Software Design Document

Michael Farghali

April 28, 2016

1 Introduction

This Software Design Document (SSD) will give a very high overview of my Pascal complier which is based on the grammar rules given at the start of the course. The goal of this keystone course was to write a compiler in Java that accepts a text file representing a reduced version of Pascal. The text file will be scanned, parsed, converted into a syntax tree and finally written to an output file in assembly language. Upon completion of the course an executable .jar file should be submitted that does the above along with a user manual that explains how to run the program and any possible error messages.

2 Scanner

The purpose of the scanner class is to read in the source code and match all valid keywords and IDs with their respective TokenType which are in the LookupTable class. It is based on a Deterministic Finite Automaton, Fig. 1, and the given grammars seen in appendix B. Each state in the DFA represents a state in the scanner.

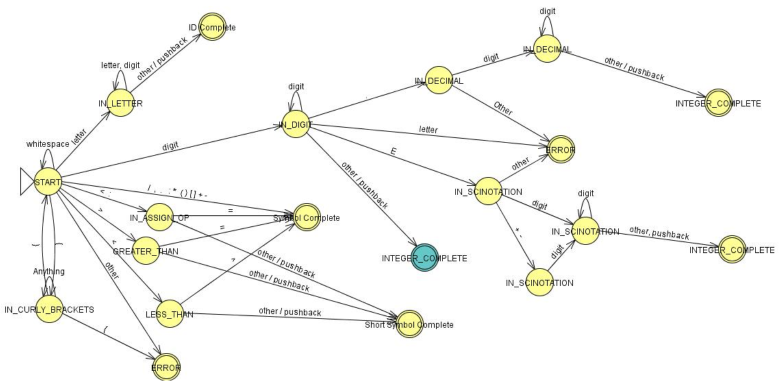


Figure 1: DFA

The scanner passes information to the Parser class primarily through the methods `nextToken`, `getToken`, and `getLexeme`. Most of the work done in the scanner class is done in the nextToken method which is responsible for checking that keywords are valid in our language and that IDs are also in a format allowed and returns true is so or false if not. The getToken method will be used by the parser to grab the current token from the scanner. Finally the getLexeme method is used to pass the name of a variable or invalid statement in the source code, to the parser.

I chose to deviate from the example given in class and added methods `pushBackChar` and `getNextChar`. These methods don't do anything new but the code is used multiple times in the scanner class so I decided to create their own functions to clean up the nextToken method. I also added a method `getCount()` which simply returns the line number where the scanner encounters an unexpected or invalid string. The UML for the scanner class can be seen below.

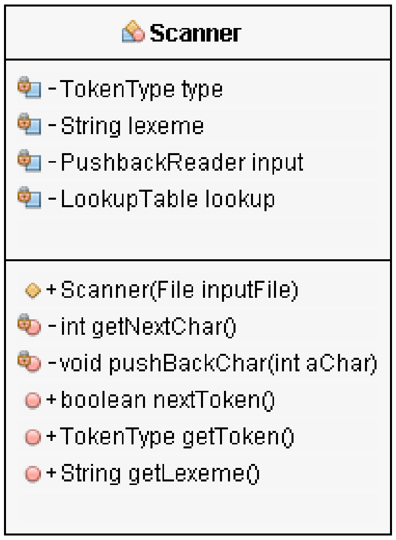


Figure 2: Scanner

2.1 Testing

To test the scanner I created a JUnit test that tested all the methods focusing mainly on the `nextToken` method. I tested it using several valid and invalid strings. Some examples

3 Parser

The Parser class is responsible for a large portion of the work being done in this compiler. It uses the Context-Free grammar rules listed in Appendix B and a push down automata to either accept or reject a source file. It uses top down recursive decent parsing which is convenient for implementation as all 24 nonterminal rules in our grammar become methods in the code. The UML diagram can be seen in 3.

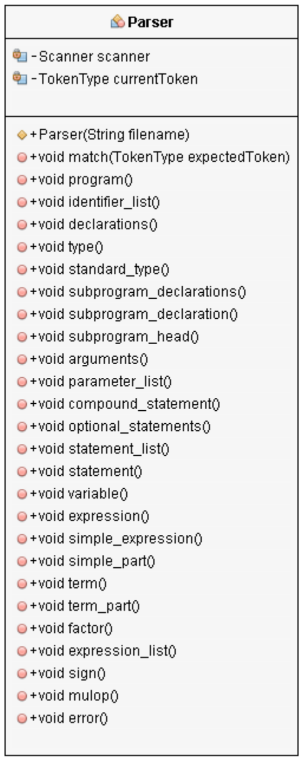


Figure 3: Parser UML

In the parser I chose not to follow the grammar rules exactly. I added methods `addop` and `mulop` which determine which kind of arithmetic or multiplication operator is being used. Adding these two functions helped to clean up the code and make it more readable in my opinion. I also deleted two methods, `simple_part` and `term_part` and added their functionality to `simple_expression` and `term`. To me this seems more straightforward and easy to follow.

