Erik Ayavaca-Tirado
4/28/19
CSC 451
SDD- Software Design Document
Compiler Project

**Table of Contents:**


**Overview:**

In this project a compiler will be created. This compiler will be taking in pascal code and compiling it, into MIPS assembly. Currently only the scanner and recognizer have been created. This is just the beginning part for the pascal compiler. This project will be divided in 6 main sections:
1. Scanner
2. Parser
3. Symbol Table
4. Syntax Tree
5. Semantic Analyzer
6. Code Generation

Each of which to a lot in terms in compiling a valid pascal program into a executable MIPS program.

**Design:**

This project will compile a pascal program into MIPS assembly. In this project there are four different parts, the 1st one being implemented is the scanner. The 2nd part being the parser but before the parser is created, we will create a recognizer class which determines if user input is a valid pascal program. After this we create a symbol table, which is used to store id's the were scanned into the parser. The final part is using the syntax tree package to create a tree , which is used for the code generation part.

**Scanner:**

A scanner was created based on the grammar for the obtained in class. Based on the grammar I was able to find 23 Keywords and 21 symbols, which are important for the pascal language. Below is the table for both the keywords and symbols important for the pascal language.

| # | Key Words |
|---|---|
| 1 | program |
| 2 | while |
| 3 | If |
| 4 | int(integer) |
| 5 | div |
| 6 | mod |
| 7 | or |
| 8 | and |
| 9 | do |
| 10 | else |
| 11 | then |
| 12 | function |
| 13 | var |
| 14 | read |
| 15 | begin |
| 16 | end |
| 17 | of |
| 18 | real |
| 19 | procedure |
| 20 | write |
| 21 | array |
| 22 | not |
| 23 | return |

Table 1: A table containing all keywords

| # | Symbols |
|---|---|
| 1 | + |
| 2 | - |
| 3 | = |
| 4 | > |
| 5 | >= |
| 6 | < |
| 7 | <= |
| 8 | * |
| 9 | / |
| 10 | ; |
| 11 | : |
| 12 | { |
| 13 | } |
| 14 | ( |
| 15 | ) |
| 16 | , |
| 17 | . |
| 18 | := |
| 19 | lamda |
| 20 | [0-9] |
| 21 | <> |

Table 2: Table containing all symbols

So, in order to understand where to start from with all this information. A state diagram was created to see how a scanner would process the keywords and symbols themselves.

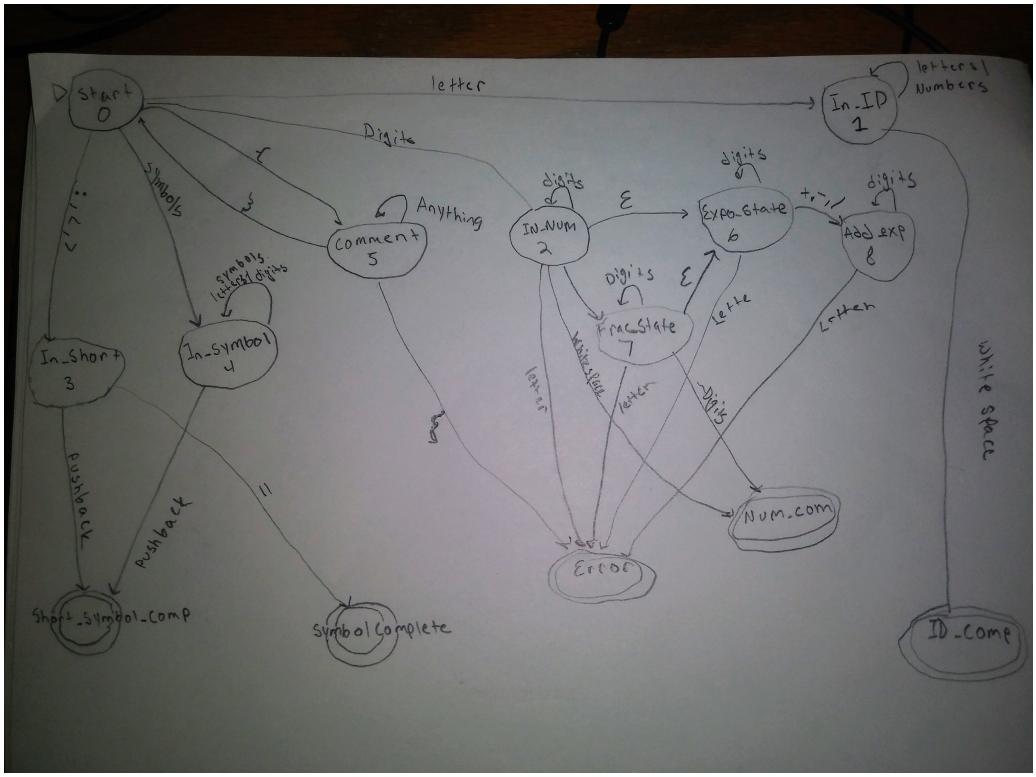

Image 1: State diagram used for coding references

Explanation:

