Compiler Software Design

Version 1.0

CSC 450

Alfonso Vazquez

Table Of Contents

1.0) Main Design

This Design Consist of different Sections explaining how the compiler from simple Pascal to MIPS assembly. The compiler only has the first two parts of the whole series implemented; both scanning a file and parsing a file are successful. This parse only deals with ASCII characters anything in UNICODE will not be scanned nor parsed.

2.0) Scanner Design

The Lexeme list was the first part of the design for this project. The lexeme list came out of the grammar (provided by Prof. Steinmetz).

Lexical Units

Q	Reserved Words	Other Units
1	if	;
2	then	.
3	else	:=
4	while	[
5	do	]
6	program	(
7	var	)
8	begin	+
9	end	-
10	array	*
11	of	/
12	function	=
13	procedure	<>
14	or	<
15	not	<=
16	mod	>=
17	and	>
18	real	:
19	integer	,
20		

The Scanner was designed from a Finite State Automata (Drawn with JFLAP). The Finite State Automata overtakes the whole simple Pascal language.
Starting with the 0 state. The 0 state initializes the process: From white space, to comments, to ID, Floating Values and Error state. All the 200's states are final. Instructions on nodes which have square brackets surrounding "[ ]" refers to as a push back the last input to stream (state 0). The Error state indicates to break the input stream, hence if the Automata does not read: Integer, symbol from the lexeme list, letter (ASCII Value 65-90 and 97-122). Or "{" then it will go to the error state, and terminate.

