2019

DorseyCompiler

A COMPILER MADE BY COHL DORSEY

COHL DORSEY

Overview

This compiler is built to compile a very minimalistic version of pascal into MIPS assembly. The compiler will take in a file containing the code for this mini version of pascal where it will then scan the file using the rules defined in the scanner class of the java code. It will then be parsed by the parser class and compiled into a functioning MIPS assembly program. This assembly program will then be able to be run and function as defined by what was written in the pascal code.

The Scanner

This scanner is programmed in java and is generated using jflex. The scanner will take in a file and scan it, once the file has been scanned it will print out to the terminal a list of accepted tokens and unexpected tokens. The code for the Scanner and all its needed classes are stored in the dorseyScanner package.

JFlex File

The jflex file holds the tags to generate the java scanner file. It also contains the grammar and definitions for how each grammar object should be treated. When compiled this creates the the Scanner.java file that can then be compiled.

Scanner

This Scanner is the java file created from the jflex file. The scanner will take in a file and scan it for the tokens that have been defined. To see what these tokens are you can find the list of them in the key words and symbols section in this document. Once the file has finished being scanned the scanner will output the tokens that were defined and it will also show the ones that are errors.

TokenType

This class contains all the types of objects that a token can be. When the class is called in the driver it has a list of types that it iterates through. The correct token type is then taken and passed into the scanner.

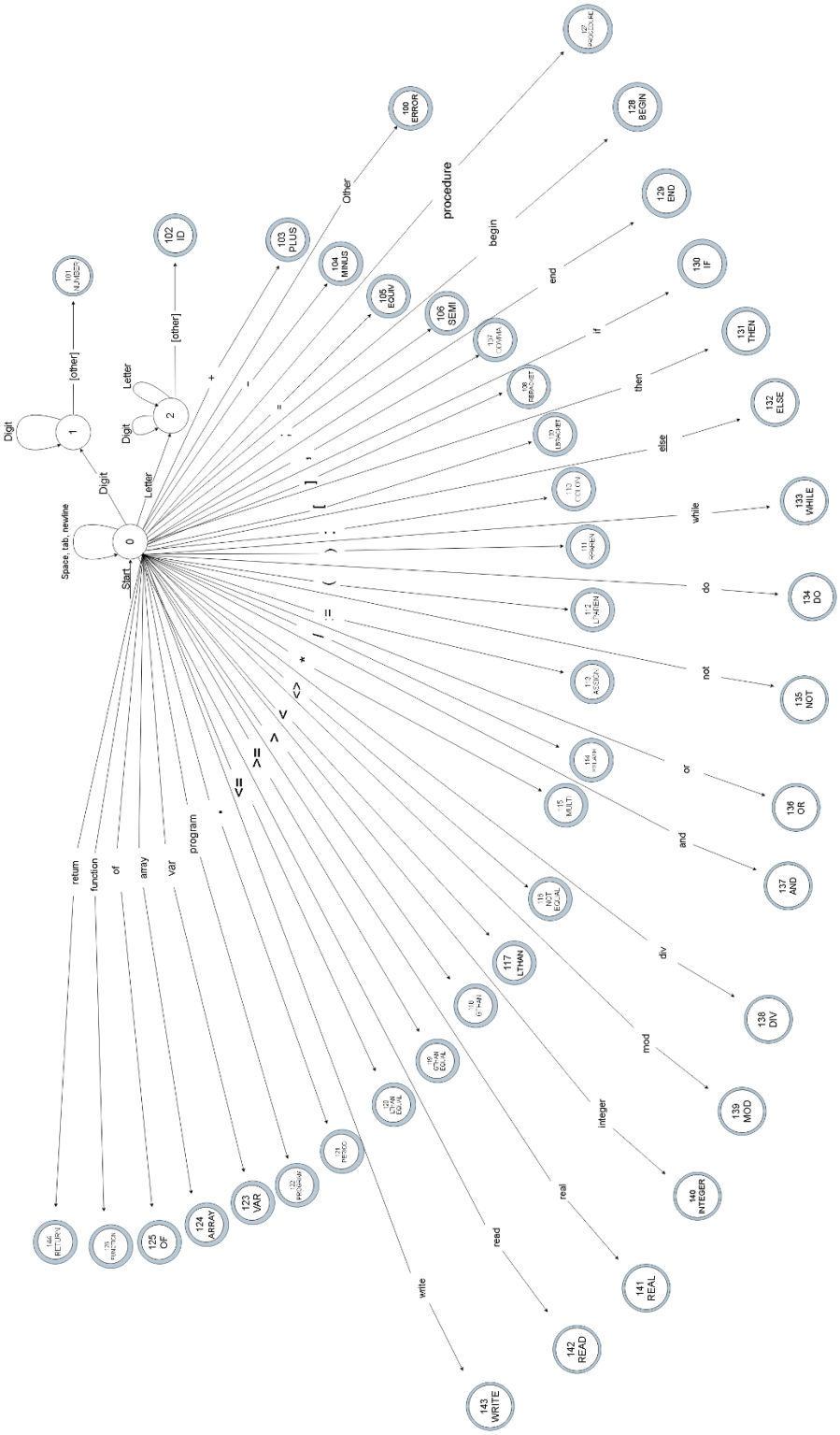
Token

This class is the constructor for a Token object. This class creates a Token that once its type is determined gets used by the scanner to parse the file that was given as input.

