GRAMMAR;
import parsing.Instruction;
import parsing.InstructionList;
import parsing.SyntaxRule;
import parsing.*;
import runtime.*;
import java.util.Iterator;
import scanner.GenericScanner;
/**
 *
 * @author Wendy
 */
public class Macro extends ELSEInstruction{
    private String rule;
    private SyntaxRule syntaxRule;
    private String macroBody;
    private Instruction skeletonTree;
    public Macro ()
    {}

    //private Instruction parseMacro();
    public Macro (String macroRule, String macroBody)
    {
        //macro rule: generate grammar rule
        rule = macroRule;
        syntaxRule = new SyntaxRule(GRAMMAR, rule);
        GRAMMAR.addRule(syntaxRule, this,
                        new GenericParser(syntaxRule.lhs()));

        this.macroBody = macroBody;

        //macro body: generate parse tree
        // Now create a scanner
        Grammar grammar = ELSEInstruction.getGrammar();
        GenericScanner scanner;
        scanner = new GenericScanner(grammar, macroBody);
        scanner.get();   // Prime the pump with the first token
        skeletonTree =
grammar.parse(grammar.createSymbol(syntaxRule.lhs().toString()), scanner);
    }

    public Instruction createInstruction(InstructionList il) {
             Macro result = new Macro();
             result.macroBody = macroBody;
             result.skeletonTree = skeletonTree;
             result.rule = rule;
             result.syntaxRule = syntaxRule;
             result.il = il;

             return result;
    }
    public Value eval(EnvironmentVal env) {
             Instruction macroTree = copy(skeletonTree);
             expand(macroTree, il);
             return macroTree.eval(env);
       }
}
```

MultOpDiv.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;
```

```
/**
 * @author Wendy
 */

public class MultOpDiv extends ELSEInstruction {

       private static String rule = "               <MultOp> ::= /";
       private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

       public static void initialize() {  // static initialization
              GRAMMAR.addRule(syntaxRule,
                              new MultOpDiv(),
                              new GenericParser(syntaxRule.lhs()));
       }
       public MultOpDiv(){ }

       @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
              MultOpDiv result = new MultOpDiv();
              result.il = il;
              return result;
       }

       @Override
       public Value eval(EnvironmentVal env) {
              ListVal list = new ListVal();
              list.insert(new StringVal(trace()));
              for (Instruction instruction: il) {
                     Value sn = instruction.eval(env);
                     list.insert(sn);
              }
              return new ListVal(list);
       }
    public String trace() {
              return this.getClass().getName();
    }
}
```

MultOpMult.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class MultOpMult extends ELSEInstruction {

       private static String rule = "               <MultOp> ::= *";
       private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

       public static void initialize() {  // static initialization
              GRAMMAR.addRule(syntaxRule,
                              new MultOpMult(),
                              new GenericParser(syntaxRule.lhs()));
       }
       public MultOpMult(){ }
       @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
              MultOpMult result = new MultOpMult();
              result.il = il;
              return result;
       }

       @Override
       public Value eval(EnvironmentVal env) {
              ListVal list = new ListVal();
              list.insert(new StringVal(trace()));
              for (Instruction instruction: il) {
                     Value sn = instruction.eval(env);

                     list.insert(sn);
              }
              return new ListVal(list);
       }
    public String trace() {
              return this.getClass().getName();
    }
}
```

StmtAssign.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class StmtAssign extends ELSEInstruction {

       private static String rule = "              <Statement> ::=
<Assignment>";
       private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

       public static void initialize() {  // static initialization
              GRAMMAR.addRule(syntaxRule,
                             new StmtAssign(),
                             new GenericParser(syntaxRule.lhs()));
       }
       public StmtAssign(){ }

       @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
              StmtAssign result = new StmtAssign();
              result.il = il;
              return result;
       }

       @Override
       public Value eval(EnvironmentVal env) {
              return il.get(0).eval(env);
       }
    public String trace() {
            return this.getClass().getName();
    }
}
```

StmtList.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class StmtList extends ELSEInstruction {

       private static String rule = "                <StatementList> ::=
<Statement>";
       private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

       public static void initialize() {  // static initialization
              GRAMMAR.addRule(syntaxRule,
                              new StmtList(),
                              new GenericParser(syntaxRule.lhs()));
       }
       public StmtList(){ }

       @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
            StmtList result = new StmtList();
            result.il = il;
            return result;
       }

       @Override
       public Value eval(EnvironmentVal env) {
              return il.get(0).eval(env);
       }
    public String trace() {
            return this.getClass().getName();
    }
}
```

StmtListRecur.java

```
package ELSE;

import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class StmtListRecur extends ELSEInstruction {

       private static String rule = "           <StatementList> ::= <Statement>
<StatementList>";
```

```
        private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);


        public static void initialize() {  // static initialization
                GRAMMAR.addRule(syntaxRule,
                               new StmtListRecur(),
                               new GenericParser(syntaxRule.lhs()));
        }
        public StmtListRecur(){ }

        @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
                StmtListRecur result = new StmtListRecur();
                result.il = il;
                return result;
        }

        @Override
        public Value eval(EnvironmentVal env) {
        Value value= il.get(0).eval(env);
                il.get(1).eval(env);
                return value;
        }
    public String trace() {
                return this.getClass().getName();
    }
}
```

TEST_ELSE.java

```
package ELSE;
import intergen.FileUtils;
import intergen.Intergen;
import static intergen.Intergen.author;
import static intergen.Intergen.classList;
import static intergen.Intergen.className;
import static intergen.Intergen.email;
import static intergen.Intergen.input;
import static intergen.Intergen.packageName;
import static intergen.Intergen.rule;
import static intergen.Intergen.startSymbol;
import static intergen.Intergen.workingDirName;
import static intergen.Intergen.year;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.logging.Level;
import java.util.logging.Logger;
import scanner.*;
import parsing.*;
import runtime.*;

/**
 *
 * @author  Wendy
 */

public class TEST_ELSE {
    public static void run(Instruction instr) {
        instr = instr.createInstruction(null);
        instr.exec();
    }
```

```
        static void print(Value val, int indent) {
        if (val instanceof ListVal) {
                        indent = indent + 4;
                        ListVal list = (ListVal) val;
                        list.reset();
                        while (list.on()) {
                        Value v = list.get();
                        print(v, indent);
                        list.move();
                        }
        }
        else {
                        for (int i=0; i < indent; i++) System.out.print(" ");
                        System.out.println(val);
        }
        }
        private static void loadClasses() {
                StmtList.initialize();
                StmtListRecur.initialize();
                StmtAssign.initialize();
                Assign.initialize();
                ExpTerm.initialize();
                ExpTermAddOpExp.initialize();
                TermFactor.initialize();
                TermFactorMultOpTerm.initialize();
                FactorExp.initialize();
                AddOpAdd.initialize();
                AddOpSub.initialize();
                MultOpMult.initialize();
                MultOpDiv.initialize();
                FactorId.initialize();
                FactorNo.initialize();
      }
      public static void loadMacro()
      {
                String macroRule = "";
                String macroBody = "";
                boolean preRuleLine = false;
                boolean preBodyLine = false;
                try {
                String workingDir = System.getProperty("user.dir");
                java.util.Scanner in = new java.util.Scanner(new File(workingDir +
"/src/ELSEMacro.grm"));

                in.skip(" *");
                // First line is the package name in the form
              // "package Packagename"
                while (in.hasNext()) {
                        String type = in.nextLine();
                        if (type.startsWith("//")) {
                                in.nextLine();
                                continue;
                        }else {
                                if (type.equalsIgnoreCase("#MACRO_RULE"))
                                {
                                preRuleLine = true;
                                preBodyLine = false;
                                continue;
                                }
                                else if (type.equalsIgnoreCase("#MACRO_BODY"))
                                {
```

```
                        preRuleLine = false;
                        preBodyLine = true;
                        continue;
                        }
                        else if (type.equalsIgnoreCase("#MACRO_END"))
                        {
                        new Macro(macroRule, macroBody);
                        macroRule = "";
                        macroBody = "";
                        preRuleLine = false;
                        preBodyLine = false;
                        continue;
                        }
                        else if (preRuleLine == true)
                        {
                        macroRule += " " + type;
                        }
                        else if (preBodyLine == true)
                        {
                        macroBody += " " + type;
                        }
                    }
              }
              } catch (FileNotFoundException ex) {
              Logger.getLogger(Intergen.class.getName()).log(Level.SEVERE, null,
ex);
              }
    }
      public static void main(String[] args) {
      // load grammar rules
              loadClasses();

       // load macro rules
              loadMacro();

              // set start symbol
              ELSEInstruction.getGrammar().setStart("<StatementList>");

              // Now create a scanner
              Grammar grammar = ELSEInstruction.getGrammar();
              // set input sentences
              GenericScanner scanner = new GenericScanner(grammar, "a := 1 b :=
++ a bump b square b");
              scanner.get();

              // parsing
              Instruction instr = grammar.parse(scanner);

              EnvironmentVal env = new EnvironmentVal();
              // evaluating
              if (instr != null) {

             System.out.println("Instructions:\n"+instr.toString()+"\nResults:");
              Value list = instr.eval(env);
              } else {
              System.err.println("Syntax error in your input.");
              }
    }
}
```

TermFactor.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class TermFactor extends ELSEInstruction {

	private static String rule = "              <Term> ::= <Factor>";
	private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

	public static void initialize() {  // static initialization
		GRAMMAR.addRule(syntaxRule,
		                new TermFactor(),
		                new GenericParser(syntaxRule.lhs()));
	}
	public TermFactor(){ }

	@Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
		TermFactor result = new TermFactor();
		result.il = il;
		return result;
	}

	@Override
	public Value eval(EnvironmentVal env) {
		return il.get(0).eval(env);
	}
    public String trace() {
		return this.getClass().getName();
    }
    public String toString() {
		return il.getFirst().toString();
    }
}
```

TermFactorMultOpTerm.java

```
package ELSE;
import parsing.*;
import runtime.*;
import java.util.Iterator;

/**
 * @author Wendy
 */

public class TermFactorMultOpTerm extends ELSEInstruction {

	private static String rule = "    <Term> ::=  <Factor> <MultOp> <Term>";
	private static SyntaxRule syntaxRule = new SyntaxRule(GRAMMAR, rule);

	public static void initialize() {  // static initialization
		GRAMMAR.addRule(syntaxRule,
                              new TermFactorMultOpTerm(),
                              new GenericParser(syntaxRule.lhs()));
        }
        public TermFactorMultOpTerm(){ }

        @Override
    // Default to building a full parse tree
    public Instruction createInstruction(InstructionList il) {
                TermFactorMultOpTerm result = new TermFactorMultOpTerm();
                result.il = il;
                return result;
        }

        @Override
        public Value eval(EnvironmentVal env) {
                Value a = il.get(0).eval(env);
                Value b = il.get(2).eval(env);
        if (a instanceof DoubleVal && b instanceof DoubleVal){
                if (il.get(1) instanceof MultOpMult)
                                return new DoubleVal(((DoubleVal)a).getVal() *
((DoubleVal)b).getVal());
                if (il.get(1) instanceof MultOpDiv)
                                return new DoubleVal(((DoubleVal)a).getVal() /
((DoubleVal)b).getVal());
                }
                return null;
    }
    public String trace() {
                return this.getClass().getName();
    }
    public String toString() {
                if (il.get(1) instanceof MultOpMult)
                        return il.get(0).toString() + "*" + il.get(2).toString();
                if (il.get(1) instanceof MultOpDiv)
                        return il.get(0).toString() + "/" + il.get(2).toString();
                return null;
    }
}
```

Code in package "scanner"

AbstractBuffer.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */
package scanner;
import java.io.FileReader;


/**
 *
 * @author Larry Morell <morell@cs.atu.edu>
 */
public interface AbstractBuffer {
    AbstractBuffer copy();
```

```
  void setSource(String s);
  /**
*
* @param fr
* @param fn
*/
  void setSource(FileReader fr, String fn);
  String getFileName();

  char charAt(int i);

  char peek();  // the next character that get will return

  int currentPos();
  public void advance();

  int lineNumber();  // logical result
  int columnNumber(); // logical result

  boolean eof();

  boolean eoln();

  char get();

  String getBuffer();

  String getId();

  String getNumber();

  String getOperator();

  String getToDelimiter(String delim);

  String getToEoln();

  char nextChar();

  int size();

