Erik Ayavaca-Tirado
2/15/19
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 4 main sections:
1. Scanner
2. Parser
3. Symboltable
4. Syntax tree

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.

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.

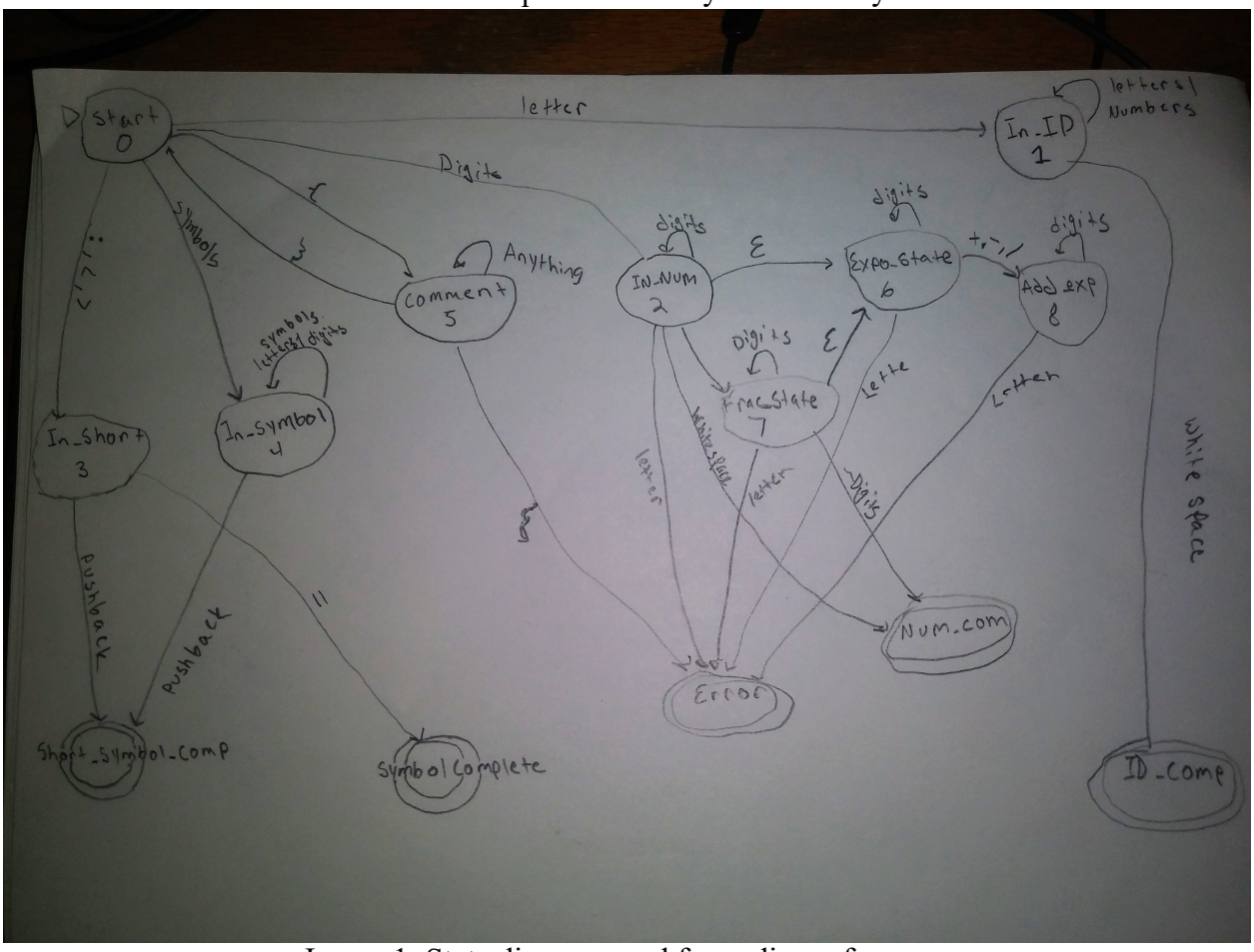

Image 1: State diagram used for coding references

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 lookUpTable 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 id's 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 ;** *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* *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* **)** |
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 -> **+** |
-

Version history:
2/15/19 – Parser chapter
1/27/19 – improved scanner
12/16/18 – Original Scanner