AllenCompiler

Marissa Allen

Overview:

The AllenCompiler will read the input characters from a plain text file or a String, scan the file using the rules defined in the jflex generated Scanner class of the java code, and output a sequence of tokens for the parser. Every token will consist of the lexeme, or characters in input stream, and the token type. The scanner will distinguish which characters are keywords and which are unexpected tokens. The scanner and its classes are stored in the scanner package.

The recognizer and its classes are stored in the parser package. It will look at the stream of tokens from the scanner, in a particular order, and interpret the token types. It will then compare the current lookahead token to the token type. If it matches all of the expected token types listed in the method and any other methods called; then the method will execute the rule for the non-terminal symbol in the micro pascal grammar.

The symbol table and its classes are stored in the parser package and it interacts with the recognizer and parser in order to help differentiate between the different program, variable, function, or procedure identifiers that are stored in the symbol table. It does this by adding the current lexeme to the symbol table as one of the different identifiers, depending on what kind of identifier that method in the grammar is.

The CompilerMain is the driver class that connects different classes together. When the class reads in a pascal file, the program method in the Parser class is called through its constructor. A symbol table is constructed when information from the file is

stored for each symbol name. A formatted symbol table and syntax tree are then written to separate files as long as the file that was input exists and is a pascal file.

The parser constructs a syntax tree for the input sequence of tokens from the scanner. The parser checks to see if the string of token names can be generated by the micro pascal grammar, and constructs a syntax tree if it is a pascal program. A syntax tree is then built for the program method in the middle of the recursive decent.

The syntax tree is used to represent a program in this tree structure. The syntax tree is built by using the non-terminal symbols in the micro pascal grammar in the Parser class and nodes from the classes in the syntaxtree package. When the pascal program is parsed in the parser class, nodes from the syntax tree classes are created and added to the syntax tree.

CompilerMain:

This class connects multiple classes together, namely the Parser and SymbolTable classes, in order to generate an output file containing a formatted symbol table when a pascal file is passed in. It also connects together the Parser and the syntaxtree classes to generate an output file containing a syntax tree of a pascal program from the input file. First, the file length is checked to see if anything was entered or not. If no file is entered, then it asks the user to enter a file name, and then exits the file. If the user did pass in a file, the file is read from the args arguments, and passed into the program method in the Parser class through its constructor. A symbol table is constructed as information is stored about each symbol name in the pascal file.

As long as the file the user entered was a correct pascal program file, a symbol table and a syntax tree are generated from the pascal program from the toString and indentedToString methods and written to a file.

<u>Scanner:</u>

The scanner program is going to scan through the input file and return a lexeme. The lexeme is going to be the string containing the actual characters that were read in that make up one particular token in the language. The token type is the individual token type descriptor of the keywords, identifiers, and symbols. The keywords and symbols are listed after the design section, anything else is an identifier.

Scanners have a state machine that does pattern matching, a diagram of the state machine can be seen below. As long as the scanner sees characters that are not spaces, newline characters, or tabs, it will keep adding them to the lexeme. If the scanner does see those, and it already has a lexeme, it's going to push any spaces, newline characters, or tabs back into the input stream and return the lexeme because it's done. If the scanner sees a space, newline characters, or tab, and it doesn't have a lexeme yet it keeps going to find a lexeme and throws the whitespace character away.

This program contains four java files – Token, TokenType, ScannerTest, and Scanner. It also contains one jflex file called AllenScanner. The AllenScanner file contains the instructions on how each Token should be handled. The Token class is a constructor class for a token object. All the tokens will consist of the lexeme, or characters in the input stream, and the token type. The TokenType class consists of all

of the different types a token can be. These types are then passed into the scanner

class. There is a type for every keyword and symbol as well as two types for ID and

number. ScannerTest takes an input file or a string and feeds the characters into the

scanner class. As long as the scanner doesn't hit a null token, the scanner prints out

any tokens that are not whitespace tokens or null tokens. The Scanner file is the java

file that was created when the AllenScanner file was run. The scanner for this class

uses the keywords and symbols defined for the token and prints out the tokens that are

not whitespace tokens or null tokens along with any errors.

<u>Recognizer:</u>

The beginning of the parser is created by building a recognizer implemented as a

top-down recursive descent parser. The recursive descent parser will act as a

recognizer, answering the yes-no question "is the input a pascal program?". A

recognizer is a parser that doesn't perform any syntax-directed translation. The

recognizer will only say whether the input string is from the language described by the

grammar or not. In all of the non-terminal methods listed in the grammar, it will do this

by looking at a stream of tokens from the scanner in a particular order, interpreting the

token types, and comparing the current lookahead token to the token type. If the current

lookahead token matches all of the expected token types listed in the method and any

other methods called; then the method will execute the rule for the non-terminal symbol

in the micro pascal grammar. The recognizer works with the symbol table in order to

help differentiate between the different program, variable, function, or procedure identifiers that are stored in the symbol table.