MyScanner

This is the main class of the scanner. It takes in a file to be scanned, creates a scanner object. The file is then passed into that scanner where it generates the tokens and processes them according to the definitions that are defined by its type.

The DFS



FSM Design: Designed by Marissa Allen and Cohl Dorsey

List of Keywords/Reserved Words:	**List of Symbols:**
array	;
program	,
var	[
do	]
if	)
else	(
then	{
of	}
function	:
procedure	+
begin	-
end	*
while	=
and	<>
or	<
not	<=
div	>=
mod	>
integer	/
real	:=
	.

The Parser

The Parser is written in all Java code and has test files written using Junit. The Parser is responsible for using the scanner to parse through the pascal code and break the code up into the appropriate symbolic tokens. The parser package consists of several class files that work in conjunction such as the actual Parser class, and SymbolTable class which creates a table of the symbols. The parser package contains several methods that help construct the

Parser

The Parser uses the scanner package to parse through the pascal code and tokenize it into its specified tokens. It then adds any new symbols into the token table.

SymbolTable

The SymbolTable class is the class used to represent the symbol table that stores the tokens. It includes methods to see if a token already exists in the table, it also has methods to determine what type of token is being stored into it based on the grammar that we specified in the Recognizer.

Recognizer

The Recognizer class contains our actual pascal grammar. When this class goes over the code it turns the code into the appropriate token based on what is specified in our grammar.

CompilerMain

This class represents the main aspect of the compiler. When the compiler is fully complete this is the class that will manage the compiler as a whole.

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. In the future this class will work with the CodeGeneration more directly by helping to create functions and other assembly instructions.

Semantic Analyzer

The semantic analyzer uses the syntax tree and the information in the symbol table to check that the declarations and statements of a program are semantically correct for the micro pascal grammar. An important part of semantic analysis is type checking, where the compiler finishes the analysis begun by the parser and confirms that a type is allowed to store other types across assignment. This semantic analyzer allows types to store other types across assignment, as long as a variable is not of type integer and its assignment is of type real. A variable of type integer is not able to hold a real, but a variable of type real is able to hold an integer type. It also checks that a type isn't being assigned to a variable declared to be a different type. Unless of course, the declared variable is of type real, because a real is able to hold an integer value. The semantic analyzer also gathers the type information of an ExpressionNode and adds it to the syntax tree. The information added to the syntax tree will be used during intermediate code generation. Another job of the semantic analyzer is to check and see if variables have been declared before they are used. If a variable has not been declared the code generator will not generate code.

Code Generation

The code generator is the final phase in the compiler model that occurs after the semantic analysis is done. In this final phase, the code generator converts an intermediate representation of source code into machine code that can be executed by a machine. The code generator takes in a syntax tree and a symbol table as its input and returns a string of MIPS assembly language code as its output. The syntax tree is used by the code generator because it is a tree-like intermediate representation that depicts the grammatical structure of the token stream. There are three primary tasks important to the code generator: instruction selection, register allocation and assignment, and instruction ordering. For the instruction selection, appropriate target-machine instructions are chosen to implement the intermediate representation statements. These statements are converted into machine code that can be executed by a machine. The register allocation and assignment section has a certain number of values that are maintained during the execution. It decides what values to keep in what registers. For the instruction ordering, the order that the instructions will be executed in is decided by the code generator. The code generation class uses the information added to the syntax tree from the semantic analyzer. With this new addition to the syntax tree, the code generator has the last bit of information needed to generate assembly code. The code generator outlines the blueprint of the assembly code and uses the syntax tree nodes to fill in the missing pieces of assembly code for the declarations and statements. When a method for the node is executed, the tree structure of the syntax tree is broken down into a sequence of instructions and code is written for the given node. A String of the assembly code is returned when the method is finished executing and added to the assembly code outline. Once all of the declarations and statements have been filled in, the assembly code for the pascal program is

complete. The String of assembly code is then returned to the CompilerMain class to be written to a file.

Change Log

02/15/2019	Added the Recognizer section of the compiler.
03/9/2019	Added the Symbol Table section of the compiler.
03/14/2019	Added the CompilerMain section of the compiler.
5/1/2019	Added the Sections for the Semantic Analyzer and the Code Generation.