  void skipBlanks();
```

```
}
```

Buffer.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */
package scanner;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Buffer implements a buff which can be read from in various ways.
 * Abstract model:
 *    buff is a sequence of characters
 *    Current position is a position within, or beyond the buff
 *    Current character is the character from the buff referenced
 *       by the current position, if such a one exists.  If it does
 *       not then the the current character is the null character.
 * @author Larry Morell <lmorell@atu.edu>
 */
public class Buffer implements AbstractBuffer {

       // ----------------- Class-wide functions ----------------- //
       // These implement several functions that are also implemented in
Character
       // These assume that the characters are all ASCII and are therefore
simplified
       // accordingly, and (hopefully) will reduce runtime overhead when doing
these
       // simple tests.  This may not actuall be true since I don't know how the
       // comparison operators are implemented.  The aim here is to eliminate
the
       // number of function calls that are generated by calling
Character.isDigit(),
       // e.g., we have (at least) five function calls.  A simple test revealed
       // that isDigit() implemented as below took half the time.  However,
calling
       // any of these functions, about 10K times (a very large algorithm),
would be negligible overhead
       // of 10**-5 seconds, so this realling is not a big deal.
       /*
       public static boolean isAlphaNum(char ch) {
              /*
              if (ch >= 'A' && ch <= 'Z') {
              return true;
              }
              if (ch >= 'a' && ch <= 'z') {
              return true;
              }
              if (ch >= '0' && ch <= '9') {
              return true;
              }
              return false;

              return Character.isLetterOrDigit(ch);
       }

       static boolean isDigit(char ch) {
              // return (ch >= '0' && ch <= '9');
              return Character.isDigit(ch);
       }

       public static boolean isWhiteSpace(char ch) {
              // return (ch == ' ') || (ch == '\n') || (ch == '\t') || (ch ==
'\r');
              return Character.isWhitespace(ch);
       }
       */
       static boolean errorOnEOF = false;
       public static void setErrorOnEofAccess() {errorOnEOF = true;}
       // --------------- Instance variables --------------- //
       protected String buffer;   // the buff to be accessed
       protected String fileName; // the source file name
       protected int pos;           // position within the buff (0-based)
     protected int lineNo;     // the line within the buff (1-based)
       protected int colPos;      // the position within a line (1-based)
```

```
        //---------------- Constructors ---------------------//
        public Buffer() {
       this("");
        }
    /**
         *
         * @param string -- the string from which data is to be read
         */

        public Buffer(String string) {
               buffer = string;
               pos = 0;
               colPos= 1; // past end of line
               lineNo = 1; // past end of file
               fileName = "Source string";
        }
        /**
         * Constructor to assign
         * @param in -- the source file
         */
         public Buffer(FileReader in, String fileName){
          this("");
               StringBuilder buff = new StringBuilder();
               this.fileName = fileName;
               int count;
               char buf[] = new char[512];
               int offset = 0;
               try {
                      while ((count = in.read(buf,offset,512)) >= 0) {
                buff.append(buf,offset,count);
                      }
               } catch (IOException ex) {
                      Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE,
null, ex);
               }
               this.buffer = new String (buff);
        }
        /**
         * Copy constructor
         * @param buff -- the buff to be copied
         */
          public Buffer(Buffer buffer) {
```

```
        this.buffer =buffer.buffer;
        pos = buffer.pos;
        lineNo =  buffer.lineNo;
        colPos = buffer.colPos;
        fileName = buffer.fileName;
    }
    /**
     * Override this to allow copying a Buffer
     * @param b -- create a new instance of b
     * @return
     */
public static Buffer newInstance(Buffer b){
        return new Buffer(b);
    }
    // ----------------- Getters/Setters --------------------//
    /**
     * @return The buff currently in use.
     *
     */
    public String getBuffer() {
        return buffer;
    }

    /**
     * If the current position references a point in the current buff
     * then return the character at that position; otherwise return the
     * null character.
     */
    public char charAt(int i) {
        if (i < 0 || i > buffer.length()) {
            return ('\0');  // return the null character
        }
        return buffer.charAt(i);  // Fake out the compiler
    }

    /** Return the number of characters in the buff
     */
    public int size() {
        return buffer.length();
    }

    /**
```

```
         * @return If the current position is within the buff bounds, return
         * that position; otherwise return the length of the buff.
         */
        public int currentPos() {
               return pos;
        }

        /**
         * @return If the current position is with the bounds of the buff, return
         * false, otherwise, return true
         */
        public boolean eof() {
               return (pos >=  buffer.length());
        }

        /**
         * @return
         * <table>
         * <tr>
         *    <td><b>Condition</b> </td>
         *    <td> <b>Return</b> </td>
         * </tr>
         * <tr>
         *     <td>Current position within the buff and the current char is a \n
or \r</td>
         *     <td>true </td>
         * </tr>
         * <tr>
         *    <td> Otherwise</td>
         *    <td> false</td>
         * </tr>
         * </table>
         */
        public boolean eoln() {
               return pos >= buffer.length() || buffer.charAt(pos) == '\n'
                               || buffer.charAt(pos) == '\r';
        }

        /**
         * @return
         * <table>
         * <tr>
         *    <td><b>Condition</b> </td>
         *    <td> <b>Return</b> </td>
         *    <td> <b>Side effects</b></td>
         * </tr>
         * <tr>
         *     <td>Current position is within the buff</td>
         *     <td>The associated char</td>
         *     <td>Increments the current position in the buff</td>
         * </tr>
         * <tr>
         *    <td>errorOnEof is set</td>
         *    <td>Nothing</td>
         *    <td>Error message  </td>
         * </tr>
         * <tr>
         *    <td>errorOnEof is not set (default</td>
         *    <td>null char</td>
         *    <td></td>
         * </tr>

         * </table>
         */
        public char get() {
                if (pos < 0 || pos >= buffer.length()) {
                        if (errorOnEOF) {
                        System.err.println("Attempt to read past end of buffer at
position" + pos);
                        System.exit(-1);
                        }
                        else
                                return '\0';
                }

                pos++;
                if (buffer.charAt(pos-1) == '\n') {lineNo++; colPos = 1;}
                else {colPos++;}
                return buffer.charAt(pos - 1);  // return the char just advanced
past
        }
         /**
         * <table>
         * <tr>
```

```
     *    <td><b>Condition</b> </td>
     *    <td> <b>Behavior</b> </td>
     * </tr>
     * <tr>
     *     <td>Current position is within the buff</td>
     *     <td>Current position is one greater</td>
     * </tr>
     * <tr>
     *    <td>Current position is not with the buff</td>
     *    <td>Current position is unchanged</td>
     * </tr>
     * </table>
 */
public void advance() {
          if (pos >=0 && pos < buffer.length()) {
                pos++;
                if (buffer.charAt(pos-1) == '\n') {lineNo++; colPos = 1;}
                else {colPos++;}
    }
    }

    /**
     * @return
     * <table>
     * <tr>
     *    <td><b>Condition</b> </td>
     *    <td> <b>Return</b> </td>
     * </tr>
     * <tr>
     *     <td>Current position is within the buff</td>
     *     <td>The char at that position</td>
     * </tr>
     * <tr>
     *    <td>Otherwise</td>
     *    <td>Nothing</td>
     *    <td>Error message</td>
     * </tr>
     * </table>
     */
    public char peek() {
          if (pos < 0 || pos >= buffer.length()) {
```

```
                    System.err.println("Attempt to read past end of buffer at
position" + pos);
                    System.exit(-1);
              }
              return buffer.charAt(pos);
       }

       @SuppressWarnings("empty-statement")
       public void skipBlanks() {

              while (!eof() && Character.isWhitespace(buffer.charAt(pos))) {
                     advance();
              }
       }

       /**
        * @return
        * <table>
        * <tr>
        *    <td><b>Condition</b> </td>
        *    <td> <b>Return</b> </td>
        *    <td> <b>Side effects </b> </td>
        * </tr>
        * <tr>
        *     <td>Whitespace*No occurs next in the buff</td>
        *     <td>No</td>
        *     <td>Current position is immediately after No</td>
        * </tr>
        * <tr>
        *    <td>Otherwise</td>
        *    <td>""</td>
        * </tr>
        * </table>
        */
       public String getNumber() {
              StringBuilder sb = new StringBuilder();
      skipBlanks();
              while (pos < buffer.length() && buffer.charAt(pos) >= '0' &&
buffer.charAt(pos) <= '9') {
                     sb.append(get());
              }
```

```
            if (!eof() && buffer.charAt(pos) == '.') {
                   sb.append('.');
            advance();
                   while (pos < buffer.length() && buffer.charAt(pos) >= '0'
&& buffer.charAt(pos) <= '9') {
                          sb.append(get());
                   }
            }
            return new String(sb);
     }

     public String getId() {
            StringBuilder sb = new StringBuilder();
            skipBlanks();
            if (!eof() && Character.isLetter(buffer.charAt(pos))) {
                   while (!eof() &&
Character.isLetterOrDigit(buffer.charAt(pos))) {
                          sb.append(buffer.charAt(pos));
                          advance();
                   }
            }
            return new String (sb);
     }

     /**
      * An Operator is any sequence of characters that begin with none of
      * (whitespace, digit, or letter) and terminates at the first
      * (whitespace, digit, or letter) or eof(), whichever occurs first.
      * @return
      * <table>
      * <tr>
      *    <td><b>Condition</b> </td>
      *    <td> <b>Return</b> </td>
      *    <td> <b>Side effects </b> </td>
      * </tr>
      * <tr>
      *     <td>Whitespace*Operator appears next in buff</td>
      *     <td>Operator</td>
      *     <td>Current position follows after Operator</td>
      * </tr>
      * <tr>
      *    <td>Otherwise</td>
```

```
         *    <td>None</td>
         *    <td>Error message</td>
         * </tr>
         * </table>
         */
        public String getOperator() {
              StringBuilder sb = new StringBuilder();
              skipBlanks();
              String singletons = "()[],;";
              while (!eof()
                             && ! Character.isWhitespace(buffer.charAt(pos))
                             && ! Character.isLetterOrDigit(buffer.charAt(pos))
                             && ! singletons.contains(""+buffer.charAt(pos)))
          {
                    sb.append(get());
              }
              if (!eof()
                             && sb.length() == 0
                             && singletons.contains(""+buffer.charAt(pos))) {
                    sb.append(get());
              }
              return new String(sb);
        }

        /**
         * @return
         * <table>
         * <tr>
         *    <td><b>Condition</b> </td>
         *    <td> <b>Return</b> </td>
         *    <td> <b>Side effects </b> </td>
         * </tr>
         * <tr>
         *     <td>Current position within buff</td>
         *     <td>The sequence of characters from the current position
         *           up to, but not including, the next end-of-line, or end-of-
file, whichever
         *           occurs first</td>
         *     <td>Current position is set after the termination point </td>
         * </tr>
         * <tr>
         *    <td>Otherwise</td>
```

```
 *    <td>""</td>
 *    <td>Current position is set outside the buff</td>
 * </tr>
 * </table>
 */
public String getToEoln() {
       StringBuilder sb = new StringBuilder();
       while (!eof() && !eoln()) {
              sb.append(get());
       }
       advance();
       return new String(sb);
}

/**
 * @return
 * <table>
 * <tr>
 *    <td><b>Condition</b> </td>
 *    <td> <b>Return</b> </td>
 * </tr>
 * <tr>
 *     <td>There is a character in the buff after the current pos</td>
 *     <td>That character</td>
 * </tr>
 * <tr>
 *    <td>Otherwise</td>
 *    <td>Null character</td>
 * </tr>
 * </table>
 */
public char nextChar() {
       if (eof() || pos == buffer.length() - 1) {
              return '\0';
       }
       return buffer.charAt(pos + 1);
}

/**
 * @return
 * <table>
 * <tr>
```

```
         *    <td><b>Condition</b> </td>
         *    <td> <b>Return</b> </td>
         *    <td> <b>Side effects </b> </td>
         * </tr>
         * <tr>
         *     <td>delim occurs in the buff at or after the current position</td>
         *     <td>the sequence of characters to to that delimeter</td>
         *     <td>Current position is placed after delim</td>
         * </tr>
         * <tr>
         *    <td>Otherwise</td>
         *    <td></td>
         *    <td>Error message</td>
         * </tr>
         * </table>
         */
        public String getToDelimiter(String delim) {

                StringBuilder sb = new StringBuilder();
                int endPos = buffer.indexOf(delim, pos);
                if (endPos >= buffer.length()) {
                        System.err.println("Error, attempting to find " + delim +
"; it was not found.");
                }
                // Slow append one char at at time so line and character
positioning
                // will be done by get()
                for (int i = pos; i < endPos; i++ ) sb.append(get());
                for (int i =0; i < delim.length()-1; i++) advance();  // skip
delim
                advance();
                // System.err.println("gotodelim returns '" + sb + "'");
                return new String(sb);
        }

        public AbstractBuffer copy() {
                return new Buffer(this);
        }

        public void setSource(String s) {
                buffer = s;
                pos = 0;
```

```
		colPos= 1; // past end of line
		lineNo = 1; // past end of file
		fileName ="String source";
	}

	public void setSource(FileReader in, String fileName) {
		this.fileName = fileName;
		StringBuilder b = new StringBuilder();
		int count;
		char buf[] = new char[512];
		int offset = 0;
		try {
			while ((count = in.read(buf,offset,512)) >= 0) {
		 b.append(buf,offset,count);
			}
			buffer = new String(b);
		} catch (IOException ex) {
			Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE,
null, ex);
		}
		pos = 0;
		colPos= 1; // past end of line
		lineNo = 1; // past end of file
	}
		public int lineNumber() {
		return lineNo;
	}

	public int columnNumber() {
		return colPos;
	}
	public String getFileName() {
		return fileName;
	}
	public static void check(boolean b, String msg) {
		if (!b)
			System.err.println(msg +": failed!");
	   //else
			//System.err.println(msg + ": succeeded");
	}
```

```
       public static void main (String[] args) {
          // test to see how buffers work
              // Check an empty buff
              Buffer b = new Buffer ("");
      check (b.eoln(), "eoln for an empty buffer should be true")        ;
              check (b.eof(), "eof for an empty buffer should be true");

              //              0123456 789012345678901234567
          b = new Buffer ("This \n    has 14 characters");
              //              123456 123456789012345678901
              check (b.get() == 'T', "First get should return the first char");
       check (b.get() == 'h', "Second get should return the second char");
              b.skipBlanks();
       check (b.get() == 'i', "Third get should return the third char");
              b.get(); b.skipBlanks();
              check (b.get() == 'h', "skipBlanks should skip whitespace");
              check (b.getId().equals("as"), "getId() should return an id");
      check (b.getId().equals(""), "getId should return empty string if no
id");
              //System.out.println("gn: '" +b.getNumber() +"'");
              check (b.getNumber().equals("14"),"getNo should return an
number");
      check(b.currentPos()== 16 , "currentPos() should return the buffer
position");
      check(b.columnNumber()== 11 , "columnNumber() should return the col
position");
      check(b.lineNumber()== 2, "lineNumber() should return the current
position");
      check(b.getFileName().equals("Source string"), "Filename for string is
correct");

      b = new Buffer("Here we go again = /= *8&^%$");

              check(b.getId().equals("Here"),"getId skips leading blanks");
              check(b.getId().equals("we"),"getId skips leading blanks");
              check(b.getId().equals("go"),"getId skips leading blanks");
              check(b.getId().equals("again"),"getId skips leading blanks");
              check(b.getOperator().equals("="), "getOperator obtains string");
              check(b.getOperator().equals("/="), "getOperator obtains string");
              check(b.getOperator().equals("*"), "getOperator obtains string");
              check(b.getNumber().equals("8"), "getNumber obtains string");
```

```
             check(b.getOperator().equals("&^%$"), "getOperator obtains
string");
             check(b.getOperator().equals(""), "getOperator obtains string");

             b = new Buffer("   And\n    again");
             check(b.getToEoln().equals("   And"),"getToEoln() does not include
newline");
             check(b.getToEoln().equals("    again"),"getToEoln() skips
newline");

//           b = new Buffer("abc x  $543.79");
             b.setSource("abc x  $543.79");
             check (b.getToDelimiter("c").equals("ab"), "getToDelim does not
include delim");

             check (b.getToDelimiter("$").equals(" x  "), "getToDelim does not
include delim");
                                                  //  ' x  '
             check (b.peek() == '5', "currentChar() should return the that
would be fetched by get()");
             check (b.nextChar() == '4', "nextChar() should return the char
that follows that which will be fetched by get()");
             check (b.size() == "abc x  $543.79".length(),"size() should return
the number of chars in the buffer");
                          try {
                    FileWriter out = new FileWriter("temp");
                    out.write("abc x  $543.79");
                    out.close();

             } catch (IOException ex) {
                    Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE,
null, ex);
             }
             FileReader in;
             try {
                    in = new FileReader("temp");
                    b.setSource(in,"temp");
             } catch (FileNotFoundException ex) {
                    Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE,
null, ex);
             }
             check (b.getToDelimiter("c").equals("ab"), "getToDelim does not
include delim");

             check (b.getToDelimiter("$").equals(" x  "), "getToDelim does not
include delim");
                                                      //  ' x  '
             check (b.peek() == '5', "currentChar() should return the that
would be fetched by get()");
             check (b.nextChar() == '4', "nextChar() should return the char
that follows that which will be fetched by get()");
   }
}
```

BufferWrapper.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package scanner;

import scanner.AbstractBuffer;

/**
 * Decorator pattern -- descendents of this class will add functionality to
 *    Buffers
 * @author Larry Morell
 */
public abstract  class  BufferWrapper implements  AbstractBuffer {

       // ---------------- Instance variables ----------------//
       public AbstractBuffer buffer;  //

       // ---------------- Constructors ----------------------//
       BufferWrapper() { buffer = null;}
       // ---------------- Getters/Setters -------------------//
       public String getFileName() {return buffer.getFileName();}

       // ---------------- Trivial wrap methods  ------------//

       @Override
       public char charAt(int i) {
```

```
                return buffer.charAt(i);
        }
        @Override
        public char peek() {
                return buffer.peek();
        }
        @Override
        public int currentPos() {
                return buffer.currentPos();
        }
        @Override
        public boolean eof() {
                return buffer.eof();
        }
        @Override
        public boolean eoln() {
                return buffer.eoln();
        }
        @Override
        public char get() {
                return buffer.get();
        }
        @Override
        public String getBuffer() {
                return getBuffer();
        }
        @Override
        public String getId() {
                StringBuilder sb = new StringBuilder();
                skipBlanks();
                if (!buffer.eof() && ( Character.isLetter(buffer.peek()) ||
buffer.peek() == '_')) {
                        while (!eof() && ( Character.isLetterOrDigit(buffer.peek())
|| buffer.peek() == '_')) {
                                sb.append(buffer.peek());
                                buffer.advance();
                        }
                }
                return new String (sb);
        }