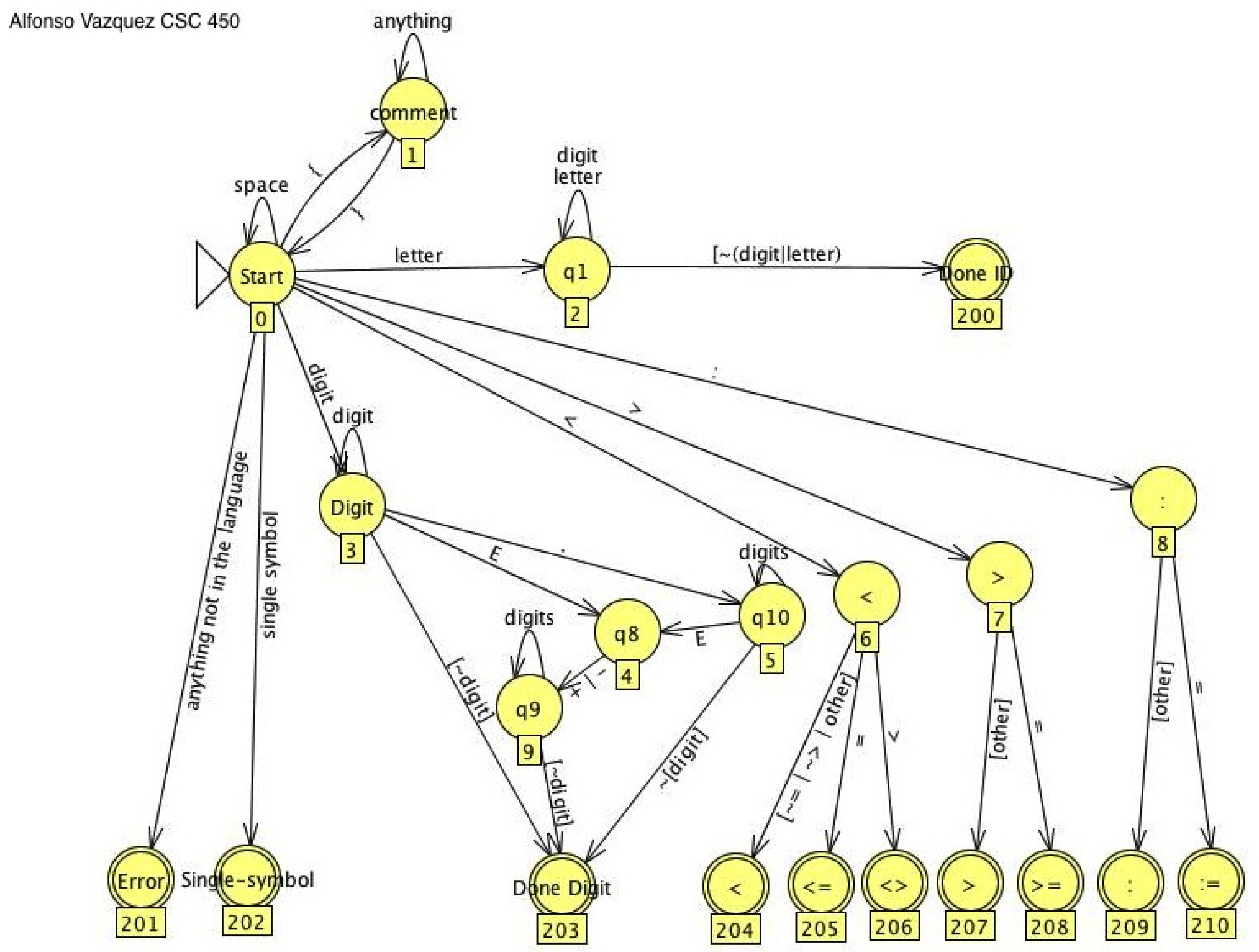


Figure (1).
**Boxes under the states represent the number of the state. This serves for coding purposes.

The scanner implementation:
Constructor takes care of the file read. Scanner has all the states declared as static final integers since they should not be changed unless the above diagram changes. The scanner can accept a token by IS_A_LEXEME, IS_NOT_A_LEXEME, or IS_AN_ERROR. The scanner has an implementation of cases for the input stream to go to. If a token reaches case 201 then the parser implementation is to keep going, since the scanner does not care if the token is accepted. The parser should take care of that.

3.0)The Parser Designed

The parser was implemented from the grammar, which essential is the pseudo code.
The implementation of the parser is to replace λ with an empty else statement returning nothing. As well to match all the current tokens that are been scanned in by the scanner. The parser is the responsible part of the code to error out if a lexeme is not available. The error is then showed, followed by the line number, which still needs to be implemented. The parser contains the calls of the syntax tree, and it is where the syntax tree is build.

3.1)The Grammar

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 :=.

4.0) Syntax Tree

This section consist of how the tree is build by the parser, the parser uses the nodes that are in the syntax tree package, and implements them by calling a Node of type Xnode, so for example, If a ExpressionNode needs to be called a Node is called and creates a ExpressionNode since all of the nodes are subsets of the Head Node.

The reason for the syntax tree is to be passed to the code generation class, along with the VariableSymbolTable class. The tree is for the code generator to know how the expressions are formed so if there is an expression a:= 3 + 4; and the VariableSymbolTable hashtable contains a type real, and it is variable 2, the code generation would know which registry to assign to a and how many add instructions it need to do.

The Syntax tree, is implemented in a 3 level UML tree.

Top Level:

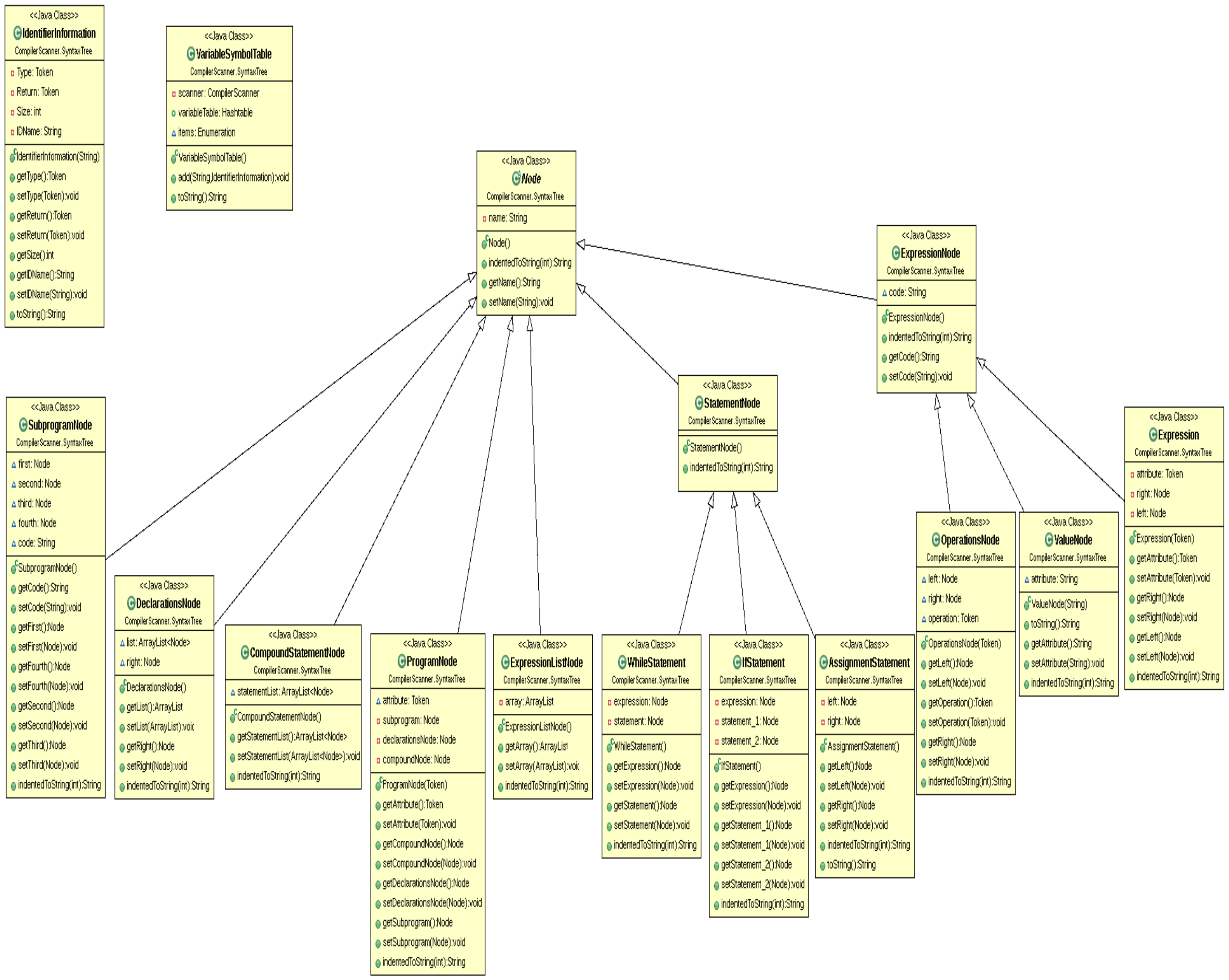
Node- is the top level, this level is here to be inherited from or extended by all the other lower levels of the tree.

Second Level:

DeclarationsNode- This class is for declarations it contain the indentedToString and sets right and left
CompoundStatementNode- This class is for compounds of the Grammar, it is an array list so it can contains many compound statements, and keep the tree balanced, rather than growing down and wasting heap size.
ProgramNode-This class is for the program to be contained, this is one of the first nodes called in the parser, it is of type Node, attributes storage, and compound and subProgram.
ExpressionListNode-This class is to create expressions, and print them out, it is kept in an array list, so basically it is the same idea as CompoundStatementNode, the tree does not grow ridiciously long by keeping all the expressions in a ArrayList.
StatementNode-This class is the a super class for the WhileStatement class, IfStatementClass, and AssignmentStatement class, all this does is keep a format for these sub classes.
ExpressionNode-This calss is a super class for OperationNode class, ValueNode class, Expression class.

Third Level:

WhileStatement-This is to keep the while pascal statemetns.
IfStatement-This is to keep the if pascal statements
AssignmentStatement- Sets the left and the right of an assigment.
OperationsNode- This class is to keep the Operations in order, for example +, –, /, and *
ValueNode- This class keeps the value of the ExpressionNode
Expression- This keeps the token, and attribute of the ExpressionNode.