The parser class communicates with the Scanner, Symbol Table, and all the classes in Syntax Tree. It uses the scanner to get tokens and the name or string that the token represents. The symbol table is built up and printed to a file in the parser. The parser also creates a condensed syntax tree and if there is an error in the syntax this is where it should be caught.

3.1 Testing

The parser was tested with multiple files containing both valid and invalid code. It tested for simple programming errors such as a missing semicolon as well as syntax error.

3.2 Semantic Analysis

In the semantic analysis stage of the complier three errors should be handled without ending the compilation process. First, variables and assignments should be assigned their data type so that variables of type real can be assigned integer values but not the other way around. There should also be a check that variables are declared before being used. Finally, the syntax tree should print out the variable or expression's type.

We were given the option of implementing the semantic analysis code in the parser or in a separate module. I chose to include it in the parser by adding the function `undeclaredVarError` to handle undeclared variables. To add the variable and expression type to the symbol table I was only able to add their type at declaration time.

4 Symbol Table

The Symbol Table class implements a hashtable and data structure to keep track of variables, functions, procedures, and the program ID. It should also handle arrays and their index. By integrating the symbol table into the parser it is possible for the parser to do the semantic analysis by checking if an ID is in the table. I implemented my symbol table by creating a separate method to add each kind of ID, program, array, function, procedure, and variable. I also created a method to check if each type is already in the table thus allowing the parser to check if the same ID is being used more than once in the same scope. The symbol table should also be used in the parser to handle function and variable scope. The UML diagram can be seen in Fig. 4.

4.1 Testing

Testing the symbol table was a simple task. I checked that the same ID couldn't be added to the table if it already existed and also checked that if it already existed in the table it would return true.

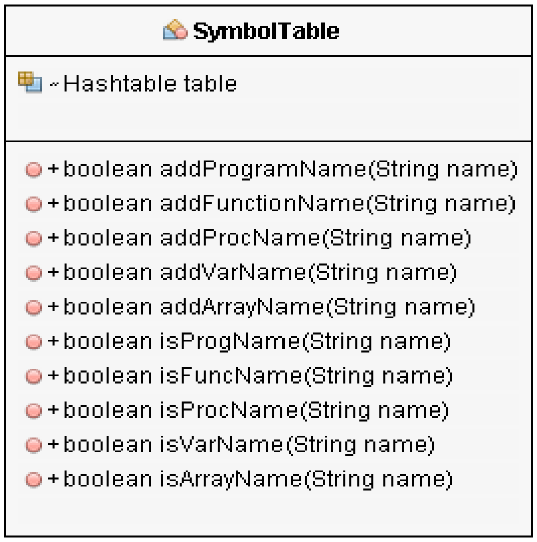

SymbolTable
~Hashtable table
+boolean addProgramName(String name)
+boolean addFunctionName(String name)
+boolean addProcName(String name)
+boolean addVarName(String name)
+boolean addArrayName(String name)
+boolean isProgName(String name)
+boolean isFuncName(String name)
+boolean isProcName(String name)
+boolean isVarName(String name)
+boolean isArrayName(String name)

Figure 4: SymbolTable UML

5 Syntax Tree

The syntax tree package contains 14 classes that represent the different statements, variables, and operations in our Pascal. Each class (node) is responsible for adding it's part of the tree to the syntax tree which will be used by the code generation part of the compiler. The UML diagrams for the multiple classed can be seen in appendix C.

5.1 Testing

Testing the syntax tree was straightforward but tedious. I created a JUnit test and manually built a properly formated syntax tree for our bitcoins sample program. I then ran the complier against the sample code verifying that the two matched.

6 Code Generator

The final task of the compiler was of course to produce some MIPS assembly code. A new class, CodeGeneration, was used to write all the assembly code to a file named after the program. The code generator creates assembly code similar to the way the parser created a syntax tree.

6.1 Testing

To test the code generation I used QTSpim and manually stepped through the code seeing that the registers were being assigned the proper value.