        @Override
```

```
	public String getNumber() {
		return buffer.getNumber();
	}
	@Override
	public String getOperator() {
		return  buffer.getOperator();
	}
	@Override
	public String getToDelimiter(String delim) {
		return buffer.getToDelimiter(delim);
	}
	@Override
	public String getToEoln() {
		return buffer.getToEoln();
	}
	@Override
	public char nextChar() {
		return buffer.nextChar();
	}
	@Override
	public int size() {
		return buffer.size();
	}
	@Override
	public void skipBlanks() {
		while (!eof() && Character.isWhitespace(buffer.peek())) {
			advance();
		}
	}
	@Override
	public int lineNumber() {
		return buffer.lineNumber();
	}
    @Override
	public int columnNumber() {
		return buffer.columnNumber();
	}
	public void advance() {
		buffer.advance();
	}
}
```

GenericScanner.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */
package scanner;

import java.io.FileReader;
import parsing.Grammar;
import parsing.Symbol;

/**
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class GenericScanner implements Scanner {

       // ---------------- Static variables ------------------//
       // ---------------- Static initialization -------------//
       // ---------------- Static methods --------------------//
       // ---------------- Instance variables ----------------//
       StackedBuffer buffer;   // The input sourse
       Grammar grammar;          // The grammar this scanner is written for
       GenericScanner saved;
       Token token;               // The most recent token returned by get()
       // ---------------- Constructors ----------------------//
       public GenericScanner(Grammar g) {
              grammar = g;
              buffer = new StackedBuffer();
              saved = null;
       }

       public GenericScanner(Grammar g, FileReader fr, String fn) {
              this(g);
              buffer.setSource(fr, fn);
       }

       public GenericScanner(Grammar g, String src) {
              this(g);
              buffer.setSource(src);
       }
       // ---------------- Getters/Setters -------------------//
	// ---------------- Other member functions ------------//
	public Token get() {
		// Strategy is to skip whitespace, branch on the encountered
character,
		// invoke the appropriate buffer routine, convert if necessary
		// NOTE: the variable ch throughout this code denotes the
character
		// the character that will be read by the next call to get()
		Token t;
		String value;
		Symbol symbol;
		buffer.skipBlanks(); // align after ws
		int lineNo = buffer.lineNumber();
		int charPos = buffer.currentPos();

		if (buffer.eof()) { // set up an EOF symbol
			symbol = Symbol.createSymbol(grammar, "");  // creates an
EOF symbol
			t = new Token(symbol, buffer, lineNo, charPos);
		}
		else {
			// for each of these we must create a symbol, then form a
token from it
			char ch = buffer.peek();
			if (Character.isLetter(ch) || ch == '_') {
				value = buffer.getId();
				symbol = Symbol.createSymbol(grammar, value);
				boolean b = symbol.isKeyword();
				if (symbol.isKeyword())
					t = new KeywordToken(symbol, buffer, lineNo,
charPos);
				else
					t = new IdToken(symbol, buffer, lineNo,
charPos);
			}
			else if (Character.isDigit(ch)) {
				value = buffer.getNumber(); // get the string version
of the number
				symbol = Symbol.createSymbol(grammar, value);
				t = new NumberToken(symbol, buffer, lineNo, charPos);
			}
```

```
                    else if (ch == '"' || ch == '\'') {  // it is a literal
string
                        char terminator = ch;
                        StringBuilder val = new StringBuilder();
                        // val.append(terminator);
                        buffer.advance();
                        ch = buffer.peek(); // get the first char after the
quote

                        while (!buffer.eof() && ch != terminator) {

                            if (ch == '\\') {
                                buffer.advance();
                                ch = buffer.peek();
                                if (ch == 'n') {
                                    ch = '\n';
                                }
                                else if (ch == 't') {
                                    ch = '\t';
                                }
                                else if (ch == 'r') {
                                    ch = '\r';
                                }
                                // if the character is anything else,
including a '\'
                                // then just keep it
                            }
                            val.append(ch);
                            buffer.advance();
                            ch = buffer.peek();
                        }
                        if (ch != terminator) {
                            System.err.println("Double quotes are not
balanced.");
                            System.err.println("There was no matching
quote("
                                     + terminator + ") "
                                     + "for the quote found on or
before line "
                                     + lineNo + '.');
                            System.err.println("The mistake could well be
much earlier in the algorithm.");
```

```
                              System.err.println("Check all pairs of quotes
to find the mistake.");

                              System.exit(1);
                        }
                        // val.append(ch); // Append the "
                        ch = buffer.get();
                        symbol = Symbol.createSymbol(grammar, new
String(val));
                        t = new StringToken(symbol, buffer, lineNo, charPos);
                  }

                  else if (ch == '/') {
                        int tempCounter = 0;
                        ch = buffer.get();   // toss it
                        ch = buffer.peek(); // peek at the next character
                        if (ch == '/') {  //  a //-comment found
                              buffer.getToEoln();
                              tempCounter = 1;
                              t = get();  // Note the recursion
                        }
                        else if (ch == '*') {  // sm: handle multi-line /*
comments */
                              boolean mlComment = true;
                              ch = buffer.get();
                              while (!buffer.eof() && mlComment) {
                                    if (ch == '*') {
                                          ch = buffer.get();
                                          if (ch == '/') {
                                                mlComment = false;   //
Muntha error

                                                ch = buffer.get();
                                          }
                                    }
                                    else {
                                          ch = buffer.get();
                                    }
                              }
                              if (mlComment) {
                                    System.err.println("'/*' comment not
closed with '*/'");
```

```
                                        symbol = Symbol.createSymbol(grammar,
"");
                                        t = new Token(symbol, buffer, lineNo,
charPos);
                                 }
                                 else {
                                        t = get();  // recursive call ...  go
get the token after the comment
                                 }
                          }
                          else {
                                 String operator = "/";
                                 symbol = Symbol.createSymbol(grammar,
operator);
                                 t = new OperatorToken(symbol, buffer, lineNo,
charPos);
                          }
                   }
                   else if (ch == '#') { // single line comment
                          buffer.getToEoln();
                          t = get(); // note the recursion
                   }
                       else if (ch == '$') {//place holder for macro tree
                           buffer.advance();
                           value = buffer.getId();
                           symbol = Symbol.createSymbol(grammar, value);
                           t = new PlaceholderToken(symbol, buffer, lineNo,
charPos);
                       }
                   else { // not a digit, letter, quote, or comment, must be
an operator
                          value = buffer.getOperator();
                          symbol = Symbol.createSymbol(grammar, value);
                          t = new OperatorToken(symbol, buffer, lineNo,
charPos);
                   }
            }
      token = t; // save the token for calls to current()
             return t;


       }
```

```
// Future modification: note that grammar is not being copied here.
// This means that if the grammar is modified after the copy is made,
// then the copy will have its grammar modified as well.  This would only
// be of consequence if in the process of backtracking we needed to undo,
// say, the creation of a grammar rule.  This appears to be unlikely,
// but this note will serve as a reminder that someday you will be bit
// by this if you're not careful.
// To fix it we need some way to copy a grammar.  This would require
// a massive amount of computation, so it is being avoided here.

public Scanner copy() {
        GenericScanner gs = new GenericScanner(grammar);
        gs.buffer = (StackedBuffer) buffer.copy();
        return gs;
}

public void mark() {
        saved = (GenericScanner) copy();
}

public void reset() {
        grammar = saved.grammar;
        buffer = saved.buffer;
        saved = null;
}

public boolean eof() {
        buffer.skipBlanks();
        return buffer.eof();

}

// Test the scanner
public static void main(String[] args) {
        // First set up the grammar and some simple rules
        Grammar grammar = Grammar.buildGrammar();
        Token t;
        String[] testInput = new String[]{
                "+ - += =+ /- -/ ", // various operators
                "a b c move def g", // ids
                "17 17.3 49.8 0 12345", // numbers
                "// This is a comment\n1 2 3", // single-line comment
```

```
                    "#  this is a comment\na b c", // single-line comment
                    "/* multi\nline\n  comment*/ with other ", // multi-line
comment
                    "'a string'",
                    "'a string\n across several lines '",
                    "\"a string\"",
                    "\"a string \n across \n several lines\"",
             };
             for (int i = 0; i < testInput.length; i++) {
                    String source = testInput[i];
                    System.out.println("Source =" + source);

                    GenericScanner scanner = new GenericScanner(grammar,
source);

                    do {
                           t = scanner.get();
                           System.out.println(t);
                    } while (!scanner.eof());
             }
      }

      public Token current() {
             return this.token;
      }
}
```

IdToken.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package scanner;

import parsing.Symbol;
import scanner.Token;

/**
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
```

```
public class IdToken extends Token {

   // ---------------- Instance variables ----------------//
   String value;  // intentional shadowing.  The inherited value must be "<id>"
       // ---------------- Constructors ----------------------//
   public IdToken (Symbol s, AbstractBuffer b, int lineno, int charpos) {
              super(s,b,lineno,charpos);
       value = s.getValue();
              symbol = Symbol.createSymbol(s.getGrammar(), "<id>");  // set the
inherited value
       }

       // ---------------- Getters/Setters -------------------//
    public String getId() {return value;}
       // ---------------- Other member functions ------------//
       @Override
       public String toString () {return "[" + value + "]"; }
}
```

KeywordToken.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package scanner;

import parsing.Symbol;
import scanner.Token;

/**
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class KeywordToken extends Token {

    // ---------------- Instance variables ----------------//

    // ---------------- Constructors ----------------------//
    public KeywordToken (Symbol s, AbstractBuffer b, int lineno, int charpos) {
             super(s,b,lineno,charpos);
    }
	// ---------------- Getters/Setters -------------------//
	public String getKeyword () {return value;}
	// ---------------- Other member functions ------------//
	@Override
	public String toString () {return "[" + value + "]"; }
}
```

NumberToken.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package scanner;

import parsing.Symbol;

/**
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class NumberToken extends Token {

	// ---------------- Instance variables ----------------//
	protected double number;
	String value;
	// ---------------- Constructors ----------------------//
	public NumberToken (Symbol s, AbstractBuffer b, int lineno, int charpos)
{
		super(s,b,lineno,charpos);
		value = s.getValue();
		number = new Double(value);  // convert to a double
		symbol = Symbol.createSymbol(s.getGrammar(),"<no>");
	}



	// ---------------- Getters/Setters -------------------//
	public double getNumber() {return number; }
	// ---------------- Other member functions ------------//
	@Override
```

```
       public String toString(){
              return "[" +value + "]";
    }
}
```

OperatorToken.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package scanner;

import parsing.Symbol;
import scanner.Token;

/**
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class OperatorToken extends Token {

       // ---------------- Instance variables ----------------//
     // None.  The operator is stored in the symbol
       // ---------------- Constructors ----------------------//
       public OperatorToken (Symbol s, AbstractBuffer b, int lineno, int
charpos) {
              super(s,b,lineno,charpos);
       }

       // ---------------- Getters/Setters -------------------//
       public String getOperator() {return symbol.getValue(); }
       // ---------------- Other member functions ------------//
}
```

PlaceholderToken.java

```
/*
 *  Copyright (C) 2016 Wendy Wan<xwan@cs.atu.edu>
 */

package scanner;
```

```
import parsing.Symbol;
import scanner.Token;

/**
 *
 * @author Wendy Wan (xwan@atu.edu)
 */
public class PlaceholderToken extends Token {

   // ---------------- Instance variables ----------------//
   String value;  // intentional shadowing.  The inherited value must be
"<PlaceHolder>"
       // ---------------- Constructors ----------------------//
   public PlaceholderToken (Symbol s, AbstractBuffer b, int lineno, int
charpos) {
              super(s,b,lineno,charpos);
       value = s.getValue();
              symbol = Symbol.createSymbol(s.getGrammar(), "<PlaceHolder>");  //
set the inherited value
       }

       // ---------------- Getters/Setters -------------------//
    public String getId() {return value;}
       // ---------------- Other member functions ------------//
       @Override
       public String toString () {return "[" + value + "]"; }
}
```

Scanner.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
*/

package scanner;

import scanner.Token;

/**
 * Provides a rudimentary mechanism for mapping strings found in a input stream
 * to tokens.
 * @author Larry Morell
 */
```

```
public interface  Scanner {
       /**
        *
        * @return The token most recently retrieved via get()
        */
       public Token current();
       /**
        * returns the next token in the input stream associated with the scanner
        * @return
        */
       public Token get();

       /**
        * Returns a copy of the state of the scanner that will not be modified
        * by further scanner activity.
        * @return
        */
       public Scanner copy();

       /**
        * Save the current state of the scanner for later
        */
       public void mark();

       /**
        * Restores the state to that previously marked
        */
       public void reset();
       /**
        *
        * @return Returns true iff no more tokens can be read by the scanner.
        * Note that this may advance the read point past existing whitespace
        */
       public boolean eof();
}
```

StackedBuffer.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */
```

```
package scanner;

import java.io.FileReader;
import java.util.Stack;

/**
 * A StackedBuffer implements a stack of buffers, which gives the impression
 * of a continuous stream of access through nested buffers, effectively hiding
the
 * nested buffers from a client program which only needs to request a change of
 * input streams, either by supplying a file name or a string.  When the
substituted
 * buffer is read to completion, reading continues smoothly from the previously
 * stacked buffer.  Tokens cannot be broken across buffers.
 * @author Larry Morell
 */
public class StackedBuffer extends BufferWrapper {
	// ---------------- Instance variables ----------------//
	Stack<AbstractBuffer> stack;
	// ---------------- Constructors ----------------------//
	/**
	 * Default constructor: no buffer, empty stack
	 */
	public StackedBuffer() {
		super();
		buffer = new Buffer("");
		stack = new Stack<AbstractBuffer>();
	}

	/**
	 * Supply a buffer as the starting source
	 * @param b Buffer to be wrapped
	 */
	public StackedBuffer(AbstractBuffer b) {
		super();
		stack = new Stack<AbstractBuffer>();  // create an empty stack
	buffer = b.copy();  // invoke the overriden copy in b
	}

	/**
	 * Copy constructor
	 * @param sbw
```

```
         */
        public StackedBuffer(StackedBuffer sbw) {
               StackedBuffer temp = new StackedBuffer();
               temp.stack.addAll(stack);
        }
        /**
         *
         * @return a copy of the stacked buffer
         */



    /**
         * Establish a new reading source for this buffer from the specified
string
         * @param s -- the string to read from
         */
        public void setSource(String s) {
               if (buffer != null)
               stack.push(buffer.copy());   // copy current buffer onto the stack
               buffer.setSource(s);                    // Set the source to the
string
        }

        /**
         * Establish a new reading source for this buffer from the specified file
         * @param fileReader -- reader for accessing the file
         * @param fileName -- the file name of the file
         */
        public void setSource(FileReader fileReader, String fileName) {
               AbstractBuffer ab = buffer.copy();
               stack.push(ab);
               buffer.setSource(fileReader,fileName);
        }



        public AbstractBuffer copy() {
               return new StackedBuffer(this);
        }
        @Override
        public boolean eof() {
               while (buffer.eof() && ! stack.empty()) {
                      buffer = stack.pop();
        }
        return buffer.eof();
}

@Override
public char get() {
        char ch;
        if (!eof())
                return buffer.get();
        else return '\0';
}
@Override
public void advance(){
        if ( !eof())
                buffer.advance();
}

// ---------------- Getters/Setters -------------------//

// test routine
public static void check(boolean b, String msg) {
        Buffer.check(b,msg);
}
public static void main (String[] args) {
 /* The thing to test is the ability to start with one buffer, switch to
 * another buffer, finish reading from that buffer, then switch back
 */
        Buffer b = new Buffer("This is the first buffer");
        // Read 2 words
        StackedBuffer sb = new StackedBuffer(b);
        String w = sb.getId();
        check (w.equals("This"),"getId worked");
        w = sb.getId();
        check (w.equals("is"),"getId worked");
        // Now switch to a new buffer
sb.setSource("1 2 3 4");
        w = sb.getNumber();
        check (w.equals("1"),"getId worked");
        w = sb.getNumber();
        check (w.equals("2"),"getId worked");
        w = sb.getNumber();
        check (w.equals("3"),"getId worked");
```

```
            w = sb.getNumber();
            check (w.equals("4"),"getId worked");
            w = sb.getId();
            check (w.equals("the"),"getId worked");
      }
}
```