<<Java Class>>
IdentifierInformation
CompilerScanner.SyntaxTree
Type: Token
Return: Token
Size: int
IDName: String
IdentifierInformation(String)
getType():Token
setType(Token):void
getReturn():Token
setReturn(Token):void
getSize():int
getIDName():String
setIDName(String):void
toString():String
<<Java Class>>
VariableSymbolTable
CompilerScanner.SyntaxTree
scanner: CompilerScanner
variableTable: Hashtable
items: Enumeration
VariableSymbolTable()
add(String,IdentifierInformation):void
toString():String
<<Java Class>>
Node
CompilerScanner.SyntaxTree
name: String
Node()
indentedToString(int):String
getName():String
setName(String):void
<<Java Class>>
ExpressionNode
CompilerScanner.SyntaxTree
code: String
ExpressionNode()
indentedToString(int):String
getCode():String
setCode(String):void
<<Java Class>>
StatementNode
CompilerScanner.SyntaxTree
StatementNode()
indentedToString(int):String
<<Java Class>>
Expression
CompilerScanner.SyntaxTree
attribute: Token
right: Node
left: Node
Expression(Token)
getAttribute():Token
setAttribute(Token):void
getRight():Node
setRight(Node):void
getLeft():Node
setLeft(Node):void
indentedToString(int):String
<<Java Class>>
SubprogramNode
CompilerScanner.SyntaxTree
first: Node
second: Node
third: Node
fourth: Node
code: String
SubprogramNode()
getCode():String
setCode(String):void
getFirst():Node
setFirst(Node):void
getFourth():Node
setFourth(Node):void
getSecond():Node
setSecond(Node):void
getThird():Node
setThird(Node):void
indentedToString(int):String
<<Java Class>>
DeclarationsNode
CompilerScanner.SyntaxTree
list: ArrayList<Node>
right: Node
DeclarationsNode()
getList():ArrayList
setList(ArrayList):voic
getRight():Node
setRight(Node):void
indentedToString(int):String
<<Java Class>>
CompoundStatementNode
CompilerScanner.SyntaxTree
statementList: ArrayList<Node>
CompoundStatementNode()
getStatementList():ArrayList<Node>
setStatementList(ArrayList<Node>):void
indentedToString(int):String
<<Java Class>>
ProgramNode
CompilerScanner.SyntaxTree
attribute: Token
subprogram: Node
declarationsNode: Node
compoundNode: Node
ProgramNode(Token)
getAttribute():Token
setAttribute(Token):void
getCompoundNode():Node
setCompoundNode(Node):void
getDeclarationsNode():Node
setDeclarationsNode(Node):void
getSubprogram():Node
setSubprogram(Node):void
indentedToString(int):String
<<Java Class>>
ExpressionListNode
CompilerScanner.SyntaxTree
array: ArrayList
ExpressionListNode()
getArray():ArrayList
setArray(ArrayList):voic
indentedToString(int):String
<<Java Class>>
WhileStatement
CompilerScanner.SyntaxTree
expression: Node
statement: Node
WhileStatement()
getExpression():Node
setExpression(Node):void
getStatement():Node
setStatement(Node):void
indentedToString(int):String
<<Java Class>>
IfStatement
CompilerScanner.SyntaxTree
expression: Node
statement_1: Node
statement_2: Node
IfStatement()
getExpression():Node
setExpression(Node):void
getStatement_1():Node
setStatement_1(Node):void
getStatement_2():Node
setStatement_2(Node):void
indentedToString(int):String
<<Java Class>>
AssignmentStatement
CompilerScanner.SyntaxTree
left: Node
right: Node
AssignmentStatement()
getLeft():Node
setLeft(Node):void
getRight():Node
setRight(Node):void
indentedToString(int):String
toString():String
<<Java Class>>
OperationsNode
CompilerScanner.SyntaxTree
left: Node
right: Node
operation: Token
OperationsNode(Token)
getLeft():Node
setLeft(Node):void
getOperation():Token
setOperation(Token):void
getRight():Node
setRight(Node):void
indentedToString(int):String
<<Java Class>>
ValueNode
CompilerScanner.SyntaxTree
attribute: String
ValueNode(String)
toString():String
getAttribute():String
setAttribute(String):void
indentedToString(int):String

5.0) Variable Symbol Table / Hashtable

The Hashtable is used to keep the variables from the the initial scanner code. This class does not extend the Hastable rather it has a Hashtable encapsulated in the class itself. It uses an Enumaration to print out as nice as possible from main.

6.0)Code Generation

The code generation takes in the tree as the input, it starts generating from the top to the lower levels. For now the code generation is not working properly. In case you input anything you will always have some bad code that you cannot run in qtSpim.

7.0) Limitation

The limitation of this compiler is: no arrays, and no functions. These two where not allowed to me to implement because of time limitation.

8.0)Profiling

Java profiling is possible in this compiler, I have a "devMode" implementations, this would be invoked at runtime with the extra argument '-d' this would allow the user to mount a profiler for 100,000 millisecond. This causes a thread to go to sleep in the environment. This only works as the compiler starts, and not as it goes on compiling. After the 100,000 millisecond are done, the compiler resumes normal mode.