This program contains two classes – Recognizer and RecognizerTest. The Recognizer class checks to see if the input passed into its constructor is a pascal program in the micro pascal grammar. If it is, then the rule for the method is executed. The RecognizerTest class is a JUnit testing class that tests the methods from the Recognizer class by testing to see if the current token matches the expected token type in that method.

Parser:

The parser will look at a stream of input tokens from the scanner and treat the token names as terminal symbols of a context-free grammar. The parser will then construct a syntax tree for that input sequence of tokens. The parser builds a syntax tree in the middle of the recursive decent for the program method. It will use the first components of the tokens produced by the scanner to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation. The parser also works with the symbol table in order to help differentiate between the different program, variable, function, or procedure identifiers that are stored in the symbol table.

This program contains two java files – Parser and ParserTest. The Parser class creates a parser that is implemented as a top-down recursive descent parser. The

parser builds a syntax tree using the non-terminal symbols from the grammar and the nodes from the classes in the syntaxtree package. Every time a node returns, it returns a part, and that part is added to the syntax tree. The ParserTest class produces an indentedToString version of the syntax tree. It does this by manually building a syntax tree and passing a pascal program into the non-terminal methods in the Parser class and the nodes from the classes in the syntaxtree package.

Syntax Tree:

A syntax tree is a tree structure where each interior node represents an operation and the children of the node represent the arguments of the operation. The syntax tree will be used to represent a program in this tree structure. A syntax tree is a variant of a parse tree. The reason why we use a syntax tree instead of a parse tree is because a parse tree contains more details than are actually needed, which makes it difficult for a compiler to parse a parse tree. By using a syntax tree, those extra details can be eliminated. The syntax tree is built by using the non-terminal symbols in the micro pascal grammar in the Parser class and nodes from the classes in the syntaxtree package. When the pascal program is parsed, nodes from the syntax tree classes are created and added to the syntax tree.

Symbol Table:

The symbol table is used to store information about each identifier (symbol name) found in a pascal program. The symbol table is created by implementing a hash

map to store an entry into the symbol table by adding a key-value pair. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its datatype, and its kind. The symbol table interacts with the recognizer and parser in order to help differentiate between the different program, variable, function, or procedure identifiers that are stored in the symbol table. It does this by adding the current lexeme to the symbol table as one of the different identifiers, depending on what kind of identifier it is. For example, in the program method, it has a program identifier so the current lexeme would be added to the symbol table as a program name.

This program contains three java files – KindEnum, SymbolTable, and SymbolTableTest. The KindEnum file contains the different kinds for every symbol table identifier. The SymbolTable file contains methods that allow a symbol to be added to the symbol table; along with methods checking to see if it is either a program, variable, array, function, or procedure identifier. The SymbolTableTest file uses JUnit tests to see whether the methods from the SymbolTable file are able to add a symbol to the symbol table and verify the kind of symbol it is.

Change Log:

02/15/2019	Marissa Allen	Added the Recognizer section of the compiler.
03/9/2019	Marissa Allen	Added the Symbol Table section of the compiler.