StringToken.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package scanner;

import parsing.Symbol;
import scanner.Token;

/**
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class StringToken extends Token {
      // ---------------- Instance variables ----------------//
      String value;  // intentional shadowing; inherited value must be
"<string>"
      // ---------------- Constructors ----------------------//
      public StringToken (Symbol s, AbstractBuffer b, int lineno, int charpos)
{
            super(s,b,lineno,charpos);
            value = s.getValue();
            symbol = Symbol.createSymbol(s.getGrammar(), "<string>");  // set
the inherited value
    }

      // ---------------- Getters/Setters -------------------//
    public String getString() {return value; }
      // ---------------- Other member functions ------------//
      @Override
      public String toString () {return "[" + value + "]"; }
}
```

Token.java

```
package scanner;
import parsing.Symbol;

/**
 * The Token class associates strings with their source (filename, line, pos)
 * and symbol, and classifies them into one of the following:
 * <ul>
 * <li>&lt;id>  -- a char followed by one or more chars, digits, or
underscores</li>
 * <li>&lt;no -- a typical floating point of integer literal </li>
 * <li>&lt;string -- a quote (single or double) followed by zero or more
intervening chars
 *      followed by a matching quote (single or double; special characters must
be escaped by a backslash.
 *      These include:
 *      <ul>
 *         <li>a backslash </li>
 *         <li>a double quote (if the opening quote is a double quote)</li>
 *         <li>a single quote (if the opening quote is a single quote)</li>
 *         <li>a newline</li>
 *      </ul>
 * </li>
 * <li>&lt;char> </li>
 * <li>&lt;operator</li>
 * </ul>
 *
 * @author Larry Morell <morell@cs.atu.edu>
 */

public class Token {
	//-----------------  Member variables -----------------//
	String value ;              // The string      extracted from the buffer
	protected Symbol symbol;  // the classification of this string
	protected AbstractBuffer buffer; // the buffer from whence the token was
extracted
	protected String fileName;          // the name of the file from whence
this token was extracted
	protected int lineNo;                 // the line number in the source
file
```

```
	protected int charPos;               // the character position in the
source file

	//--------------------- Constructors -----------------//
	public Token(Symbol s, AbstractBuffer b, int lineno, int charpos) {
		value = s.getValue(); // not really needed, may save time
		symbol = s;
		buffer = b;
		symbol = s;
		charPos = charpos;
		lineNo = lineno;
		fileName = b.getFileName();
	}

	// Getters
	public AbstractBuffer getBuffer() {
		return buffer;
	}

	public int getCharPos() {
		return charPos;
	}

	public String getFileName() {
		return fileName;
	}

	public int getLineNo() {
		return lineNo;
	}

	public Symbol getSymbol() {
		return symbol;
	}

	public String getValue() {
		return value;
	}

	@Override
	public String toString() {
		return "[" + symbol + "]";
```

```
       }
}
```

Code in package "parsing"

BlockInstruction.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package parsing;



//import OriginalRuntime.Value;
import runtime.EnvironmentVal;
import runtime.Value;

/**
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class BlockInstruction extends Instruction {
   // ---------------- Instance variables ----------------//
   InstructionList instructionList;

       // ---------------- Constructors ----------------------//
       public BlockInstruction(InstructionList instructionList){
              this.instructionList = instructionList;
       }
       /**
        *
        * @return The instruction list to be executed is returned so the
        *         UI can interpret it
        */
       @Override
       public InstructionList exec() {
      return instructionList;
       }

       @Override
       public Instruction createInstruction(InstructionList il) {
         return new BlockInstruction(il);
       }
	@Override
	public Value eval() {
		for (Instruction instruction : instructionList) {
			System.out.println("Evaling instruction:"
+instruction.eval());

		}
		return NULL_NOTE;
	}

	@Override
	public Value eval(EnvironmentVal env) {
		throw new UnsupportedOperationException("Not supported yet.");
	}

	// ---------------- Getters/Setters -------------------//

	// ---------------- Other member functions ------------//
}
```

GenericParser.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */
package parsing;

import static java.lang.Character.isDigit;
import scanner.Scanner;
import scanner.Token;

/**
 * This is the central class of the entire project.  A Generic parser is
 essentially driven by a grammar stored as a data structure, similar in
 concept to a table-driven parser.  They differ in their goals and their
 underlying implementation.

 In a table-driven parser, primitive parsing actions are stored in a table.
 The rows of the table are states and the columns of the table are input
 tokens.  Entries in the table are parse actions.  Therefore a given state and
 input leads to a particular parse action.  The parser is fast; the table can
 be of substantial size.

 In the generic parser implemented here, the table is replaced with a
 collection of tries, one trie for each non-terminal in the grammar.  A trie
 for nonterminal N represents all the rules for N.  Every nonterminal has a
 parser associated with it.  The vast majority of these are instances of this
 generic parser.  The specialized parsers are for nonterminals that
 traditionally are used to represent the tokens for id, number, and string.

 The generic parser is implemented in the style of recursive-descent.  Given a
 nonterminal N to parse, it looks up the trie for the nonterminal and then
 proceeds to parse as directed by the trie.  Each element of the trie is a
 symbol from a grammar rule.  A path from the root of the trie is a prefix of a
 grammar rule associated with N.  The generic algorithm involves using the
 current input (supplied by a scanner) to select the correct path to follow.
 Once the next node is identified, the parser associated with the symbol in
 that node is retrieved and executed.

 Each parser returns a list of instructions, one instruction for each symbol
 in the rule that is parsed.  The list of instructions returned by parsing a
 symbol is passed to a routine for creating a single instruction from the
 list; this single instruction is then appended to the instruction list being
 constructed by the parser.

 Parsing using a trie is greedy in the sense that it returns the longest
 possible parse, which corresponds to the longest possible rule of N that can
 be matched.  Thus if one rule is a prefix of a second rule, and the input
 satisfies the second rule, then the then parsing will match the second rule;
 if parsing cannot extend beyond the localFirst rule, then the localFirst rule
is what
 is matched.  Parsing fails if the localFirst rule completes and the next input
 indicates the second rule should match.  No backtracking occurs!
 *
 * @author Larry Morell
 */
public class GenericParser extends Parser {
      public GenericParser(Symbol s) {
      super(s);
      }

      public GenericParser() {
      throw new UnsupportedOperationException("Not yet implemented");
```

```
}
// parse -- assumes grammar has been set

public Instruction parse(Scanner scanner) {
InstructionList result = new InstructionList();

Symbol s;
Symbol st;
Token t = scanner.current();  // think ...  have we already read this???
Instruction instr;
s = trie.getSymbol();
st = t.getSymbol();
if (trie.getSymbol().isTerminal()) {
             // simple case: invoke the parser associated with the
terminal
             // if the symbol matches the  terminal
             if (t.getSymbol() == trie.getRoot().getSymbol()) {
             instr = trie.getInstruction();
             return instr;
             } else {
             System.err.println("Expected " + trie.getRoot().getSymbol()
             + "; found " + t);
             return null;
             }
} else {
  // the root symbol in the trie is a nonterminal
  // complex case: descend the trie associated with the nonterminal
  // invoking the appropriate parsers as we go
  // Start by getting the current input symbol
             TrieNode tn;
             s = t.getSymbol();
             if (s.value.equals("<PlaceHolder>"))
             {
             String t1 = t.getValue();
             int pos = t1.length()-1;
             while (pos >=0)
             {
                    if (!isDigit(t1.charAt(pos)))
                    break;
                    pos--;
             }
             String v = "<"+t1.substring(0, pos+1)+">";
```

```
                    st = s.getGrammar().symbols.get(v);
                    if (st.isNonTerminal())
                    {
                          Symbol ter = trie.getRoot().getTermial(st);
                          tn = trie.firstNode(ter);
                    }
                    else
                          tn = trie.firstNode(st);
                          }
                          else
                          {
                                // Calling firstNode() returns the TN where
the terminal is found
                                // This determines if the current input
symbol, s,
                          // is the prefix of the nonterminal
                          tn = trie.firstNode(s);
                    }
                    TrieNode parent = trie.getRoot();
                    if (tn == null) { // no match
                    if (trie.isNullProduction()) {
                          return parent.instruction.createInstruction(result);
                    }
                    // Find the nullable production and follow it
                    if (trie.getSymbol().isNullable()) {// if the root is
nullable, then ok
                    //return new NopInstruction(); // yep, it's empty!
                    return parent.instruction.createInstruction(result);
             } else {  // root not nullable and the next input does not match -
- error
                    // System.err.println(t + " cannot be found here");
                    return null;
             }
        }
        // At this point a legitmate starting point for the rule was found
        // Now proceed downward

       //tn = parent.getChild(s);  // the child which contains the current
symbol
             while (tn != null) {

             s = tn.getSymbol();
```

```
		if (scanner.current().getSymbol().value.equals("<PlaceHolder>") &&
st.equals(s))
		{
			PlaceHolder ph = new
PlaceHolder(scanner.current().getValue());
			instr = ph.parse(scanner);
		}
		else
		{
			instr = s.parse(scanner);
		}
		if (instr != null) {
			result.add(instr);
			parent = tn;
			s = scanner.current().getSymbol();
			if (s.value.equals("<PlaceHolder>"))
			{
				String t1 = scanner.current().getValue();
				int pos = t1.length()-1;
				while (pos >=0)
				{
				if (!isDigit(t1.charAt(pos)))
					break;
					pos--;
				}
				String v = "<"+t1.substring(0, pos+1)+">";
				st = s.getGrammar().symbols.get(v);

				if (!(st.isNonTerminal()))
					tn = tn.getChild(st);
				else
				{
				Symbol ter = tn.getTermial(st);
				if (ter!=null)
					tn = tn.getChild(ter);
				else
					tn = null;
				}
			}
			//if (!(s.value.equals("<PlaceHolder>")))
			else
```

```
                           tn = tn.getChild(s);
                        // If tn is null then there was no child that
                    }
            else { // parsing failed
                    tn = null;
            }
         }
         if (parent.isEndOfRule()) {
                // Here is the main effect.  We have parsed successfully up to
and
            // including the parent.  What we do now is to convert the result
            // to a single instruction by invoking the instruction builder
            // associated with the parent.  This ensures uniformity of every
            // parser: all parsing results in a single instruction, built,
perhaps,
            // from a list of constituent instructions
            return parent.instruction.createInstruction(result);
         }
         else {
              System.out.println("Failure to recognize a complete rule while
parsing");
              return null;
         }
      }
   }

   // test it
   public static void main(String[] args) {
      /* To test the generic parser we will need to have:
       * A grammar with rules that use the generic parser along with the other
parsers.
       * A scanner initialized with a program to be parsed
       */
   }
}
```

GenesisString.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
package parsing;


import parsing.GenericParser;
import parsing.Instruction;
import parsing.InstructionList;
import parsing.SyntaxRule;
import runtime.DoubleVal;
import runtime.EnvironmentVal;
import runtime.StringVal;
import runtime.Value;

/**
 *
 * @author Larry Morell
 */
public class GenesisString extends Instruction {
    private static String rule = "<GenesisString> ::= <id>";
    private static SyntaxRule syntaxRule ;
       public static void initialize(Grammar GRAMMAR) {  // static
initialization
             syntaxRule  = new SyntaxRule(GRAMMAR, rule);

             GRAMMAR.addRule(syntaxRule,
                new UnsignedNumber(),
                new GenericParser(syntaxRule.lhs()));
    }
    private String string;
    Value stringNote;

    public GenesisString() {
             string = "";
    }

    public GenesisString(String s) {
             string = s;
             stringNote = new StringVal(s);
    }

    public String value() {
             return string;
    }
    @Override
    public Instruction createInstruction(InstructionList il) {
            return il.getFirst();
    }

    @Override
    public Value eval() {
            return stringNote;
    }

    @Override
    public Value eval(EnvironmentVal env) {
            return stringNote;
    }

    @Override
    public InstructionList exec() {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }
}
```

Grammar.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package parsing;

import scanner.Buffer;
import scanner.Scanner;
import scanner.Token;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Stack;
import scanner.GenericScanner;

/**
 * A Grammar is an administrative unit which associates (N,T,P,S) where
 * <ul>
 * <li> N is a set of non-terminals, </li>
 * <li> T is a set of terminals, </li>
 * <li> P is a set of production rules of the form N := (N U T)*, </li>
 * <li> S is the default symbol for parsing </li>
 * </ul>
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class Grammar {

    LinkedHashMap<Symbol, Trie> terminals; // quick access to Terminal's
    LinkedHashMap<Symbol, Trie> nonterminals;   // stores all rules
    // Set it up so that there will be only one instance of any given symbol
    // Do so by creating a hash table of symbols already seen
    private Symbol start;
    HashMap<String, Symbol> symbols = new HashMap<String, Symbol>();
    Token current;
    Buffer buffer;
    Stack<Buffer> bufferStack;

    /**
     * Builds a grammar with no terminals, nonterminals or rules
     */
    public Grammar() {
        terminals = new LinkedHashMap<Symbol, Trie>(100, 0.75F);
        nonterminals = new LinkedHashMap<Symbol, Trie>(200, 0.75F);
        start = null;
        current = null;
    }

    public boolean isSymbol(String s) {
        return symbols.get(s) != null;
    }

    public boolean isNonterminal(String s) {
        Symbol sym = symbols.get(s);
        if (sym == null) {
            return false;
        }
        return (nonterminals.get(sym) != null);
    }
    public boolean isTerminal(String s) {
        Symbol sym = symbols.get(s);
        if (sym == null) {
            return false;
        }
        return (terminals.get(sym) != null);
    }

    public Symbol createSymbol(String s, char rep, String sep) {
        Symbol symbol = symbols.get(s);
        if (symbol == null) {  // not there
            symbol = Symbol.createSymbol(this, s, rep, sep);
            Symbol sym = symbols.get(symbol.getValue()); // see if it is
defined
            if (sym == null) {
                symbols.put(symbol.getValue(), symbol);
            }
            // Note this adds (s,symbol) to
            // symbols
            // Not sure we need to distiguish between terms and nonterms
            Trie trie = new Trie(symbol);  // new one since the symbol did not
exist
            if (symbol.isNonTerminal()) {  // addRule it to the appropriate
list
                nonterminals.put(symbol, trie);  // nonterminals here
            } else {
                // here is where we handle the pseudo-nonterminals
                terminals.put(symbol, trie);   // terminals here
// I think here is where we need to consider using my new stategy-driven
Scanner
                if (symbol.getType() == SymbolType.NUMBER)
                    trie.setParser(new NumberParser(symbol));
                else if(symbol.getType() == SymbolType.STRING)
                    trie.setParser(new StringParser(symbol));
                else if(symbol.getType() == SymbolType.ID)
                    trie.setParser(new IdParser(symbol));
//                                     trie.setParser(new
UnsignedNumberParser(symbol));
            }
        }
        return symbol;
    }
```

