MINI-PASCAL COMPILER

SOFTWARE DESIGN DOCUMENT

Beteab Gebru

February 20 2019

Table of contents

Change Log

Date	***Comment***
01/20/19	*Created document*
02/28/19	Added Overview
03/01/19	Added scanner section
03/05/19	Added section on parser design
03/06/19	Added section on symbol tables
04/02/19	Added section on Syntax Tree

1. Overview

This program when complete will be able to compile code written in pascal language into MIPS assembly code. It will be built up in section increments.

The 5 components listed below will make up the final program. Each component will be Junit Tested at every iteration.

- Scanner
- Parser
- Symbol Table + Syntax Tree
- Semantic Analyzer
- Code Generation

Design

SCANNER

pre-defined pascal language IDs and symbols are listed in the grammar document. Below are the symbols and IDs set for detection by our pascal scanner. They will form the tokens.

KEYWORDS: *AND, DIV, MOD, NOT, NUMBER, ARRAY, BEGIN, DO, ELSE, END, FUNCTION, IF, INTEGER, OF, OR, PROCEDURE, PROGRAM, REAL, THEN, VAR, WHILE*

SYMBOLS , , :=, *, /, +, -, >, <, >=, <=, =, <>, |, (,), {, }, [,],

The package scanner has the following files which will help us comb through a given pascal code and extract matching tokens in order of appearance in text.

Scanner (Scanner.Jflex): the DFA scanner is made using JFlex from the *myScannerfromJFLEX.java* which details the lexical rules as well as the predefined symbols and IDs. The scanner will identify the tokens.

Token: The class token defines what a token object will look like.
The class defines *lexeme* and *type* as properties of any given token. There is also a *toString()* function that will output detected Tokens in a format -> `"Token: \"" + this.lexeme + "\" of type: " + this.type;`.

Token Type: this class will define list of ENUMs as specifications for the *token.type* attribute.

Lookup Table: this class will extend the HashMap<> from collections. We will store all our symbols and their lexemes for lookup during Token detection. We will have ease of access for compare decisions.

Parser

The package is called parser, but it contains 3 files (Symbol Table, Parser, Syntax Tree). The Symbol table contains all tokens found by the scanner and stores them with their associated types for other classes to reference.

Parser will have valid tokens passed as parameter (String text) and then seeks to validate the pascal code against one of the predefined production rules. We will seek to create syntax tree for the pascal code that is deemed valid.

In the next section. There will be syntax tree of the symbols, implemented as series of nodes.

Symbol Type

We define ENUMS to serve as a specification file the symbol table. These types are extracted from the grammar specification document, which defines the various pascal symbols, namely variables, arrays, functions, programs, and procedures.

Symbol Table

The symbol table will serve to store all the symbols we gather and validate from our parser. We will have the identifiers stored as pairs with their types. It will be used as reference table by other components such as parser.

Parser

We have defined rules for production of code in the parser class. We will take in valid tokens and verify against the production rules we have defined. We do this by comparing the tokenType against expected possibilities after the last token. The parser will throw an error if it finds the token to be against expected TokenType (expected token). Parser will return error location (line and col number) and reason. When valid Symbol such as a variable is found it will store it in the symbol table. When there is ambiguity in a statement it will use the symbol table to resolve between the possibilities.

Syntax Tree

The parser will be used to create a parse tree to enable the recognition of valid pascal expressions. This parse trees will be used to validate the pascal code.

We have series of classes defining different types of pascal expressions in the sytaxtree package. These definitions will be used to create a tree bade up of nodes of tokens/expressions that make up a program. The tree will help our compiler decide the sequence of execution on the program. It serves as an intermediate representation of the program, and has decisive impact on the final output of the compiler.

The declared classes of expressions listed below have required functions to assist in the construction of the syntax tree. Each class represents a node in the tree and relates to parent and child nodes in order of how they out to be executed.

List of all Classes/Nodes

Assign, Assignment Statement, Compound Statement, Declarations, Function, If Statement, Operation, Procedure, Program, Read, Sign, Statement, Sub Program Declarations, Sub Program Head, Sub Program, Syntax Tree, Value, Variable, While Statement,

Semantic Analysis

During the sematic analysis our program will check to ensure the sematic rules are obeyed. We check if variables or expressions IDs are declared before being used. We take a top-level program node and iterate over the tree to check on datatype of the variable and following uses are adhering to the declared type.

We have the following functions to help us accomplish this.

• analyze(): Will be called by main to get the sematic analysis going. It will get statements for which we will use the verify and assign expression types to the statements.

• verifyVariableDeclarations(): Verifies that all variables used were declared before they were used.

• assignTypeToExp(): Will take compound statement as parameter and identify expression type. Will look at the instance to decide what type it may be. Once assigned a type the ID is ready to be used as declared.

• setExpTypes(): sets the type of expression node.

• setVarVal(): sets a variable or value to be a certain type (real or integer)

• getLNode(): returns the left child of an expression node

• getRNode(): returns the right child of an expression node

Code Generation

The ultimate outcome of this is to turn a validated pascal code into MIPS assembly language code. We first perform semantic analysis on the code, and we then go on to produce string of MIPS code. We