Juan Carlos Hernandez Puebla
Professor Steinmetz
CSC 450
April 28, 2016

Software Design Document

One of the most important component involved in the execution of a computer program, is a compiler. A compiler translates a source program written in one language to a target program in another language. An example of this is a compiler that translates Pascal programs to MIPS assembly language programs. This precise type of compiler, based on Java programming language, will be discussed on this document. In particular, two key parts of the compiler, that is the Scanner and the Parser will be explained in further detail. The source program language used for this compiler is a variation of Pascal. Its lexical convention is listed below, along with a UML diagram that illustrates all the classes and their relationship used for this compiler.

Lexical Conventions

1. Comments are surrounded by **{** and **}**. They may not contain a {. Comments may appear after any token.
2. Blanks between tokens are optional.
3. Token **id** for identifiers matches a letter followed by letter or digits:

letter -> [a-zA-Z] digit
-> [0-9]
id -> letter (letter | digit)*

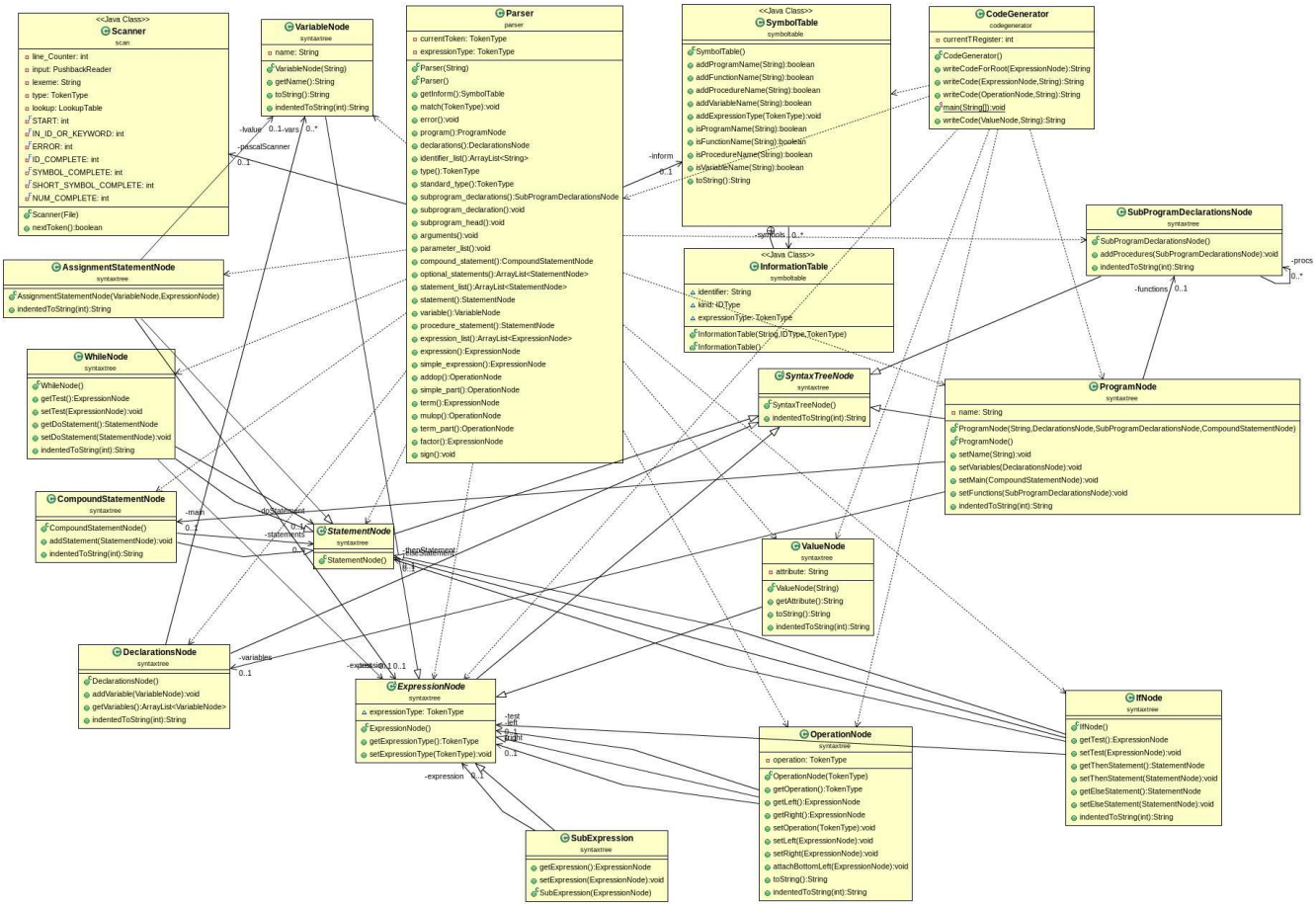
The * indicates that the choice in the parentheses may be made as many times as you wish.