```
public Symbol createSymbol(StringBuilder b, char rep, String sep) {
    return createSymbol(new String(b), rep, sep);
}

public Symbol createSymbol(String s) {
    return createSymbol(s, '1', "");
}

public Symbol createSymbol(StringBuilder s) {
    return createSymbol(s, '1', "");
}

public void deleteSymbol(Symbol s) {
    if (s.isTerminal()) {
        terminals.remove(s);
    } else {
        nonterminals.remove(s);
    }
    symbols.remove(s.value);
}

public void addRule(SyntaxRule r, Instruction instr, Parser p) {
    // This builds/extends the trie for the lhs
    // For each symbol in the rhs, addRule it to the trie
    Trie t = nonterminals.get(r.lhs());

    t.addRule(r, instr, p);
}

public void setBuffer(Buffer b) {
    buffer = b;
}

public Token getToken() {
    Buffer b = this.buffer;
    String sb;
    String result;
    Token t;
    int endPos;
    b.skipBlanks();
    int ln = b.lineNumber();
    int cn = b.columnNumber();
    if (b.eof()) { // must be eof
        t = new Token(createSymbol(""), b, ln, cn);
    } else if (Character.isLetter(b.peek())) {
        t = new Token(createSymbol(b.getId()), b, ln, cn);
    } else if (Character.isDigit(b.peek())) {
        t = new Token(createSymbol(b.getNumber()), b, ln, cn);
    } else if (b.peek() == '/') {
        int pos = b.currentPos();
        if (b.charAt(pos + 1) == '/') {
            sb = b.getToEoln();
            t = new Token(createSymbol(sb), b, ln, cn);
        } else if (b.nextChar() == '*') {
            b.get();
            b.get();  // advance past the *
            sb = "/*" + b.getToDelimiter("*/");
            t = new Token(createSymbol(b.getToDelimiter("*/")), b, ln, cn);

        } else { // must be an operator starting with /
            sb = b.getOperator();
            t = new Token(createSymbol(b.getOperator()), b, ln, cn);
        }
    } else { // must be an operator of some sort
        sb = b.getOperator();
        t = new Token(createSymbol(sb), b, ln, cn);

    }
    current = t;
    return t;
}

public Instruction parse(Symbol s, Scanner scanner) {
    Instruction result = s.parse(scanner);

    return result;
}

public Instruction parse(Scanner scanner) {
    return parse(start, scanner);
}

public Token currentToken() {
        return current;
    }

    public void setStart(String s) {
        if (symbols.get(s) == null) {
            System.out.println("Cannot set start symbol to '" +s + ": no such
non-terminal");
            System.exit(1);
        }
        start = createSymbol(s);
    }

    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        result.append("<---------------------------------");
        result.append("\nNonTerminals: ");
        for (Trie t : nonterminals.values()) {
            result.append(t.getSymbol());
            result.append(",");
        }
        result.setCharAt(result.length() - 1, ' ');
        result.append("\nTerminals: ");
        for (Symbol t : terminals.keySet()) {
            result.append(t);
            result.append(",");
        }



        result.setCharAt(result.length() - 1, ' ');
        // Now print the First sets of each symbol
        // Now print the First sets of each symbol

        result.append("\nSymbol    FirstSet\n");
        for (Trie t : nonterminals.values()) {
            Symbol s = t.getSymbol();
            result.append(s.getValue());
            result.append("   ");
            result.append(s.getTrie().getFirstSet());
            result.append("\n");
        }
        result.append("--------------------------------->\n");
        return new String(result);
    }

    public Symbol getStart() {
        return start;
    }

    public static void readAllTokens(Buffer buffer, Grammar g) {
        buffer.skipBlanks();
        Token t;
        t = g.getToken();
        while (t.getSymbol().getType() != SymbolType.EOF) {
            System.out.println("Token read: " + t);
            t = g.getToken();
        }
    }

    public static void testToken(Grammar g) {
        System.out.println("Buffer test:");
        Buffer in = new Buffer("a");
        readAllTokens(in, g);
        in = new Buffer("    17.3");
        readAllTokens(in, g);
        in = new Buffer("/* this is the time for a \n comment on \n several
lines */ \n");
        readAllTokens(in, g);
        in = new Buffer("// This is a comment \n wowie");
        readAllTokens(in, g);
        in = new Buffer("\n \t \r:= ");
        readAllTokens(in, g);
        in = new Buffer("/;");
        readAllTokens(in, g);
        in = new Buffer("This is a line of ids\n"
                + "with some numbers:: 3 43 14.7 -55.883\n"
                + "and some operators: + - * / // a comment\n"
                + "/* a more \n    complicated \n comment */\n"
                + "and some more weird operators: *^**%%%* ~#!#$$$^$@@`");
        readAllTokens(in, g);
        System.out.println("--------------------");
    }

    /**
```

```
     * Builds a grammar from the following rules:
     * <pgm> ::= <stmt>
     * <stmt> ::= <move>
     * <stmt> ::= <print>
     * <move> ::= move <id>
     * <print> ::= print <id>
     *
     * @return
     */
    public static Grammar buildGrammar() {
        Grammar g = new Grammar();
        String[] rules = new String[]{
            " <pgm> ::= <stmt>",
            " <stmt> ::= <move>",
            " <stmt> ::= <print>",
            "<move> ::= move <id>",
            "<print> ::=  <id> print",};
        // Add the rules to the grammar
        SyntaxRule rule;
        for (int i = 0; i < rules.length; i++) {
            rule = new SyntaxRule(g, rules[i]);
            g.addRule(rule, null, null);  // no instruction, no parser
        }
        return g;
    }

    public static void test() {
        System.out.println("Grammar test");
        // Set up the grammar
        Grammar g = new Grammar();
        // testToken(g);
        // Put in a single rule
        Symbol s = g.createSymbol("<pr>");
        //g.addRule(new SyntaxRule(g, "<pr> ::= print <no>\n"), new
GenericParser(s));  // fix me
        // Add in an artificial rule for <no>
        s = g.createSymbol("<no>");
        //g.addRule(new SyntaxRule(g, "<no> 123"), new NoParser(s));  // fix me
        // Set up the input source and get the first token
        g.setBuffer(new Buffer("5"));
        Token t = g.getToken();
        // Ensure the first token is "print"
        System.out.println("First Token is " + t);
        // Set up the start symbol
        g.setStart("<no>");

        // Now let's start the parsing parade!
        Scanner scanner = new GenericScanner(g, "parse this");
        g.parse(g.getStart(), scanner);
        /*
         GenesisIO infile = new GenesisIO("grammar.dat", "");
         if (infile == null) {
         System.err.println("File not found: ");
         System.exit(1);
         }
         String line = infile.readLine();
         while (line != null) {
         if (!line.matches("^ *$") && !line.regionMatches(0, "//", 0, 2)) {
         g.addRule(new SyntaxRule(g, line));
         System.out.println("Grammar:\n" + g.toString());
         } else {
         System.out.println(line);
         }
         line = infile.readLine();
         }
         */
        //
        // Now add the predefined

        System.out.println("Grammar:\n" + g.toString());
    }
}
/*
 Procedure ReadGrammar(Var  fn: string);
 Procedure PrintGrammar(Var F: Text; Var G: Grammar);
 Procedure PrintAllNonTerms (Var F: Text; G: Grammar);
 Procedure PrintAllTerms (Var F: Text; G: Grammar);
 Procedure PrintFirstSet (Var F: text; G: Grammar);
 Procedure PrintFollowSet (Var F: text; G: Grammar);
 Function NontermCnt (G:Grammar):integer;
 Function TermCnt (G: Grammar):integer;
 Function RuleCnt (G: Grammar ):integer;
 Function StartSymbol (G: Grammar): Nonterm;
 Function NontermStr  ( n : Nonterm ): SymbolStr;
 Function TermStr ( t: Term ): SymbolStr;
 Function Nthterm ( n: integer): term;
 Function NthNonterm ( n : integer ): Nonterm;
 Function NontermNum ( N: Nonterm): integer;
 Function TermNum ( T: term): integer;
 Function SyntaxRule (  i: integer; j: integer ): Symbol;
 Function FormatRule ( i: integer; j: integer ): Symbol;
 Function NumberOfRules ( n: Nonterm ):integer;
 Function LengthOfSRule ( i:integer ): integer;
 Function LengthOfFrule ( i: integer ): integer;
 Function RuleNo (n: Nonterm; i: integer ): integer;
 Procedure SymbolIndex ( i : integer; var n : Nonterm; var j : integer  );
 Function KeyStroke ( n: Nonterm; i: integer ):char;
 Function Isterm ( S: Symbol): Boolean;
 Function IsNonterm ( S: Symbol): Boolean;
 Function Termpos (S: Symbol): integer;
 Function NontermPos (S: Symbol): integer;
 Procedure First (G:  Grammar; S: Symbol; Var TS: Termset);
 Procedure CopyTail ( Var T: Srule; P: Integer; Pos: integer);
 Procedure ComputeFirst ( x: SRule;  (* working string *)
 k: integer; (* length of working string *)
 var Result: Termset);
 */
```

IdParser.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package parsing;

import scanner.Scanner;


/**
 *
 * @author Larry Morell
 */
public class IdParser  extends TerminalParser {
      public IdParser(Symbol s) { super(s);   }
```

```
    @Override
    public Instruction parse(Scanner scanner) {
             String id =(scanner.current().getValue());

             Instruction result;
       result = new UnsignedId(id);

         scanner.get();
             return result;
       }
}
```

Instruction.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package parsing;

//import OriginalRuntime.EnvironmentVal;
//import OriginalRuntime.Value;
import runtime.Value;
import runtime.EnvironmentVal;
import runtime.StringVal;

/**
 * An instruction in a virtual machine
 * @author Larry Morell
 */
public abstract class Instruction {
    final static protected Value NULL_NOTE = new StringVal("No value");
     String operator;
     public String getOperator() { return operator;}
     public void setOperator(String s) { operator = s;}
       public static String indent(int n) {
             return "
".substring(0, n);
       }
       public InstructionList il;
       protected Source source;
       protected Instruction nextInstruction = null;
```

```
        //Constructors
        public Instruction() {
        }
     public Instruction(Instruction a){
          }
     public Instruction(Source s, InstructionList il) {
               this();
               this.il = il;
               source = s;
        }
        public Instruction copy(Instruction a)
     {
        return a;
     }
     public InstructionList getInstructionList() {return this.il;//new
InstructionList(this);
   }
        /**
         * Override this instruction to execute the contained instruction
         *
         * @return A list of instructions that need to be executed for
accomplishing
         * this instruction.  null is returned if the instruction is directly
interpreted.
         * This is similar in effect to a macro expansion in which one
instruction is
         * replaced with a 0 or more instructions.  A list of instructions will
be
         * prepended to the the instruction list from which it was executed.
         */
        public abstract InstructionList exec();

        abstract public Instruction createInstruction(InstructionList il);

        public void printError() {
               System.err.println("Could not execute" + this);
               System.exit(1);
        }

        public void printError(String msg) {
               System.out.println(msg);
        }

        /**
         * Override this function to pretty print the instruction
         * @param n  The number of spaces of indentation
         * @return  A string of a pretty-printed form of the parsed input
         *
         */
        public StringBuffer prettyPrint(int n) {
                StringBuffer result = new StringBuffer(1000);
                for (Instruction instr : il) {

                        result.append(indent(n));
                        if (instr.isLiteral()) {
                                result.append(instr);
                        }
                        else {  // indent "instructions" which are at the same
level
                                result.append(instr.prettyPrint(+2));
                        }
                }
                return result;
        }

        /**
         *
         * @return Returns whether or not this instruction is a literal; a
literal instruction
         * is one which ...??? fill this in!
         */
        public boolean isLiteral() {
                return false;
        }

        public String getGrammarRule() {
                return "";
        } // default

        public void setGrammarRule() {
        }

        /**
```

```
         * Establish the next instruction to be executed after this one
         * @param next  The successor to this instruction
         */
        public void setNextInstruction(Instruction next) {
                nextInstruction = next;
        }

        abstract public Value eval();
        abstract public Value eval(EnvironmentVal env);
}
```

InstructionList.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package parsing;

import java.util.LinkedList;

/**
 *
 * @author Larry Morell
 */

public class InstructionList  extends LinkedList<Instruction> {

        public InstructionList() {
                super();
        }
        public InstructionList (InstructionList il) {
                super();
                for (Instruction it : il){
              // probably not good enough.  Probably need to create a new
                        // instruction for each it.  Hopefully this will be good
enough
                        // for now
                        add(it);
                }
        }
	public InstructionList (Instruction i) {
		this.add(i);
	}
}
```

NopInstruction.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package parsing;

//import OriginalRuntime.EnvironmentVal;
//import OriginalRuntime.Value;
import runtime.EnvironmentVal;
import runtime.Value;

/**
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class NopInstruction extends Instruction {
	final static NopInstruction NOP_INSTRUCTION = new NopInstruction("");

	NopInstruction () {operator = "";}
	NopInstruction (String s) { operator = s;}
	// ---------------- Other member functions ------------//
	@Override
	/**
	 * Do nothing
	 */

	public InstructionList exec() {
		return null;
	}

	@Override
	public Instruction createInstruction(InstructionList il) {
	if (il.size() == 0)
		return new NopInstruction();
		else
```

```
                    return new NopInstruction(il.getFirst().getOperator());
        }
    public Instruction copy (NopInstruction a)
    {
        Instruction result = new NopInstruction(a.getOperator());
        return result;
    }
       @Override
       public Value eval() {
              return NULL_NOTE;
       }

       @Override
       public Value eval(EnvironmentVal env) {
              return NULL_NOTE;
       }

       public String toString() {
              return operator;
    }
}
```

NumberParser.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */

package parsing;

import scanner.NumberToken;
import scanner.Scanner;
import scanner.Token;

/**
 *
 * @author Larry Morell
 */
public class NumberParser  extends TerminalParser {
       public NumberParser(Symbol s) { super(s);      }

       @Override
```

```
    public Instruction parse(Scanner scanner) {
       Token t = scanner.current();
       if (! (t instanceof NumberToken)) {
                    System.out.println("Expected a number that was not found");
       }
       double number = ((NumberToken) t).getNumber();
              Instruction result = new UnsignedNumber(number);
       scanner.get();
              return result;
       }
}
```

Parser.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package parsing;

import scanner.Scanner;

/**
 *
 * @author Larry Morell
 */
public abstract class Parser {
       Symbol symbol;
       Trie trie;
       public Parser () {
              symbol = null; trie = null;
       }

       Grammar getGrammar () {
              return symbol.getGrammar();
       }

       public Parser(Symbol s) {
              symbol = s;
              trie = s.getTrie();
       }
	abstract public Instruction parse (Scanner scanner);
} // End of abstract class Parser
```

ParserInstruction.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package parsing;

/**
 *
 * @author Larry Morell
 */
abstract public class ParserInstruction extends Instruction {


	@Override
	public InstructionList exec() {
		System.out.print(source);
		return null;   // It's done it's work, nothing more
	}

	@Override
	abstract public Instruction createInstruction(InstructionList il);

}
```

PlaceHolder.java

```
/*
*  Copyright (C) 2016 Xin Wan <xwan@cs.atu.edu>
*/
package parsing;

import runtime.EnvironmentVal;
import runtime.StringVal;
import runtime.Value;
import scanner.Scanner;
```

```
/**
 *
 * @author xin
 */
public class PlaceHolder extends Instruction{
    public String value;
    PlaceHolder (String v) { value = v; }
    PlaceHolder (PlaceHolder ph) {
        value = ph.value;
    }
    public Instruction parse(Scanner scanner) {
             String id =(scanner.current().getValue());

             Instruction result;
         result = new PlaceHolder(id);

             scanner.get();
             return result;
       }
    @Override
    public Instruction createInstruction(InstructionList il) {
             return il.getFirst();
    }
    public Instruction copy (PlaceHolder a)
    {
             Instruction result = new PlaceHolder(a.value);
             return result;
    }
    @Override
    public Value eval() {
             return new StringVal(value);
    }

    @Override
    public Value eval(EnvironmentVal env) {
             return new StringVal(value);
    }

    @Override
    public InstructionList exec() {
             throw new UnsupportedOperationException("Not supported yet.");
//To change body of generated methods, choose Tools | Templates.
```

```
    }
}
```