So, we start of in state 0. The scanner reads in each character one by one to determine which state it needs to move on to. The cases are denoted below:

1. If a letter is read in, go to In_ID state
2. If a digit is read in, go to In_Num state
3. If "{" is read in , go to comment state
4. If a symbol is read in, go to In_Symbol state
5. If either: ; , <, or < is read in go to In_Short state

Note: Have to update image as this not the latest version of state diagram used to create the scanner.

The scanner was created using the jflex tool. There are 3 other specific classes in addition to the jflex file. The TokenType java file contains the list of keywords as well as the symbols found in the grammar obtained in class. The Token java file creates the token used in the scanner file. The lookup Table java file contains the list of all of the lexemes followed by their representing token type. The MyScannerTest.java is a j unit test is used to make sure that all pre-determined token's such as keywords and symbols are being scanned in properly as well as being able to capture numbers and ids of different forms.

**Parser:**

For this part, a java class named recognizer was created based on the grammar rules. These different rules are what define what a valid pascal program is to structure like. This is the first step before a parser class is created. Current there are 24 different grammar rules, which have been implemented as functions within the recognizer class.

**Production Rules**

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

Image 3: Image of grammar rules for mini pascal compiler part 1

*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* **)** |
**return** *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* -> **+** |
**-**

Image 4: grammar rules for mini pascal compiler part 2

After the creation of the recognizer program, we will transfer all the current code and call it a parser. A parser is a component that breaks data into smaller elements for easy translation into another language. Our parser takes input in the form of a sequence of tokens builds a syntax tree. So, this means the parser has calls to the syntax tree package

**Symbol_Table:**

The symbol table will store information about identifiers found within the pascal program. Each entry for an identifier in the Symbol Table will need to contain appropriate information about the identifier: its lexeme, the kind of identifier and any other information appropriate to the kind of identifier. The types of identifiers are program, variable, array, or function. Information is stored using a HashMap. It contains the following function:

1. addProgram()
2. addVariable()
3. addArray()
4. addFunction()
5. addProcedure()
6. isProgram()
7. isVariable()
8. isArray()
9. isFunction()
10. isProcedure()

for example, if were to do the following example:

```
@Test
public void test() {
    SymbolTable symbols = new SymbolTable();
    symbols.addProgram("RED");
    symbols.addArray("GREEN");
    symbols.addFunction("YELLOW", Type.REAL);
    symbols.addVariable("RUBY ", Type.INTEGER);
    System.out.println(symbols.toString());
}
```

Image 5: example of symbol table input

We would end up getting the following representation as our symbol table:

| NAME | KIND | TYPE |
|---|---|---|
| RED | PROGRAM | null |
| YELLOW | FUNCTION | REAL |
| GREEN | ARRAY | null |
| RUBY | VARIABLE | INTEGER |

Image 6: Example of symbol table output.

**Syntax Tree:**

This section consists of how the tree is built by the Parser. The Parser uses the nodes that are in the Syntax Tree Package and creates nodes. The reason for the implementation of the syntax tree in the parser is for the use of code generation and semantic analysis.

The tree is implemented in three levels.

Top Level:

**SyntaxTreeNode:** The top level of the tree.

Second Level:

**DeclarationsNode:** This class is for the declarations

**ProgramNode:** This class is for the program to be contained
**ExpressionNode:** This class is to create expressions and print them out.
**StatementNode:** This class is the superclass of CompoundStatementNode and AssignmentStatementNode. This keeps the formatting of the classes properly neat.
**SubProgramDeclarationsNode:** This class takes care of the sub nodes within the tree.

