Compiler Software Design

Version 1.0

CSC 450

Alfonso Vazquez

**Table Of Contents**

1.0 Main Designed …...............................................................................................................................1

2.0 Scanner …...........................................................................................................................................1

3.0 Parser …..............................................................................................................................................3

3.1 Grammar ….............................................................................................................................3

4.0 Syntax Tree ….....................................................................................................................................8

UML …..........................................................................................................................................9

5.0) Hashtable..........................................................................................................................................10

6.0) Code Generation …..........................................................................................................................10

7.0) Limitation …....................................................................................................................................10

8.0) Profiling ….......................................................................................................................................10

**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.



**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.