Source.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package parsing;

/**
 *
 * @author Larry Morell
 */
public class Source {
   String source;
       int lineNumber;
       int charPos;
       String filename;

       public Source() {
              source = "";
              lineNumber = 0;
              charPos = 0;
              filename = "Unknown";
       }

       public Source(String source, int lineNumber, int charPos, String
fileName) {
              this.source = source;
              this.lineNumber = lineNumber;
              this.charPos = charPos;
              this.filename = fileName;
       }
}
```

StringParser.java

```
/*
 *  Copyright (C) 2014 Joshua Townsend <jtownsend2@atu.edu>
 */
```

```
package parsing;

import scanner.NumberToken;
import scanner.Scanner;
import scanner.StringToken;
import scanner.Token;


/**
 *
 * @author Larry Morell
 */
public class StringParser  extends TerminalParser {
       public StringParser(Symbol s) { super(s);      }

       @Override

public Instruction parse(Scanner scanner) {
      Token t = scanner.current();
      if (! (t instanceof StringToken)) {
         System.out.println("Expected a string that was not found");
      }
      String string = ((StringToken) t).getString();
              Instruction result = new GenesisString(string);
      scanner.get();
              return result;
       }

}
```

Symbol.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package parsing;
import static java.lang.Character.isDigit;
import scanner.Scanner;
import java.util.Collection;
import java.util.LinkedList;
import java.util.TreeSet;
import scanner.Token;

/**
 *
 * @author lmorell
 */
public class Symbol implements Comparable<Symbol> {

    /**
     * ***************** Member variables******************
     */
    SymbolType type;     // indicates whether it is a terminal or non-terminal
    String value;         // indicates the particular  uniq id for the term or
non-term
    // like non-terminal N, number is 3.945, id is transformer,
    // operator is like :=, <, <>!, !, etc
    final static boolean debug = false;
    boolean inGrammar = false;
    boolean dl = false;             // true iff this symble derives lambda
    boolean repeating = false;
    char repetitionChar = '1';
    String separatorString = "";
    Grammar grammar;                // containing grammar
    private String description;     // reserved  for better messages ...
future
    LinkedList<TrieNode> locations; // list of locations where this symbol
appears

    // Interface implementation
    /**
     *
     * @param s
     * @return negative, 0 or positive from comparing this.value with s.value
     *
     */
    public int compareTo(Symbol s) {
        return value.compareTo(s.value);
    }
    /**
     * *************** Constructors *****************
     */
    private Symbol(Grammar g, String v) {
        type = SymbolType.NONTERMINAL;
        value = v;
        dl = false;
        description = "No description";
        locations = new LinkedList<TrieNode>();   // not in a trie yet !
        grammar = g;
    }

    // To facilitate symbols being unique to a grammar, we store the list of
    // symbols in the grammar, so that the symbol lists from different grammars
    // will not overlap.
    public static Symbol createSymbol(Grammar g, String v, char rep, String
sep) {
        if (g == null) {
            System.err.println("createSymbol: attempting to create a symbol "
                + v + " for a non-existent grammar");
            System.exit(1);

        }
        Symbol symbol = g.symbols.get(v);   // first see if it is there
        if (symbol == null) {     // it is not
            symbol = new Symbol(g, v);    // create it and init it with string
v
            if (v.equals("")) {
                symbol.setType(SymbolType.EOF);
            } else {
                char ch = v.charAt(0);
                // A nonterminal matches "<[a-zA-Z]"
                if (ch == '<' && v.length() > 1 &&
Character.isLetter(v.charAt(1))) {
                    // probably need to generalize this for an arbitrary set of
                    // pseudo-nonterminals
                    if (v.equals("<id>")) {
                        symbol.setType(SymbolType.ID);
                    } else if (v.equals("<no>")) {
                        symbol.setType(SymbolType.NUMBER);
                    } else if (v.equals("<string>")) {
                        symbol.setType(SymbolType.STRING);
                    } else if (v.equals("<PlaceHolder>")) {
                        symbol.setType(SymbolType.PLACEHOLDER);
                    }
                    else {
                        symbol.setType(SymbolType.NONTERMINAL);
                    }
                } else { // must be a terminal that does not exist
                    if (Character.isLetter(ch)) {
                        symbol.setType(SymbolType.ID);
                    } else if (Character.isDigit(ch)) {
                        symbol.setType(SymbolType.NUMBER);
                    } else if (ch == '"' || ch == '\'') {
                        symbol.setType(SymbolType.STRING);
                    } else if (ch == '\'') {
                        symbol.setType(SymbolType.CHAR);
                    } else {
                        symbol.setType(SymbolType.OPERATOR);
                    }
                    //   symbol.firstSet.put(symbol,null);  // it is its own
symbol; there
                    // is no TrieNode accessible from
                    // a terminal, so the corresponding
                    // TrieNode is empty
                }
            }

            symbol.setDerivesLambda(false);
            //g.symbols.put(v, symbol);   // add it to the list of symbols
        } else { // the base symbol was there, see if it has the same
repetition
            // factor and separator
            if (symbol.repetitionChar != rep ||
!symbol.separatorString.equals(sep)) {
                // There is a difference; create a new symbol
                symbol = new Symbol(g, v);    // create it and init it with
string v
                if (v.equals("")) {
                    symbol.setType(SymbolType.EOF);
                } else {
                    char ch = v.charAt(0);
                    // A nonterminal matches "<[a-zA-Z]"
                    if (ch == '<' && v.length() > 1 &&
Character.isLetter(v.charAt(1))) {
                        symbol.setType(SymbolType.NONTERMINAL);
                    } else { // must be a terminal
                        if (Character.isLetter(ch)) {
                            symbol.setType(SymbolType.ID);
                        } else if (Character.isDigit(ch)) {
                            symbol.setType(SymbolType.NUMBER);
                        } else {
                            symbol.setType(SymbolType.OPERATOR);
                        }
                        //   symbol.firstSet.put(symbol,null);  // it is its
own symbol; there
                        // is no TrieNode accessible from
                        // a terminal, so the corresponding
                        // TrieNode is empty
                    }
                }
            } // the symbol was already there
            else {

                if (Character.isLetter(v.charAt(0))) { // if a char, then it
must be a key
                    symbol.inGrammar = true;
                    symbol.setType(SymbolType.KEYWORD);

                }
            }
        }
        return symbol;
    }

    public static Symbol createSymbol(Grammar g, String v) {
        Symbol s = createSymbol(g, v, '1', "");
        return s;
    }

    public static Symbol createSymbol(Grammar g, String v, String d) {
        Symbol s = createSymbol(g, v, '1', "");
        s.description = d;
        return s;
    }
```

```
/**
 * ************ Getters and setters
 *
 *****************
 * @return the associated set of first symbols
 */
public Trie getTrie() {
    if (isNonTerminal()) {
        return grammar.nonterminals.get(this);
    } else {
        Trie trie = grammar.terminals.get(this);

        return trie;
    }
}

public Trie getTrie(Symbol s) {
    if (s.type == SymbolType.NONTERMINAL) {
        return grammar.nonterminals.get(s);
    } else {
        Trie trie = grammar.terminals.get(s);

        return trie;
    }
}

/**
 *
 * @return The first set of the symbol as a Collection
 */
public Collection<Symbol> getFirstSet() {
    if (isTerminal()) {
        Collection<Symbol> first = new TreeSet<>();
        first.add(this);
        return first;
    }
    Trie t = getTrie();
    return getTrie().getFirstSet();
}
```

```
public SymbolType getType() {
    return type;
}

public void setType(SymbolType st) {
    type = st;
}

/**
 * Set the string value of the symbol
 *
 * @param v
 */
public void setValue(String v) {
    value = v;
}

public String getValue() {
    return value;
}

public String getDescription() {
    return description;
}

/**
 * @param description the description to set
 */
public void setDescription(String description) {
    this.description = description;
}

public TreeSet<TrieNode> getFirstOf() {
    return getTrie().getFirstOf();
}

public LinkedList<TrieNode> getLocations() {
    return locations;
}

// Mutator operators
```

```
    // Accessor operators
    public boolean isNonTerminal() {
        return type == SymbolType.NONTERMINAL;
    }

    public boolean isTerminal() {
        return type != SymbolType.NONTERMINAL;  // note this includes
PSEUDONONTERMINAL
    }

    public boolean isId() {
        return type == SymbolType.ID;
    }

    public boolean isKeyword() {
        return type == SymbolType.KEYWORD;
    }

    public boolean isNumber() {
        return type == SymbolType.NUMBER;
    }

    public boolean isString() {
        return type == SymbolType.STRING;
    }

    public boolean isEof() {
        return type == SymbolType.EOF;
    }

    public boolean isOperator() {
        return type == SymbolType.OPERATOR;
    }

    public boolean derivesLambda() {
        return dl;
    }

    public boolean isNullable() {
        return dl;
    }
    public void setDerivesLambda(boolean derivesLambda) {
        this.dl = derivesLambda;
    }

    // Abstract operations
    /**
     *
     * @param in the input stream to parse from
     * @return true iff the parsing was successful
     */
    public Instruction parse(Scanner scanner) {
        Trie trie = getTrie();
        Parser p = trie.getParser();  // get the parser associated with this
symbol
        return p.parse(scanner);

    }

    public Grammar getGrammar() {
        return grammar;
    }

    // Overriden operations
    @Override
    public String toString() {

        if (debug) {
            return value + "[" + type + "," + dl + "]";
        } else {
            return value;
        }
    } //override this

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Symbol other = (Symbol) obj;
        if ((this.value == null) ? (other.value != null) :
!this.value.equals(other.value)) {
            return false;
        }
        return true;
    }

    @Override
    public int hashCode() {
        int hash = 3;
        hash = 41 * hash + (this.value != null ? this.value.hashCode() : 0);
        return hash;
    }
    /**
     * @return a description suitable for use in error messages related to the
     * symbol
     *
     */
}
```

SymbolType.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package parsing;

/**
 *
 * @author lmorell
 */
public enum SymbolType {

    ID, // terminal that begins with a char or underscore,     // not
previously defined
    KEYWORD, // terminal that as previously defined
    NUMBER, // terminal that begins with a digit
    OPERATOR, //terminal that begins witha any other char
    NONTERMINAL, // only designation for non-terminal
    PSEUDONONTERMINAL, // except for this one!
    STRING, // String literal  " ...  "
    CHAR, // Character literal 'X'
    COMMENT, // Any form of comment
    PLACEHOLDER, //placeholder for macro tree
    EOF, // end of file
    UNKOWN,   // placeholder for a generic Symbol
};
```

SyntaxRule.java

```
package parsing;

/**
 * Defines the structure of a grammar rule as a sequence of Symbol's.
 *
 * @author Larry Morell
 */
import java.util.LinkedList;

public class SyntaxRule {

    private LinkedList<Symbol> rule;
    private int pos;
    private StringBuffer s;
    Grammar grammar;

    // ------------- Constructors -------------------
    /**
     * Constructor that parses a rule and store as a sequence of Symbol'sb.
     * Non-termainals are surrounded by <[letter]...>; blanks are separators
     *
     * @param g grammar associated with rule r
     * @param r grammar rule as string of symbols; angle brackets around
     * nonterms
     */
    public SyntaxRule(Grammar g, String r) {
        grammar = g;
        pos = 0;
        rule = new LinkedList<Symbol>();
        s = new StringBuffer(r);
        Symbol symbol = getSymbol();
        rule.add(symbol);
        symbol = getSymbol();
        if (symbol != null
            && (symbol.getValue().equals("::=") || symbol.getValue().equals("-
>"))) {  // skip it
            g.deleteSymbol(symbol);
            symbol = getSymbol();
        }
        while (symbol != null) {
            rule.add(symbol);
//         System.out.println("symbol = '" + symbol + "'");
            symbol = getSymbol();
        }
        // if empty production, set derives lambda in the lhs
        if (rule.size() == 1) {  // no rhs
            rule.get(0).setDerivesLambda(true);
        }
    }

    private void skipBlanks() {
        // System.out.println("got here with pos=" + pos);
//      System.out.println("sb = " + sb);
        while (pos < s.length() && s.charAt(pos) == ' ') {
            pos++;
        }
    }

    final public Symbol getSymbol() {

        Symbol symbol = null;
        skipBlanks();
        char factor = '1';
        String separator = "";
        if (pos < s.length() - 2
            && s.charAt(pos) == '<'
            && Character.isLetter(s.charAt(pos + 1))) {  // symbol is a non-
terminal
            StringBuilder nonTerminal = new StringBuilder("");
//            pos++;
            // Process up to the first digit, *, ?, or +
            while (pos < s.length() && s.charAt(pos) != '>'
                && s.charAt(pos) != '?'
                && s.charAt(pos) != '+'
                && !Character.isDigit(s.charAt(pos))) {
                    nonTerminal.append(s.charAt(pos));
                    pos++;
                }
                if (s.charAt(pos) != '>') { // A repetition factor must be present
                    if (Character.isDigit(s.charAt(pos))) { // must be numeric
                        // ignore for future extension
                    } else {
                        factor = s.charAt(pos);
                    }
                    // Now check for an optional comma and single character
                    pos++;
                    if (s.charAt(pos) == ',') {
                        pos++;
                        separator = "" + (s.charAt(pos));
                        pos++;
                    }
                    if (s.charAt(pos) != '>') {
                        System.err.println("Expecting a >");
                        System.exit(pos);
                    }
                }
                nonTerminal.append(s.charAt(pos));

                // At this point we have found what looks like a non-terminal.
                // if it is actually <id>, <string>, or <no> then it is a pseudo-NT
                // which means it is actually a terminal
                String nt = new String(nonTerminal);
                if (nt.length() != 0) {
                    symbol = grammar.createSymbol(nt, factor, separator);  // note
that createSymbol handles
                    // issues regarding pseudo-nt's
                } else {
                    System.err.println("Bad grammar non-terminal");
                    System.exit(pos);
                }
                pos++;
            } else {  // the symbol is a terminal
                StringBuilder terminal = new StringBuilder("");
                while (pos < s.length() && s.charAt(pos) != ' ') {
                    terminal.append(s.charAt(pos));
                    pos++;
                }
                String termString = new String(terminal);

                if (termString.length() == 3 && termString.charAt(0) == '$' &&
    termString.charAt(2) == '$') // terrible patching
                {
                    termString = "" + termString.charAt(1);
                }
                if (termString.equals("")) {
                    symbol = null;
                } else {
                    symbol = grammar.createSymbol(termString, factor, separator);
                }
            }

            return symbol;
        }

        /**
         *
         * @return the number of elements in the right-hand side
         */
        public int length() {
            return rule.size();
        }

        /**
         * #return the NonTerminal on the left-hand side
         */
        public Symbol lhs() {
            return rule.get(0);
        }

        /**
         *
         * @return a copy of the rule, with the first element removed
         */
        public LinkedList<Symbol> rhs() {
            LinkedList<Symbol> r = new LinkedList<Symbol>();
            for (int i = 1; i < rule.size(); i++) {
                r.add(rule.get(i));
            }
            return r;
        }

        /**
         *
         * @param i position in the rule
         * @return the Symbol at that position
         */
        public Symbol elementAt(int i) {
            return rule.get(i);
        }

        /**
         *
         * @return the number of Symbol's in the rule
         */
        public int size() {
            return rule.size();
        }

