TMouaCompiler Project

By Timly Moua

Overview

In this software project will take a subset of the C language and implement a translator for it through a series of programming packages that build up each of the compiler's major functional units. We will be using a previous and enhance scanner created from JFlex tool to help create a scanner to capture and return certain keywords and symbols as described in the grammar.

Design

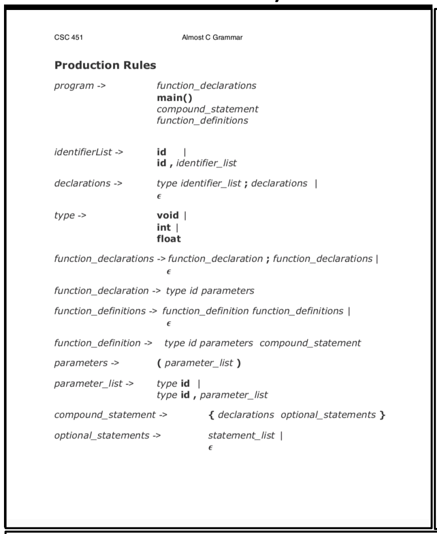
The project consists of different packages explaining how the compiler takes C language into MIPS assembly. The Compiler will have six main packages for implementation, a completion of a Scanner, Parser, Syntax Tree, Semantic and Code Generation.

SCANNER PACKAGE

The design of the Scanner is divided into one JFlex file and five java classes. In the JFlex file, it contains all the patterns, declarations and lexical rules. The JFlex file gets parsed into a Jflex tool which is lexical analyzer generator that takes inputs and specification with a set of regular expressions and corresponding actions. It generates a program that reads input, matches the input against the regular expressions in the spec file, and runs the corresponding action if a regular expression matched. A Token class is created defining the token object since each token has a lexeme that contains a string with actual contents and types. For TokenType class it is generated to set the patterns of the keywords in its each distinguish type. A lookuptable is generated by using the HashMap function to hold all token types for a lookup. At the end a main class is created for being able to read in any text file to get parsed for distinguishing the token variables.

RECOGNIZER

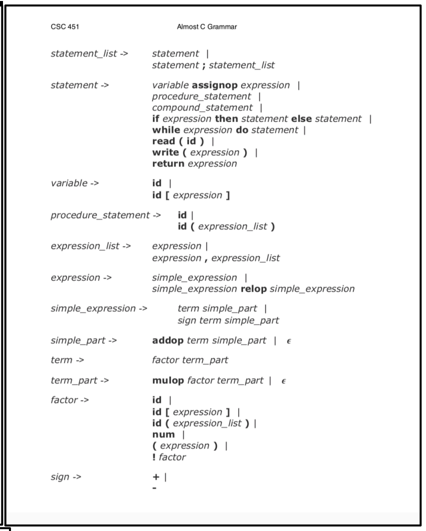
The Recognizer class will be based on these grammar rules to create a micro c program structure to start the beginning of a Parser. Once the rules are correctly formatted it be transferred onto the actual parser where it will take inputs and builds a Syntax Tree which will be created eventually.



CSC 451 Almost C Grammar

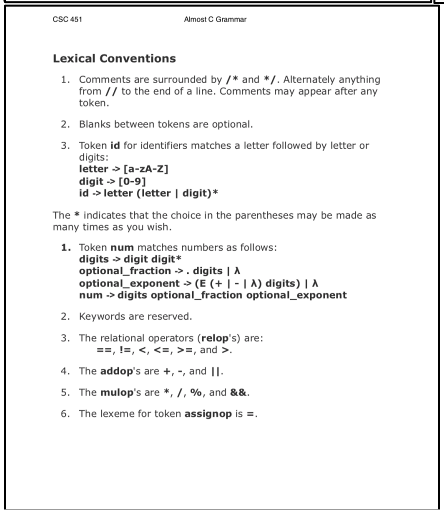
Production Rules

```
program ->                 function_declarations
                           main()
                           compound_statement
                           function_definitions

identifierList ->          id     |
                           id , identifier_list

declarations ->            type identifier_list ; declarations  |
                           ϵ

type ->                    void |
                           int |
                           float

function_declarations -> function_declaration ; function_declarations |
                            ϵ

function_declaration ->  type id parameters

function_definitions -> function_definition function_definitions |
                            ϵ

function_definition ->    type id parameters  compound_statement

parameters ->              ( parameter_list )

parameter_list ->          type id  |
                           type id , parameter_list

compound_statement ->               { declarations  optional_statements }

optional_statements ->              statement_list  |
                                    ϵ
```



CSC 451 Almost C Grammar

```
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  |   ϵ

term ->                    factor term_part

term_part ->               mulop factor term_part  |   ϵ

factor ->                  id  |
                           id [ expression ]  |
                           id ( expression_list ) |
                           num  |
                           ( expression )  |
                           ! factor

sign ->                    + |
                           -
```



CSC 451 Almost C Grammar

Lexical Conventions

1. Comments are surrounded by **/*** and ***/**. Alternately anything from **//** to the end of a line. Comments may appear after any token.
2. Blanks between tokens are optional.
3. Token **id** for identifiers matches a letter followed by letter or digits:
   letter -> [a-zA-Z]
   digit -> [0-9]
   id -> letter (letter | digit)*