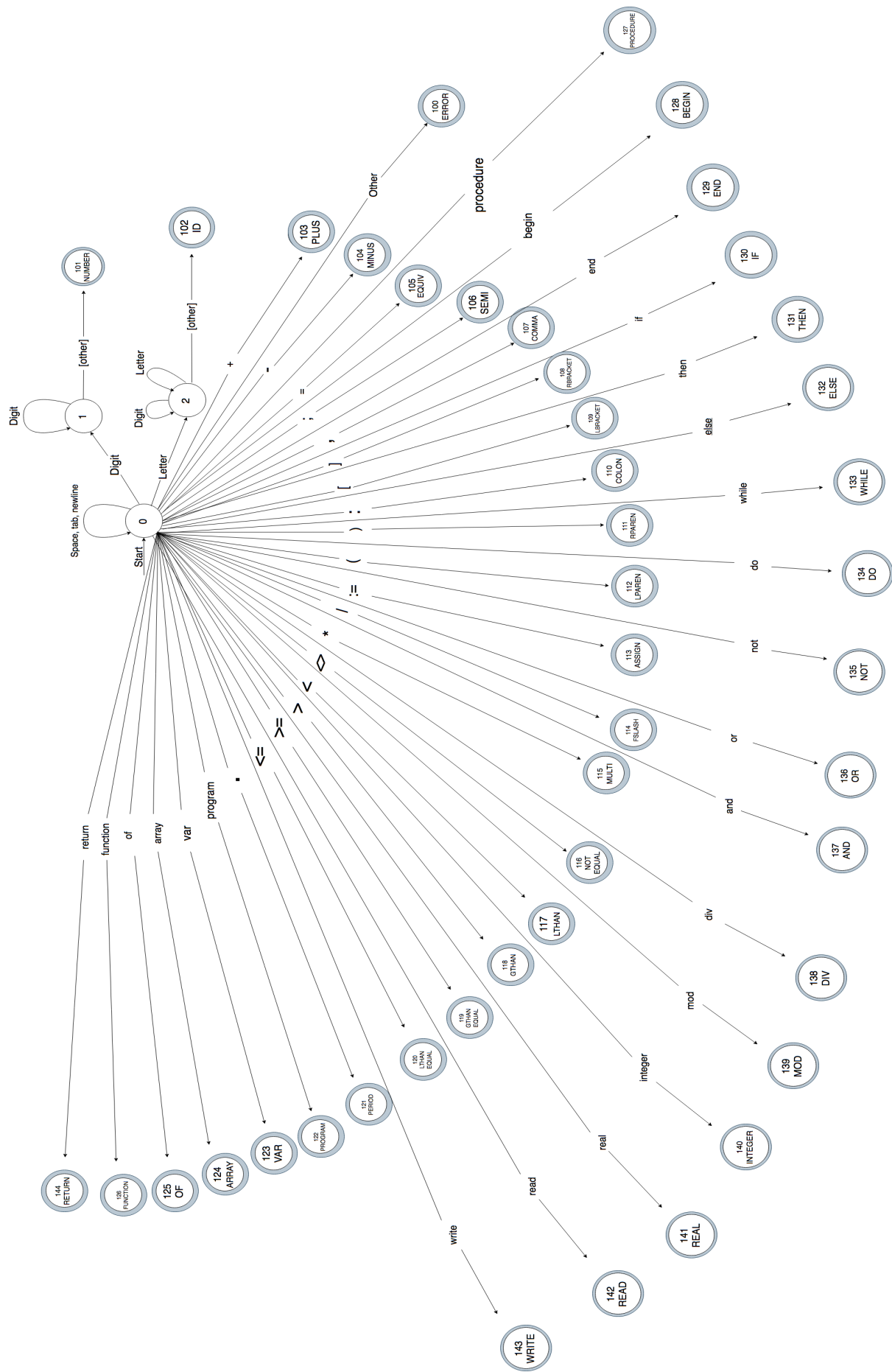| 03/14/2019	Marissa Allen	Added the CompilerMain section of the compiler.
03/17/2019	Marissa Allen	Added the Parser and Syntax Tree section of the compiler

List of Keywords/Reserved Words:
program
var
array
of
function
procedure
begin
end
if
then
else
while
do
not
or
and
div
mod
integer
real
read
write
return

List of Symbols:
;
,

```
[
]
:
)
(
+
-
*
=
<>
<
<=
>=
>
/
:=
.
```

State transition diagram (lexical analyzer DFA)

States:
Start → 0

- 0 —Digit→ 1 (1 —Digit→ 1 loop)
- 1 —[other]→ 101 NUMBER
- 0 —Letter→ 2 (2 —Letter/Digit→ 2 loop)
- 2 —[other]→ 102 ID
- 0 —Space, tab, newline→ 0 (loop)

Transitions from 0:

- + → 103 PLUS
- - → 104 MINUS
- = → 105 EQUIV
- ; → 106 SEMI
- , → 107 COMMA
-] → 108 RBRACKET
- [→ 109 LBRACKET
- : → 110 COLON
-) → 111 RPAREN
- (→ 112 LPAREN
- !! → 113 ASSIGN
- / → 114 RSLASH
- * → 115 MULTI
- ◊ → 116 NOT EQUAL
- < → 117 LTHAN
- > → 118 GTHAN
- >= → 119 GTHAN EQUAL
- <= → 120 LTHAN EQUAL
- . → 121 PERIOD
- program → 122 PROGRAM
- var → 123 VAR
- array → 124 ARRAY
- of → 125 OF
- function → 126 FUNCTION
- return → 144 RETURN
- Other → 100 ERROR
- procedure → 127 PROCEDURE
- begin → 128 BEGIN
- end → 129 END
- if → 130 IF
- then → 131 THEN
- else → 132 ELSE
- while → 133 WHILE
- do → 134 DO
- not → 135 NOT
- or → 136 OR
- and → 137 AND
- div → 138 DIV
- mod → 139 MOD
- integer → 140 INTEGER
- real → 141 REAL
- read → 142 READ
- write → 143 WRITE

FSM Design: Designed by Marissa Allen and Cohl Dorsey