Third Level:
**ValueNode:** This class is keeps the values of the ExpressionNode.
**OperationsNode:** This class is to keep the Operations in order(PEMDAS), +. -. / , *
**AssignmentStatement:** Sets the left and right of an assignment.
**CompoundStatementNode:** This class is for the compounds of the grammar. It is in an array list so it can contain multiple statements and keep the tree balanced, rather than growing down and wasting time.

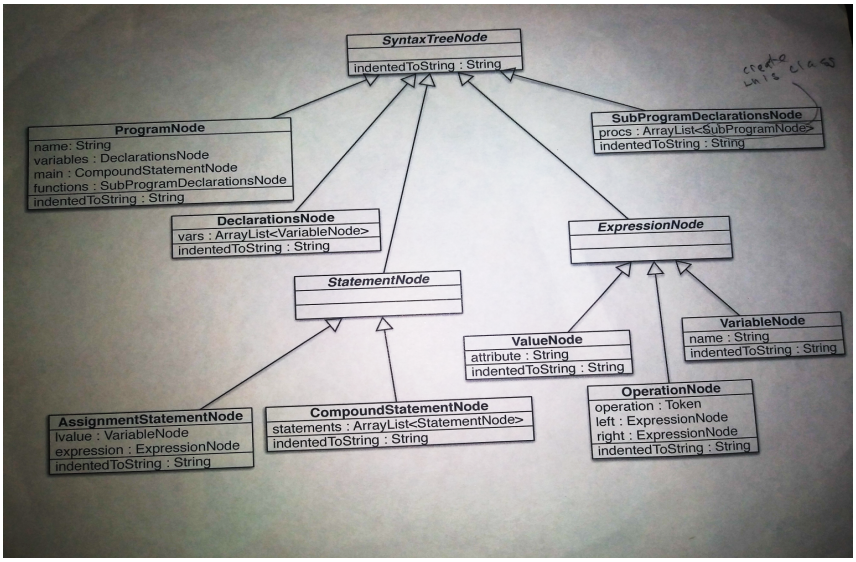

Image 5: Representation of the syntax package

For the following program of:

```
program sample;
var dollars, yen, bitcoins: integer;

begin
dollars := 1000000;
yen := dollars * 110;
bitcoins := dollars / 3900
end
.
```

Would produce the below output using the parser with the syntax tree implemented.

```
Program: sample
|-- Declarations
|-- --- Name: dollars
|-- --- Name: yen
|-- --- Name: bitcoins
|-- SubProgramDeclarations
|-- Compound Statement
|-- --- Assignment
|-- --- --- Name: dollars
|-- --- --- Value: 1000000
|-- --- Assignment
|-- --- --- Name: yen
|-- --- --- Operation: ASTERISK
|-- --- --- --- Name: dollars
|-- --- --- --- Value: 110
|-- --- Assignment
|-- --- --- Name: bitcoins
|-- --- --- Operation: SLASH
|-- --- --- --- Name: dollars
|-- --- --- --- Value: 3900
```

**Semantic Analyzer:**

For the semantic analyzer section, we are to make sure that the following conditions are met:

- Make sure all variables are declared before they are used.
- Assign a datatype of integer or real, to each expression (declare a type field in ExpressionNode, for example). ---
- The type of any expression node in the tree should be printed in the tree's indentedToString.Make sure that types match across assignment.

**Code Generation:**

In this section we are going to create a code generation module which takes the syntax tree as its input. The output returns a string with the MIPS assembly language code as its output. For example, if we had the expression: 3 * (4 + 7), the generated MIPS assembly code would look like the below text:

```
.data
answer:   .word   0
.text
main:
addi   $t0,   $zero, 3
addi   $t2,   $zero, 4
addi   $t3,   $zero, 7
add    $t1,   $t2,   $t3
mult   $t0,   $t1
mflo   $s0
sw     $s0,   answer
addi   $v0,   10
syscall
```

**Limitations:**

Due to how certain elements were implemented and or coded, this compiler does have its limitations. Below is a list of a few of these limitations

- Does not have code generation working properly.
- Semantic analyzer is not working properly.
- Can only recognize valid pascal programs and parse the simplest pascal program.
- Parser is not creating nodes properly.

**Version history:**
**5/1/19 – added in chapter of compiler limitations**
**4/28/19 – added information to Code Generation chapter**
**4/25/19 – Symbol Table edited, Added the syntax tree and semantic analyzer chapters**
**2/24/19 – Symbol Table chapter added**
**2/15/19 – Parser chapter added**
**1/27/19 – improved scanner**
**12/16/18 – Original Scanner**