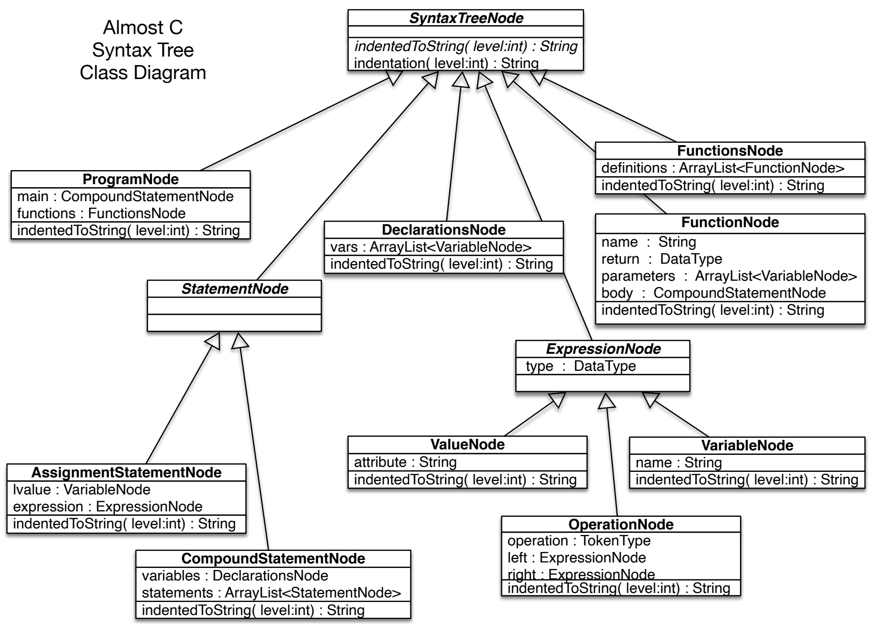
The * indicates that the choice in the parentheses may be made as many times as you wish.

1. Token **num** matches numbers as follows:
   digits -> digit digit*
   optional_fraction -> . digits | λ
   optional_exponent -> (E (+ | - | λ) digits) | λ
   num -> digits optional_fraction optional_exponent
2. Keywords are reserved.
3. The relational operators (**relop**'s) are:
   ==, **!=**, **<**, **<=**, **>=**, and **>**.
4. The **addop**'s are **+**, **-**, and **||**.
5. The **mulop**'s are *, **/**, **%**, and **&&**.
6. The lexeme for token **assignop** is **=**.

Symbol Table

The design of the symbol table is too store information about the identifiers founded in a C program. The Symbol Table would hold information about the identifier in the format of finding its lexeme first, the kind of identifier (PROGRAM, VARIABLE, ARRAY, or FUNCTION) and other information such as its type (INT, VOID, FLOAT). The Symbol table will be able to add symbols when declarations are being made. And is able to read in arguments and write its own symbol table into a file.

Syntax Tree:


Almost C
Syntax Tree
Class Diagram
SyntaxTreeNode
indentedToString(level:int) : String
indentation(level:int) : String
ProgramNode
main : CompoundStatementNode
functions : FunctionsNode
indentedToString(level:int) : String
FunctionsNode
definitions : ArrayList<FunctionNode>
indentedToString(level:int) : String
FunctionNode
name : String
return : DataType
parameters : ArrayList<VariableNode>
body : CompoundStatementNode
indentedToString(level:int) : String
DeclarationsNode
vars : ArrayList<VariableNode>
indentedToString(level:int) : String
StatementNode
ExpressionNode
type : DataType
AssignmentStatementNode
lvalue : VariableNode
expression : ExpressionNode
indentedToString(level:int) : String
CompoundStatementNode
variables : DeclarationsNode
statements : ArrayList<StatementNode>
indentedToString(level:int) : String
ValueNode
attribute : String
indentedToString(level:int) : String
VariableNode
name : String
indentedToString(level:int) : String
OperationNode
operation : TokenType
left : ExpressionNode
right : ExpressionNode
indentedToString(level:int) : String

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.

Semantic Analysis:
In the Semantic Analysis, it is supposed to have a function to check that variables in expressions have been declared. Also have a function to assign the datatype to every expression node and a Datatype displayed in indentedToString. Function to check type-matching across assignment.

Code Generation:
The final package in the Compiler is the Code generation. This package handles MIPS Assembly code. The purpose of Code Gen is too take the Syntax Tree as a input, and return a String with the MIPS assembly language code as its output. In the Code Gen, when variables are declared, their memory address will be added onto the symbol table. The Variables are then written in the .data section and local variables and function/arguments are pushed to stack which will have their offset from the stack pointer as their memory address.

Change Log:
01/26/2020 – Added Overview, Design, Scanner implementation
02/23/2020 – Added Recognizer and Explanation of Grammar
03/01/2020 – Added Partial Design of the Symbol Table
03/08/2020 – Added Inputs of Integration of the Symbol Table

03/25/2020 – Added Syntax Tree Design
04/29/2020 – Added Semantic and CodeGeneration