1. Token **num** matches numbers as follows: **digits -> digit digit***
   optional_fraction -> . digits | λ

optional_exponent -> (E (+ | - | λ) digits) | λ num ->
digits optional_fraction optional_exponent

2. Keywords are reserved.
3. The relational operators (**relop**'s) are: **=**, **<>**, **<**, **<=**, **>=**, and **>**.
4. The **addop**'s are **+**, **-**, and **or**.

5. The **mulop**'s are ***, */*, **div**, **mod**, and **and**.
6. The lexeme for token **assignop** is **:=**.



Scanner

The Scanner's main task is to examine every lexeme recognized in the Pascal language. A lexeme is a series of characters that match the pattern for a token. This particular Scanner implementation has an attribute and token type associated for each token. There are token types for each keyword and symbol in the Pascal grammar. The following list includes all the keywords and symbols.

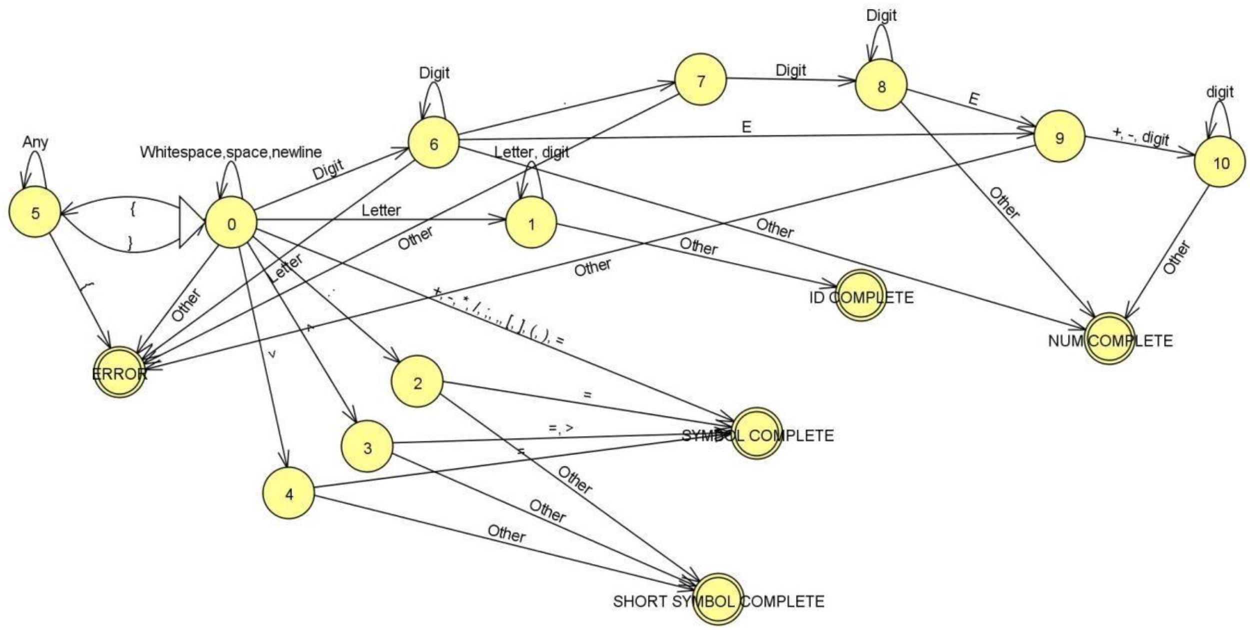
Keywords and Symbols List

keywords: "program"
"if"
"var"
"array"
"of"
"integer"
"real"
"function"
"procedure"
"begin"
"end"
"then"
"else"
"while"
"do"
"or"
"div"
"mod"
"and"
"not"

symbols:
"{"
"}"
"+"
"-"
"*"
"/"
":="
";"
","
"." ":"
"["
"]"
"("
")"
"="
"<>"
"<"
"<="
">="
">"