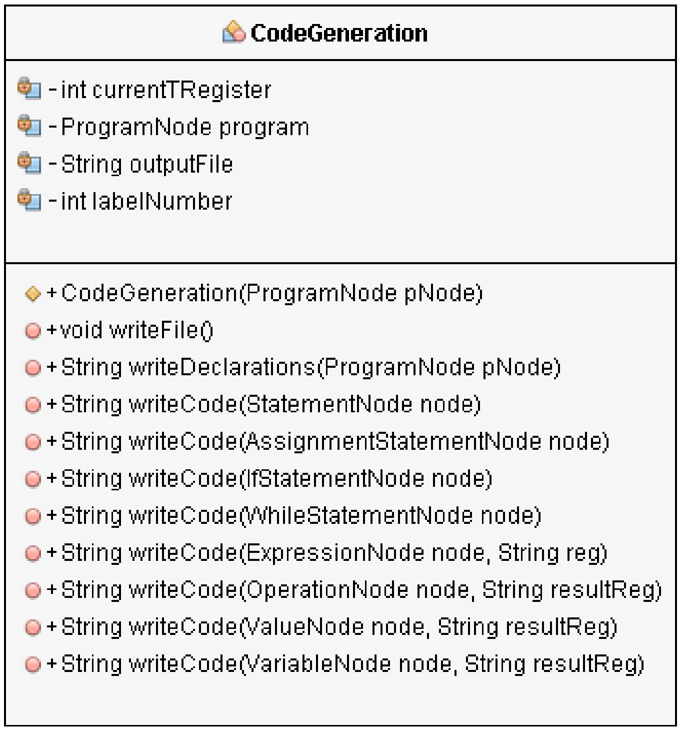

CodeGeneration
- int currentTRegister
- ProgramNode program
- String outputFile
- int labelNumber
+ CodeGeneration(ProgramNode pNode)
+void writeFile()
+ String writeDeclarations(ProgramNode pNode)
+ String writeCode(StatementNode node)
+ String writeCode(AssignmentStatementNode node)
+ String writeCode(IfStatementNode node)
+ String writeCode(WhileStatementNode node)
+ String writeCode(ExpressionNode node, String reg)
+ String writeCode(OperationNode node, String resultReg)
+ String writeCode(ValueNode node, String resultReg)
+ String writeCode(VariableNode node, String resultReg)

Figure 5: CodeGeneration UML

Appendix A List of Keywords

Symbols: .,−+∗/()[] <<=>>=::=;
Keywords: div, mod, and, program, id, var, array, of, num, integer, real, function, procedure, begin, end, if, then, else, while, do, not

Appendix B Grammar Rules

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 ->	**+** |
	-

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.

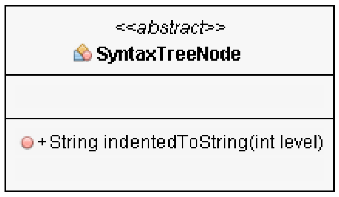
3. The relational operators (**relop**'s) are:
   =, **<>**, **<**, **<=**, **>=**, and **>**.

4. The **addop**'s are **+**, **-**, and **or**.

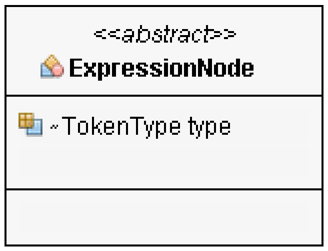
5. The **mulop**'s are *****, **/**, **div**, **mod**, and **and**.

6. The lexeme for token **assignop** is **:=**.

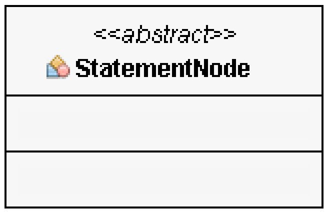
Appendix C Syntax Tree UML



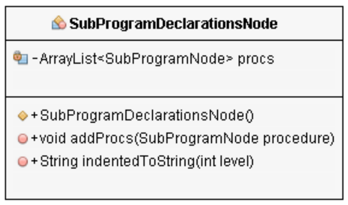
(a)



(b)