        /**
         *
         * @return the rule as a string
         */
        @Override
        public String toString() {
            int i;
            StringBuffer sb = new StringBuffer();
            sb.append(rule.get(0));
            sb.append(" ::= ");
            for (i = 1; i < size(); i++) {
                sb.append(rule.get(i));
                sb.append(' ');
            }
            return new String(sb);
        }
    }
```

TerminalParser.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
```

```
*/
package parsing;

import scanner.Scanner;

/**
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class TerminalParser extends Parser {
	// ---------------- static variables  ----------------------//

	  final static TerminalParser TERMINAL_PARSER = new TerminalParser();
	// ---------------- Constructors ----------------------//
	public TerminalParser() {}
	public TerminalParser(Symbol s) {
		symbol = s;
	}

	@Override
	public Instruction  parse(Scanner scanner) {
		// Nothing to do.  The keyword has already been recognized

//		Token current = scanner.current();
//		Symbol s = current.getSymbol();
//

		Instruction instruction = new
NopInstruction(scanner.current().getValue());
		scanner.get();
		return instruction;
	}



	// ---------------- Getters/Setters -------------------//
	// ---------------- Other member functions ------------//
}
```

TerminalType.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package parsing;

/**
 *
 * @author lmorell
 */
enum TerminalType {

}
```

Trie.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package parsing;

import java.util.Collection;
import java.util.Enumeration;
import java.util.LinkedHashMap;
import java.util.Set;
import java.util.TreeSet;

/**
 * Represents the rules of a grammar as a trie.  A path from the root of the
trie
 * to a node that indicates the end of a rule is a rule in the grammar.
 *
 * @author Larry Morell (lmorell@atu.edu)
 */
public class Trie {
    // ----------------------- Static variables ------------------------ //

    // -----------------------
    Symbol symbol;  // the contained symbol

    /*
     firstSet is the representation for {t in Terminal | this =>* t alpha}
     Given a terminal symbol t, firstSet(t) yields the trieNode which
     begins the rule that inserted t into firstSet.  Note that any attempt
     to insert the same t twice indicates that the grammar is not LL(1).

     */
    LinkedHashMap<Symbol, TrieNode> firstSet;
    /*
     firstOf represents all the rules in which the current symbol
     X s the leading symbol.  TrieNode tn is an element of firstOf iff
     tn.symbol is a non-terminal N  and N => X ...
     */
    private TreeSet<TrieNode> firstOf;
    TrieNode root = null;
    final static boolean debug = false;
    /*
     parser is the parser for the root symbol of this Trie
     It is likely a GenericParser, which performs recursive descent
     */
    Parser parser;

    /**
     * Build a Trie with a root and no right-hands sides
     *
     * @param s Symbol for which this Trie represents the rules
     */
    public Trie(Symbol s) {
        symbol = s;
        root = new TrieNode(s, this);
        root.instruction = NopInstruction.NOP_INSTRUCTION;
        root.nullablePrev = true;
        firstSet = new LinkedHashMap<Symbol, TrieNode>(); // {t in T | this =>*
t alpha}
        firstOf = new TreeSet<TrieNode>();
        parser = TerminalParser.TERMINAL_PARSER;  // most symbols are keywords,
so default to this
    }

    /**
     * Construct a Trie from a Symbol and a Parser
     *
     * @param s
     * @param p
```

```
 */
public Trie(Symbol s, Parser p) {
    this(s);
    parser = p;

}

/**
 * Gets the symbol at the root of this Trie
 *
 * @return the symbol at the root of the trie
 */
public Symbol getSymbol() {
    return symbol;
}

public Instruction getInstruction() {
    return root.instruction;
}

boolean isNullProduction() {
    return firstSet.size() == 0;
}

/**
 *
 * @return The set of symbols of which this symbol is in the localFirst
 */
public TreeSet<TrieNode> getFirstOf() {
    return firstOf;
}

/**
 *
 * @return The root node of the Trie.
 */
TrieNode getRoot() {
    return root;
}

/**
 *
```

```
     * @return the symbols in the localFirst set of this Trie's symbol
     */
    public Collection<Symbol> getFirstSet() {
        Set<Symbol> keys = firstSet.keySet();
        // If there are no elements in the keySet, then it must be a terminal;
        // Put that terminal in the FirstSet
//        if (keys.isEmpty()) {
//            firstSet.put(symbol, null);
//        }
        // and return it (or whatever was there originally
        return keys;
    }

    /**
     *
     * @return the TrieNodes that correspond to the localFirst set of this
     * Trie's
     * symbol
     */
    public Collection<TrieNode> getFirstNodes() {
        return firstSet.values();
    }

    /**
     *
     * @param s
     * @return the root TrieNode of the tree for rules that begin with s
     */
    TrieNode firstNode(Symbol s) {
        TrieNode tn = firstSet.get(s);
        // if null then there may be a nullable rhs; if so, return it
        if (tn == null) {
            Enumeration children = root.children();
            while (children.hasMoreElements() && tn == null) {
                TrieNode child = (TrieNode) children.nextElement();
                if (child.nullableRest && child.symbol.isNullable()) {
                    tn = child;
                }
            }
        }
        return tn;
    }
    /**
     *
     * @param s
     * @return symbol s is in the localFirst set of this Trie
     */
    boolean firstContains(Symbol s) {
        return firstSet.keySet().contains(s);

    }

    /**
     *
     * @return the parser for this Trie
     */
    public Parser getParser() {
        return parser;
    }

    /**
     * If tn holds a terminal it is added to the first set of this Trie
     * otherwise the localFirst set of of the symbol found in tn is unioned to
     * the
     * First set of this trie.
     *
     * @param tn trie node that holds the symbol to be updated
     * @return The set of TrieNodes newly added to the the localFirst set
     */
    Set<Symbol> addFirstSymbol(TrieNode tn) {
        boolean success = false;
        Symbol s = tn.getSymbol();  // symbol contained in the node
        Set<Symbol> update = new TreeSet<>(); // new symbols that were added
        if (s.isTerminal()) { // add in terminals
            if (firstSet.put(s, tn) == null) {  // add successful, add to
update as well
                update.add(tn.getSymbol());
            } else {
                System.err.println("Duplicate " + s + "added to first set for
" + this.symbol);
            }
        } else { // tn holds a nonterminal that is to be referenced via the
hash

            // First, update the firstOf set of tn
            Trie tnTrie = tn.getSymbol().getTrie();    // Get tn's trie/s
            Set<TrieNode> ts1 = tnTrie.getFirstOf();   // firstOf set and
            TrieNode tn1 = getRoot();                 // add the root of the
current trie
            boolean add = ts1.add(tn1);               // to it.

            // Then for each symbol in the first set of the trie of the symbol
            // fount in tn, add it to the firstSet  of this trie and link
            // tn to it.
            Collection<Symbol> symbols = tnTrie.getFirstSet();  // the symbols
to linked to tn
            for (Symbol sym : symbols) {
                if (true)//sym.isTerminal())
                {
                    if (firstSet.put(sym, tn) == null) { // sym was newly
added, record this
                        update.add(sym);
                    } else {
                        System.out.println("Duplicate " + sym + " added to
first set for " + symbol + "\n");
                    }
                } else {  // Sym is non-terminal, add its elements to the
update
                    System.err.println("Trying to map a non-terminal " + sym +
" to " + tn);
                }
                /* ignore any nonterminals -- in case these are added in the
future */
            } // for
        }  // else

        // Return the set of nodes that were newly added to the localFirst set
of the lhs
        return update;
    } // addFirstSymbol

    /**
     *
     * @param ts set of nodes to be added to the set
     * @return true if a duplicate was found
     */
    boolean addFirstSet(TreeSet<TrieNode> ts) {
        // Need to add each one in separately because if any one is duplicated,
error
        boolean duplicate = false;
        for (TrieNode tn : ts) {
            if (firstSet.put(tn.getSymbol(), tn) != null) {
                duplicate = true;
                System.err.println("Duplicate add of '" + tn + "' to the first
set of trie:\n "
                    + this);
            } else if (debug) {
                System.err.println("Non-Duplicate add of " + tn + " to the
first set of "
                    + this);
            }
        }
        return !duplicate;
    }

    /*
     * Propagate a set terminals to each nonterminal that left-derives them.
     * @param update  The set of terminals to be propagated.
     */
 /*
     private void propagateFirst(Set<Symbol> update) {
     // Union 'update' to the localFirst set of each trie whose root symbol is
s
     // where  this.lhs is in the s.firstOf;

     // For each nonterminal N that left-derives the root of this trie
     //    N.firstSet U= update
     // This is accomplished via a standard info-flow, breadth-localFirst,
     // algorithm, starting localFirst at the root of this trie,
     // and propagating from any node which is updated
     //
     // s.addFirstSet(update) returns false if left recursion occurs.
     // Since update never changes, adding update twice will fail and
     // infinite recursion in therefore avoided.
     LinkedList<Symbol> working = new LinkedList<Symbol>();
     working.add(symbol);  // start from the root of this trie

     while (!working.isEmpty()) {
     Symbol w = working.getFirst();  // get the localFirst element nonterm
     working.remove(w);             // toss it
     // For each s, s =>* w alpha
     for (TrieNode tn : w.getFirstOf()) {
     Trie trieToUpdate = tn.getTrie();
     /*  Fix this
     if (trieToUpdate.addFirstSet(update)) { // update firstset
     working.add(tn.getSymbol());  // propagate from s if s.firstSet is updated
     } else { // firstSet was not updated, meaning there was a duplicate
     System.err.println("Propagage localFirst could not update" +
tn.getSymbol() + " with "
     + update);
     }
     END OF FIx
     * /
     }  // for
     } // while
     }
     */
    /**
     * dest = dest + src
     *
     * @param dest the set to be updated
     * @param src the set of elements for the update
     * @return the elements in src not found in dest before the
     */
    Set<Symbol> add(Set<Symbol> dest, Set<Symbol> src) {
        Set<Symbol> update = new TreeSet<>();
        for (Symbol tn : src) {
            if (dest.add(tn)) {
                update.add(tn);
            } else {
                System.err.println("Duplicate element added to first set");
            }
        }
        return update;
    }

    Symbol add(Set<Symbol> dest, Symbol src) {
        if (!dest.add(src)) {
            System.err.println("Duplicate element added to first set");
        }
        return src;
    }

    /**
     * Add the symbol nt to the localFirst set of each TrieNode location for
the
     * NT
     * of the current
     * Trie
     *
     * @param s
     */
    void propagateFirst(Symbol s) { // s is a nonterminal that is nullable
        /*
         * We begin with the nonterminal
         */
//        if (s.isNonTerminal()) {
//            System.err.println("Attempting to add Symbol " + s
//                + " to first sets of" + symbol);
//            return;
//        }

        // For all the locations where the current non-terminal occurs
        for (TrieNode tn : symbol.getLocations()) {
            // Propagate the symob s up to the localFirst nullable or root
            add(tn.localFirst, s);
            TrieNode parent = (TrieNode) tn.getParent();
            while (parent != parent.trie.root && parent.symbol.isNullable()) {
                add(parent.localFirst, s);
                tn = parent;
                parent = (TrieNode) parent.getParent();
            }

            // if at root, then put the symbol into the  trie'sroot's firstSet
            if (parent == parent.trie.root && tn.symbol.isNonTerminal()) {
                // Update the root's firstSet
                Trie trie = tn.getTrie();
                if (trie.firstSet.put(s, tn) == null) {  // Addition was
successful; propagate localFirst
//                    if (trie.firstSet)
                    trie.propagateFirst(s);
                } else {
                    System.err.println("Adding duplicate of " + s
                        + "to a first set for " + symbol);
                }

            }
        }

    }

    /**
     * Propagates derives lambda to all nonterminals which left-derive nt.  and
     * updates the localFirst sets, if needed
     *
     * @param nt The nullable nonterminal that begins the propagation
     */
    void propagateDL(Symbol nt) { // nt is a nonterminal that is nullable
        /*
         * We begin with the nonterminal
         */
        if (nt.isTerminal()) {
            return;
        }
        nt.setDerivesLambda(true);

        Symbol n = nt;
        // For all the locations that n occurs
        for (TrieNode tn : n.getLocations()) {
            // Propagate the localFirst set of tn up to the nullable
            Set<Symbol> update = new TreeSet<>();
            // Accumulate all the localFirst sets from
            // each of the children and put them into the localFirst set of
            // the current node, tn.
            Enumeration children = tn.children();
            while (children.hasMoreElements()) {
                TrieNode child = (TrieNode) children.nextElement();
                child.nullablePrev = true;
                if (child.nullableRest) {
                    tn.nullableRest = true;
                }
                // We have possible new localFirst elements for our NT in tn
                add(update, child.getLocalFirst());
            }
            // Update is now the union of tn's child's local first sets
            // Place it into the local first set of tn
            add(tn.localFirst, update);
            TrieNode prev = tn;
            tn = (TrieNode) tn.getParent();
            while (tn != tn.getTrie().root && prev.symbol.derivesLambda()) {
                add(tn.localFirst, update);
                if (prev.nullableRest && prev.symbol.isNullable())
                    tn.nullableRest = true;
                prev = tn;  // Save the child ascended from
                tn = (TrieNode) tn.getParent();
            }

            // Check to see if tn is at the root
            if (tn == tn.trie.root
                //                && tn.symbol.isNonTerminal()
                && prev.nullableRest
                && prev.symbol.derivesLambda()) {  // last condition to prevent
loops

                tn.symbol.setDerivesLambda(true);

                // Update the root's firstSet, by linking it to tn
                for (Symbol s : update) {
                    if (tn.getTrie().firstSet.put(s, prev) == null) {  //
Addition was successful; propagate localFirst
                        tn.trie.propagateFirst(s);
                    } else {
                        System.err.println("Adding duplicate of " + s
                            + "to a first set for " + symbol);
                    }
                }

            }
        }

    }

    public void setParser(Parser p) {
        parser = p;
    }

    /**
     * Adds a rule to the trie.
     *
     * @param r The rule to be added.
     */
    void addRule(SyntaxRule r, Instruction instr, Parser p) {
        /* This is the heart of the class.  A rule is added by descending
         * through the existing Trie, adding nodes as needed.
         * The following invariants are maintained for the trie's nodes:
         *    nullablePrev = true iff the path from a root to the parent of
node
         *                         is nullable
         *    nullableRest = true iff a path after the current node to the
         *                end of the rule is nullable
         */

        parser = p;
        int i;
        Symbol lhs = r.elementAt(0);
        System.out.println("Adding grammar rule " + r);

        if (!r.lhs().equals(symbol)) {
            // This should never happen
            System.err.println("Attempting to add a rule with lhs of " +
r.lhs()
                + "to a Trie for " + root);
            System.exit(1);
        }

        //  For each symbol in the rhs rule r, add it to the trie
        TrieNode tn = root;

        tn.nullablePrev = true;  // for propagation purposes

        Symbol s;

        if (r.size() > 1) {  // not the empty rule (null production)
            // add in the localFirst symbol;
            TrieNode topOfRHS = tn;
            Set<Symbol> trieUpdate = new TreeSet<>();
            tn = tn.addChild(r.elementAt(1));  // First member of the rhs
            if (tn.newNode()) {
                tn.nullablePrev = true;    // null occurs before the localFirst
of rhs
                addFirstSymbol(tn);
                trieUpdate.add(tn.symbol);
                // Add in the firstSet from the symbol
                Collection<Symbol> firstSet = tn.symbol.getFirstSet();
                tn.localFirst.addAll(tn.symbol.getFirstSet());
            }

            for (i = 2; i < r.size(); i++) {
                s = r.elementAt(i);
                tn = tn.addChild(s);
                if (tn.newNode()) {  // newly added node
                    TrieNode parent = (TrieNode) tn.getParent();
                    TrieNode child = tn;
                    if (parent.nullablePrev && parent.symbol.derivesLambda()) {
                        tn.nullablePrev = true;
                    }
                    Set<Symbol> update = new TreeSet<>();
                    Collection<Symbol> firstOfChild =
child.symbol.getFirstSet();
                    for (Symbol sym : firstOfChild) {  // for each symbol in
first(child)
                        if (tn.localFirst.add(sym)) // add it to tn's
localFirst
                        {
                            update.add(sym);// and to update
                        }
                    }
                    // For each nullable parent starting with tn's parent,
                    // update its localFirst set using the newly added elements
                    // to tn
                    while (parent != parent.getTrie().root &&
parent.symbol.derivesLambda()) {
                        add(parent.localFirst, update); // propagate update
upwards
                        child = parent;
                        parent = (TrieNode) parent.getParent();
                    }
                    if (parent == root && child.symbol.isNullable()) {
                        // we are at the root of the Trie and it is nullable
                        // so we have to update the trie's firstSet
                        for (Symbol sym : update) {
                            if (firstSet.put(sym, child) == null) {
                                trieUpdate.add(sym);
                            } else {  // already there
                                System.err.println("Error: Adding " + sym
                                    + " to first set for " + symbol);
                            }
                        }
                    }
                }
            }
            // We have just added in the last TrieNode of a rule, tn
            tn.setEndOfRule(true);
            if (tn.isLeaf()) {
                tn.nullableRest = true;
            }

            // If we've changed the firstSet for the trie, propagate it
            if (!trieUpdate.isEmpty()) {  // First(lhs) updated; propagate
                for (Symbol sym : trieUpdate) {
                    propagateFirst(sym);
                }
            }

            // If this and all previous nodes are nullable, propagate root
symbol
            if (tn.symbol.derivesLambda() && tn.nullablePrev) {
                symbol.setDerivesLambda(true);
                propagateDL(tn.symbol);
            }

        } // rule size > 1
        else { // we have a null production; propagateDL
            s = r.elementAt(0);
            symbol.setDerivesLambda(true);
            System.out.println("Adding null production for " + s);
            propagateDL(s);                    // Add in the instruction at the
root of the trie
```

```
        }
        // tn is at root if this is a null production, otherwise
        // it is
        tn.instruction = instr;
        // System.out.println("--------------------");
    }

    void dftraverse(TrieNode tn, int indentation, StringBuffer buffer
    ) {
        if (tn == null) {
            return;
        }
        for (int i = 0; i < indentation; i++) {
            buffer.append(' ');
        }

        buffer = buffer.append(tn);
        buffer.append("\n");
        int count = 0;
        // Ok, I simply don't get this.  To get rid of an "unchecked
conversion"
        // I had to drop the generic from Enumeration<TrieNode> e.
        // But then I had to cast e.nextElement(), which I personally consider
        // worse.  So why doesn't the compiler complain about an "unchecked
cast"
        // at that point when converting the object returned by nextElement???
        // I am mystified.
        // In looking online, I found that Java does not know the parameter
        // type at runtime which means that a weird situation can occur in
        // which a variant on a parameterized type can be assigned to a
variable,
        // but no error will be issued until a member of the variable.  The
        // error would be like "wrong class exception".  The same article
indicates
        // that you cannot create an array whose base type is a parameterized
        // type, b/c an A[] is super class of B[] if A is a super class of B.
        // However A[] is not a super class of A<E>[].
        // Apparently it is more difficulty to debug a situation in which
        // the program dies due to a method is invoked on an object that
        // does not have that object, by, say, assigning a List<String>
        // to a List<Integer>, and then trying to invoke a List<Integer>
        // operation on it.  (Java apparently does not make the type tag
        // available at runtime, so it will allow the statement.  My
        // remaining question is, how does it know that wrong operation is
        // being invoked?  Is is possible some operations will actually
        // work?  If it detects an error for each operation, why cannot
        // the JRE detect the error as assignment?

        Enumeration e = tn.children();
        indentation += 3;
        while (e.hasMoreElements()) {
            dftraverse((TrieNode) e.nextElement(), indentation, buffer);
//          buffer.append("\n");
        }
        indentation -= 3;
    }

    @Override
    public String toString() {
        StringBuffer result = new StringBuffer(1000);
        dftraverse(root, 0, result);
        return result.toString();
    }
}
```