The Scanner is able to associate a lexeme with a specific Token Type through implementation of the finite state machine diagram below. The Scanner has final constant variables such as ID_COMPLETE that represent the five complete states. It also has 12 incomplete states, two of which are the final constant variables
START and IN_ID_OR_KEYWORD, the rest are represented by case values. The process of the Scanner starts with it taking in an input file to read from. If there is no valid input file found it terminates through the System's exit function. Else, at this point the state number is 0, representing the START state and the current lexeme is “”. The Scanner proceeds to read the first character. Depending on the value of the character, it starts traversing through the correct path within the states. As the Scanner traverses through the states, the state number is updated and the current lexeme is concatenated with what is read. Once it reaches one of the final states, it assigns the current lexeme to lexeme. Then it assigns the lexeme its corresponding token type through a look up table. The look up table uses a hash table to map a keyword or symbol to its corresponding token type. The token types are declared within an enum.

Note, the hash table does not contain a token type for identifiers and numbers since there are infinitely amount of lexeme representations for these token types. Therefore, the lookup table returns null when the state number ends at ID_COMPLETE or NUM_COMPLETE. The appropriate token type of ID or NUM is set after the look up. Now, if the state number ends up at the error state, the token type is assigned null. Furthermore, the entire current lexeme that was read when the error was detected is assigned as the lexeme.



Parser

The Parser is responsible for recognizing valid written Pascal programs using the given grammar. This also includes being able to recognize invalid written Pascal programs. This Parser uses a one token look ahead implementation to accomplish this task. With the use of the Scanner it retrieves the first token. The Parser then uses functions that represent the production rules of the Pascal language to test for valid programs. The first of these functions that is called from the Parser is the program function.

Once this is called, it starts matching the expected Token Type with the current Token Type of the token that was read from the Scanner. The very first Token Type should be PROGRAM for any possible valid Pascal program. If this is not the case, then the error function is called. The error function throw an Error that prints the line where the error occurred. If the error function is not called, then the Parser continues to get the next token from the Scanner. It keeps matching it to its corresponding token type given the production functions that are called as it is reading through the Pascal program. This process is done until it has reached the end of a valid Pascal program, where the

last token should have the token type value of DOT. If at any time the expected Token Type does not match the current Token Type, then the error function is immediately called. The production rules of the Pascal programming language are listed below.

Production Rules

program -> **program id ;** *declarations*
subprogram_declarations
compound_statement
.
identifier_list -> **id**
|
id , *identifier_list*
declarations -> **var** *identifier_list* **:** *type* **;** *declarations* |
λ
type -> *standard_type* |
array [num : num] of *standard_type*
standard_type -> **integer** |
real
subprogram_declarations -> *subprogram_declaration* **;**
subprogram_declarations | *λ*
subprogram_declaration -> *subprogram_head*
declarations subprogram_declarations
compound_statement
subprogram_head -> **function id** *arguments* **:** *standard_type* **;** | **procedure**
id *arguments* **;**
arguments -> **(** *parameter_list* **)** | λ
parameter_list -> *identifier_list* **:** *type* |
identifier_list **:** *type* **;** *parameter_list*
compound_statement -> **begin** *optional_statements* **end**
optional_statements -> *statement_list* | *λ*

statement_list -> *statement* | *statement* **;**
statement_list statement -> *variable*
assignop *expression* |
procedure_statement |
compound_statement |
if *expression* **then** *statement* **else** *statement* |
while *expression* **do** *statement* | *read (id)* |
write (expression) variable -> **id** | **id [**

expression **]** *procedure_statement* -> **id** | **id (**
expression_list **)** *expression_list* ->
expression | *expression* **,** *expression_list*
expression -> *simple_expression* |
simple_expression **relop** *simple_expression*
simple_expression -> *term simple_part* | *sign*
term simple_part simple_part -> **addop** *term*
simple_part | λ
term -> *factor term_part term_part* ->
mulop *factor term_part* | *λ factor* ->
id | **id [** *expression* **]** | **id (**
expression_list **)** |
num |
(*expression* **)** | **not**
factor
sign -> **+** |
-

Symbol Table

One major component that the Parser will make use of is the Symbol Table. The Symbol Table stores important information about an identifier. This includes its name, type, whether it corresponds to an array. If applicable to the identifier, what are its start index and end index. The Symbol Table has four add functions, one for program names, one for function names, one for procedure names, and one for variable names. These functions add an identifier to the Symbol Table if it is not already in the table. It also has four is functions that correspond to the four add functions. These verify whether a given identifier is of a specific type.