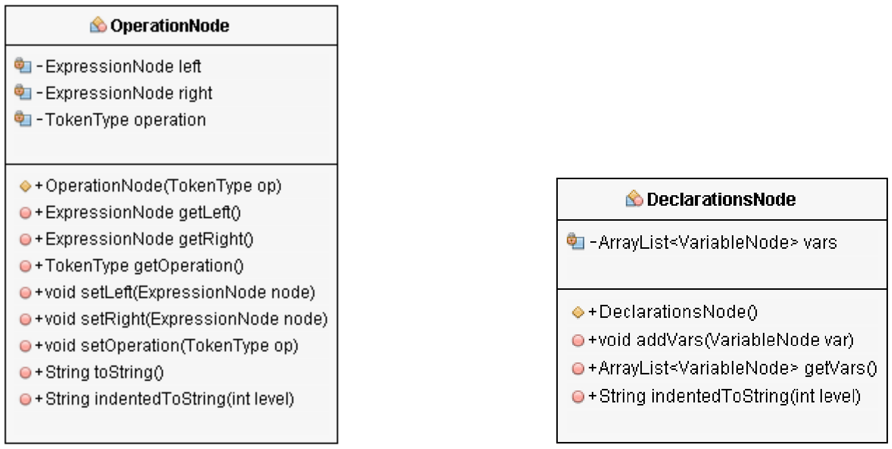


(a)


SubProgramDeclarationsNode
-ArrayList<SubProgramNode> procs
+SubProgramDeclarationsNode()
+void addProcs(SubProgramNode procedure)
+String indentedToString(int level)

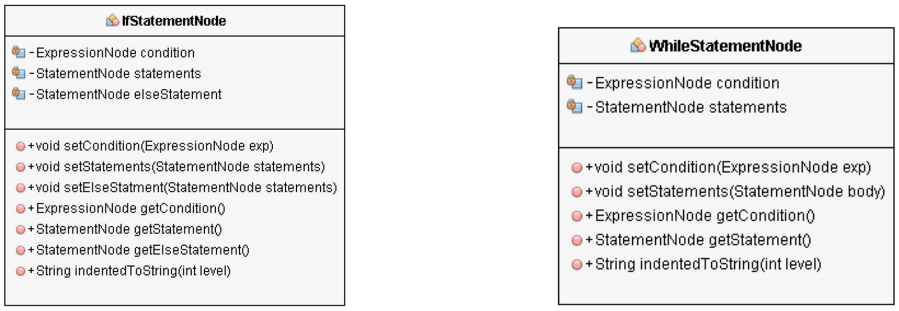
(b)

Figure 6


OperationNode
-ExpressionNode left
-ExpressionNode right
-TokenType operation
+OperationNode(TokenType op)
+ExpressionNode getLeft()
+ExpressionNode getRight()
+TokenType getOperation()
+void setLeft(ExpressionNode node)
+void setRight(ExpressionNode node)
+void setOperation(TokenType op)
+String toString()
+String indentedToString(int level)
DeclarationsNode
-ArrayList<VariableNode> vars
+DeclarationsNode()
+void addVars(VariableNode var)
+ArrayList<VariableNode> getVars()
+String indentedToString(int level)

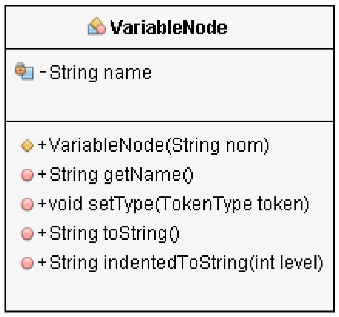
(a) (b)

Figure 7

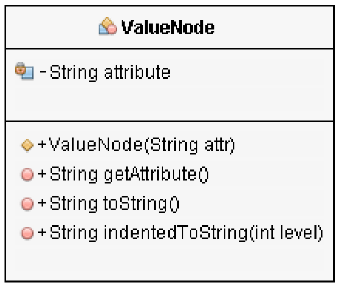

IfStatementNode
-ExpressionNode condition
-StatementNode statements
-StatementNode elseStatement
+void setCondition(ExpressionNode exp)
+void setStatements(StatementNode statements)
+void setElseStatment(StatementNode statements)
+ExpressionNode getCondition()
+StatementNode getStatement()
+StatementNode getElseStatement()
+String indentedToString(int level)
WhileStatementNode
-ExpressionNode condition
-StatementNode statements
+void setCondition(ExpressionNode exp)
+void setStatements(StatementNode body)
+ExpressionNode getCondition()
+StatementNode getStatement()
+String indentedToString(int level)

(a) (b)

Figure 8


VariableNode
- String name
+VariableNode(String nom)
+String getName()
+void setType(TokenType token)
+String toString()
+String indentedToString(int level)

(a)


ValueNode
- String attribute
+ValueNode(String attr)
+String getAttribute()
+String toString()
+String indentedToString(int level)

(b)

Figure 9