TrieNode.java

```
package parsing;

import java.util.Collection;
import java.util.Enumeration;
import java.util.Set;
import java.util.TreeSet;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import javax.swing.tree.DefaultMutableTreeNode;

/**
 * Implements the nodes used in a Trie.  Each node holds a symbol and related
 * information
 *
 * @author lmorell
 */
public class TrieNode extends DefaultMutableTreeNode
```

```
    implements Comparable<TrieNode> {
    // class fields

    Symbol symbol;             // aliased to userObject
    Instruction instruction;  // Component for an instruction list
    boolean nullablePrev;     // Path to this node is nullable
    boolean nullableRest;     // A path from a child to a leaf is nullable
    boolean endOfRule;        // Does this node form the end of a rule?
    boolean justAdded;        // true iff this is a new node
    Trie trie;                // Referent to the trie of which this node is a
part
    // TrieNode parent = null;
    // Boy, I don't mind wasting space.  Really though, there
    // are not likely to be more than a 1000 nodes in this tree
    // so does it really matter?
    final static boolean debug = false;
    /*
     localFirst contains the first set of the symbol of this TrieNode and,
     if that symbol is nullable, then it also contains the union of
     all the firstSets of all its successors in the trie that are reachable
     via a nullable path
     */
    Set<Symbol> localFirst;

    public Set<Symbol> getLocalFirst() {
        return localFirst;
    }

    /**
     *
     * @param tn TrieNode to be compared with
     * @return neg, 0, pos based on the comparing the this.symbol with
tn.symbol
     */
    public int compareTo(TrieNode tn) {
        return symbol.compareTo(tn.symbol);
    }

    /**
     * Construct a TrieNode from a Symbol and a Trie
     *
     * @param s
     * @param t
     */
    public TrieNode(Symbol s, Trie t) {
        userObject = s; // These should remain in tandem
        symbol = s;      // because all the implementation code below uses
symbol
        nullablePrev = false;
        nullableRest = false;
        endOfRule = false;
        trie = t;
        justAdded = true;
        instruction = null;
        localFirst = new TreeSet<>();
//                Collection<Symbol> firstSet = s.getFirstSet();
//        localFirst.addAll(s.getFirstSet()); // Initialize with its symbol's
firstSet
    }

    // Getters and setters
    public Instruction getInstruction() {
        return instruction;
    }

    public void setInstruction(Instruction instruction) {
        this.instruction = instruction;
    }

    public Set<Symbol> getFirstNodes() {
        Set<Symbol> ts = new TreeSet<>();
        if (symbol.isNonTerminal()) { // Copy in all the symbols in the
localFirst sets
            for (TrieNode tn : symbol.getTrie().getFirstNodes()) {
                if (tn != null) // Can be null if no rules yet for tn
                {
                    ts.add(tn.getSymbol());
                }
            }
            return ts;
        } else {  // it is terminal
            ts.add(symbol);
            return ts;
        }
    }

    public Symbol getSymbol() {
        return symbol;
    }

    public Trie getTrie() {
        return trie;
    }

    boolean isTrieRoot() {
        return trie.root == this;
    }

    /**
     *
     * @param i The position of the child from which to retrieve the symbol
     * @return The node added
     */
    public Symbol getChild(int i) {
        return ((TrieNode) getChildAt(i)).symbol;

    }

    /**
     *
     * @param symbol The symbol sought
     * @return
     */
    /**
     *
     * @param symbol
     * @return The child which has symbol in its localFirst set
     */
    public TrieNode getChild(Symbol symbol) {
//          Collection <TrieNode> children = tn
        Enumeration nodes = children();
        boolean found = false;
        TrieNode tn = null;
        TrieNode nullableNode = null;
        // It is ugly code like this that demands a change to the programming
language
```

```
        // or at least the design of enumerators
        while (!found && nodes.hasMoreElements()) {
            tn = (TrieNode) nodes.nextElement();
            Collection<Symbol> firstSet = tn.getLocalFirst();

            found = firstSet.contains(symbol);
            // Save the node tn if it is nullable
             if ( ! found &&  tn.symbol.isNullable()) nullableNode = tn;
            // At this point we are looking for a child non-terminal
            // which has symbol as a prefix.  Or we are looking for
            // the symbol itself.  The problem is what
            // happens when the non-terminal is nullable? In that
            // case we return that non-terminal iff symbol is in
            // the localFirst of the remaining rhs.
            // First (remaining RHS) = localFirst  (X) if rhs = X...  and
            //                         X is not nullable
            //                       = localFirst(X) U localFirst (...) if X is
nullable
            // Again, here I am assuming that localFirst never contains e
            // Note that when a new nonterminal is discovered to be
            // nullable, then wherever that non-terminal occurs,
            // the localFirst sets of the remaining rhs must now be propagated
            // up as well as the nullability (which may need to be
            // propagated down???)
        }
        if (found) {
            return tn;
        } else  {
            // The only possible node in the trie that can be followed is the
            // nullable node since the symbol is not the prefix of any of the
            // successors
            return nullableNode;
        }
    }

       /**
     *
     * @param symbol The symbol sought
     * @return
     */
    /**
     *
     * @param symbol
     * @return The possible termial of current nontermial that could expend
     */
    public Symbol getTermial(Symbol symbol) {
//        Collection <TrieNode> children = tn
        Enumeration nodes = children();
        boolean found = false;
        TrieNode tn = null;
        TrieNode nullableNode = null;
        // It is ugly code like this that demands a change to the programming
language
        // or at least the design of enumerators
        while (!found && nodes.hasMoreElements()) {
            tn = (TrieNode) nodes.nextElement();
            Collection<Symbol> firstSet = tn.getLocalFirst();
            while (firstSet.iterator().hasNext())
            {
                Symbol s = firstSet.iterator().next();
                if (s.isTerminal())
                        return s;
            }
        }
        return null;
    }

    public boolean isEndOfRule() {
        return endOfRule;
    }

    /**
     *
     * @return The number of children of the node
     */
    public int numberOfChildren() {
        return getChildCount();
    }

    public void setEndOfRule(boolean endOfRule) {
        this.endOfRule = endOfRule;
    }
```

```
public boolean newNode() {
    return justAdded;
}
// Mutators

/**
 * Add a symbol as a child of the current node, sorted by s, if it does not
 * exist.  If it does exist, just return it
 *
 * @param s
 * @return
 */
TrieNode addChild(Symbol s) {
    TrieNode tn = null;
    int size = getChildCount();
    int i;
    // Insert the symbol s in alphabetical order
    // Find its insertion location; linear search -- no body
    for (i = 0; i < size && s.compareTo(getChild(i)) > 0; i++) ;

    // Insert before existing child if i < size
    if (i < size) { // found a position to insert s into
        if (s.compareTo(getChild(i)) < 0) { // insert at position i
            tn = new TrieNode(s, trie);
            insert(tn, i);
            // Link this into the list of TrieNodes
            // associated with the corresponding symbol
            s.locations.addFirst(tn);
        } else { // the value is already at position i
            tn = (TrieNode) getChildAt(i);
            tn.justAdded = false;
        }
    } else { // The value was not found, append it
        tn = new TrieNode(s, trie);

        add(tn);
        s.locations.addFirst(tn);  // link this into the list of TrieNodes
        // associated with the corresponding symbol
    }

    return tn;  // returning this directs the calling alg to continue at tn
}
```

```
    public Instruction createInstruction(InstructionList il) {
        if (instruction == null) {
            System.err.println("Attempt to create an instruction from a null
instruction");
            return null;
        }
        return instruction.createInstruction(il);
    }

    @Override
    public String toString() {
        if (debug) {
            return symbol.toString() + " [" + nullablePrev + "," + nullableRest
+ "]";
        } else {
            return symbol.toString();
        }
    }
}
```

UnsignedId.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package parsing;


import parsing.GenericParser;
import parsing.Instruction;
import parsing.InstructionList;
import parsing.SyntaxRule;
import runtime.DoubleVal;
import runtime.EnvironmentVal;
import runtime.StringVal;
import runtime.Value;

/**
 *
 * @author Larry Morell
 */
public class UnsignedId extends Instruction {

    private static String rule = "<UnsignedId> ::= <id>";
    private static SyntaxRule syntaxRule ;
    public static void initialize(Grammar GRAMMAR) {  // static initialization
        syntaxRule  = new SyntaxRule(GRAMMAR, rule);

       GRAMMAR.addRule(syntaxRule,
               new UnsignedId(),
               new GenericParser(syntaxRule.lhs()));
    }
    private String id;


    public UnsignedId() {
        id = "Unknown";
    }

    public UnsignedId(UnsignedId a) {
        id = a.id;
    }
    public UnsignedId(String value) {
        id = value;
    }

    public String value() {
        return id;
    }

    @Override
    public Instruction createInstruction(InstructionList il) {
        return il.getFirst();
    }
    public Instruction copy (UnsignedId a)
    {
        Instruction result = new UnsignedId(a.value());
        return result;
    }
    @Override
    public Value eval() {
        return new StringVal(id);
    }

    @Override
    public Value eval(EnvironmentVal env) {
        return new StringVal(id);
    }

    @Override
    public InstructionList exec() {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }
   public String toString() {
    return id;
}
}
```

UnsignedNumber.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package parsing;


import parsing.GenericParser;
import parsing.Instruction;
import parsing.InstructionList;
import parsing.SyntaxRule;
import runtime.DoubleVal;
import runtime.EnvironmentVal;
import runtime.Value;

/**
 *
 * @author Larry Morell
 */
public class UnsignedNumber extends Instruction {

    private static String rule = "<UnsignedNumber> ::= <no>";
    private static SyntaxRule syntaxRule ;
```

```
public static void initialize(Grammar GRAMMAR) {  // static initialization
    syntaxRule  = new SyntaxRule(GRAMMAR, rule);

   GRAMMAR.addRule(syntaxRule,
            new UnsignedNumber(),
            new GenericParser(syntaxRule.lhs()));
}
private double number;
Value numberNote;

public UnsignedNumber() {
    number = 0;
}
public UnsignedNumber(UnsignedNumber a) {
    number = a.number;
    numberNote = new DoubleVal(a.number);
}
public Instruction copy (UnsignedNumber a)
{
    Instruction result = new UnsignedNumber(a.value());
    return result;
}
public UnsignedNumber(double d) {
    number = d;
    numberNote = new DoubleVal(d);
}

public UnsignedNumber(String value) {
    this(Double.parseDouble(value));
}

public Double value() {
    return number;
}

@Override
public Instruction createInstruction(InstructionList il) {
    return il.getFirst();
}

@Override
public Value eval() {
        return numberNote;
    }

    @Override
    public Value eval(EnvironmentVal env) {
        return numberNote;
    }

    @Override
    public InstructionList exec() {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }
   public String toString() {
    return numberNote.toString();
}
}
```

UnsignedNumberParser.java

```
/*
 *  Copyright (C) 2010 Larry Morell <morell@cs.atu.edu>
 */
package parsing;

import scanner.Scanner;
import scanner.NumberToken;

/**
 *
 * @author Larry Morell
 */
public class UnsignedNumberParser extends TerminalParser {

    public UnsignedNumberParser(Symbol s) {
        super(s);
    }

    @Override
    public Instruction parse(Scanner scanner) {
        NumberToken t = (NumberToken) scanner.current(); //
getGrammar().currentToken();
```

```
        double number = t.getNumber(); //Integer.parseInt(t.getSymbol().value);

        Instruction result = new UnsignedNumber(number);
        return result;
    }
}
```