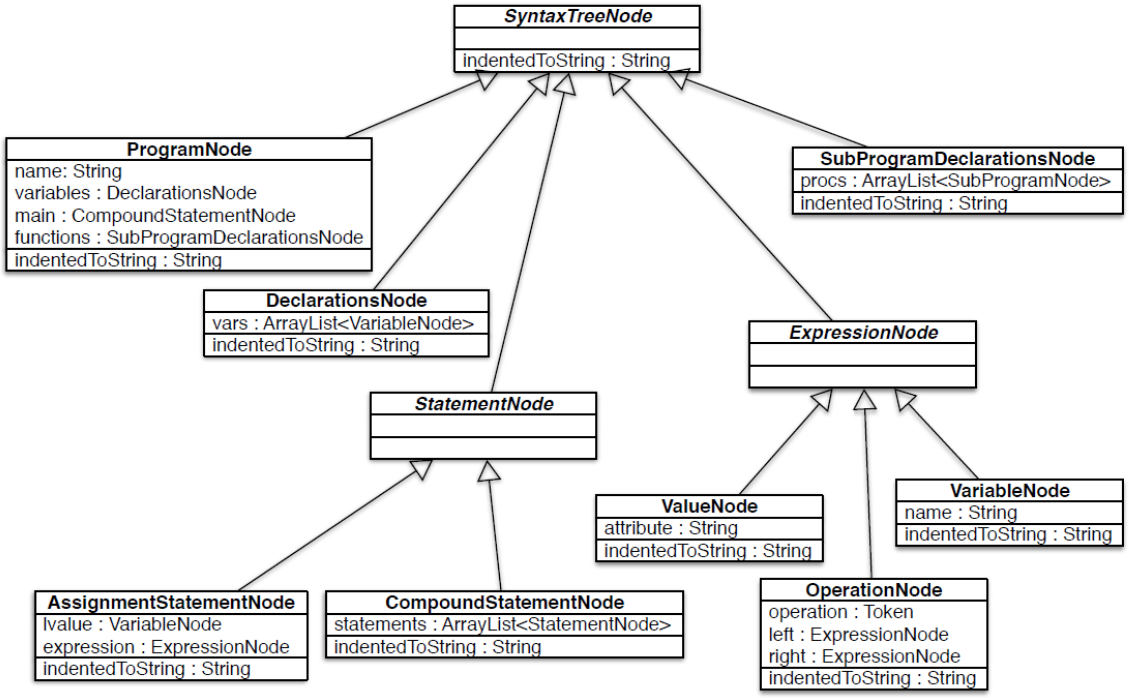
The Parser makes use of the Symbol Table by adding identifiers to it where one is declared. It also performs lookups for identifiers that are used for a task. One must first declare a variable for it to be used, hence why a lookup is needed each time it comes across an identifier that is being used.

Syntax Tree

Another important component of our Pascal compiler is the Syntax Tree. The Syntax Tree abstractly represents the structure of a given source code. When the source code is being parsed, a syntax tree is built through individual nodes. Each represents a significant programming constructs. Listed below are the nodes associated with the Syntax Tree. The way the Syntax Tree is generated, is explained further.

The following nodes, Program Node, Declarations Node, SubprogramDeclarationsNode, AssignmentStatementNode, CompoundStatementNode, ValueNode, VariableNode, and OperationNode, each have their own indentedToString function. The indentedToString function helps represent each node within the entire syntax tree. Depending on where the node rests, it will have a particular indentedToString function to represent itself. For example, the Value Node will be represented as a leaf node, meanwhile the OperationNode will be represented as an internal node, and the ProgramNode will be represented as a base node.

The present integration of the Syntax Tree to Parser correctly produces syntax trees for programs with simply declared variables, as well as programs with expression such as 5 * 7 – 6 + 3 * (2 / 4), and for programs that have an if statement, an else statement, and while statements.



Code Generator

The purpose of the Code Generator is to generate MIPS assembly code from a given Pascal program. First, it writes out the necessary root code to set up a MIPS assembly program. Then it writes out the needed registers for a declared expression.