Pascal Compiler

Ken Johnson

December 18, 2014

Contents

1 Introduction

This document is an overview of the design process and implementation of a Pascal Compiler written in Java. The compiler takes an input Pascal file, and compiles to MIPS assembly language. The following sections contain a brief ovew of the entire process of the compilation, followed by an in depth explanation of each component is given in the chapters following the overview. The explanations of design decisions are somewhat minimal, as the majority of the decisions for the first two components of the compiler were primarily a direct result of instruction, and the specifications given. As a note, there are a few things in the grammar that are not fully implemented in the compiler: the or and and operations are not functional as well and scientific notation. Functions, procedures, and arrays were all also left out of the functionality of the compiler.

2 Overview

To compile the Pascal code there are a variety of components the compiler must consist of; these will be discussed in short here.

2.1 Scanner

The scanner is the first phase of the compilation process. The scanner's purpose is to scan through the input file, and analyze its contents character by character to build up tokens. The tokens are the accepted keywords of the language, IDs, numbers, etc. A full list of these tokens can be found in the Scanner chapter, and in Appendix A in the full grammar specification of the language.

2.2 Recognizer

The next phase of the compiler is the recognizer. In a the final product this will be a full parser instead of just a recognizer. The recognizer uses the grammar found in Appendix A to analyze the tokens found using the scanner. This analysis determines whether the Pascal code will compile or not according to the grammar. If the code is correct and will compile, the compiler will print out a success message and exit with code 0. If there is an error it will state the line on which the error occurred and exit with a code corresponding to the error. These codes can be found in Appendix B.

2.3 Parser

The Recognizer has been exteded to a full parser. The purpose of the parser to perform all the same functions as the recognizer, but it now also creates a symbol table and a syntax tree. The implementation of this is a recursive descent parser.

2.3.1 Symbol Table

The symbol table hold all declared variables, functions, and procedures, as well as their types.

2.3.2 Syntax Tree

The syntax tree is simply an internal representation of the pascal code. It consists of nodes, starting from the root Program node.

2.4 Code Generator

The Code Generator is the piece of the compiler that will translate the internal representation of the code to assembly. It is given a Syntax Tree, and it generates non-optimized MIPS Assembly code.

3 Scanner

The scanner will be discussed in more depth here. The input is handled by a PushbackReader in order to push unwanted characters back in to the input stream. The scanner consists of this PushbackReader, a TransitionTable, an enum of tokens, and a SymbolTable. This TransitionTable is the corresponding table for the DFA found in Appendix C. Character by character the table appends to a working string (built with a StringBuilder), according to the current state and the next state. For example: Given the line

$$var\ x :=$$

the scanner would step through starting with v, seeing that it is a letter, it moves into the ID state. The scanner then appends the v to working string which was previously λ. It will then repeat the same process with a and r. Once the reader reaches the space it will push this back, and accept the token. Next it checks to see if *var* is in the SymbolTable. The SymbolTable has a value for *var* that is equal to Token.VAR. Since the symbol was recognized, the token value is now Token.VAR. The space is then passed over, moving to the x. The x is appended to the λ string and the space is pushed back. The string is checked against the SymbolTable, and nothing is found so the token returned it Token.ID. Next the *:* is brought in by the parser. This has its own state as *:* is a valid symbol, as is *:=*. The scanner now appends the *:* to the λ string and moves to the next state. Since the next character is = is appends this and moves to the symbol acceptance state. There is no need to push anything back here, as nothing else appended to this is a valid token, and the check for the validity of the token ordering is done later in the parser/recognizer. This is also queried against the SymbolTable and is assigned the value Token.ASSIGNOP. Granted this is not a valid line of Pascal code, but according to the scanner, it works.

4 Recognizer

The recognizer, which will later be a full-fledged parser, is the second layer of the compiler. It uses the tokens from the scanner to parse the syntax of the Pascal file. This is done by almost directly implementing the grammar specified in Appendix A. Each of the non terminals in the grammar corresponds to the same method in the recognizer. For example take the first rule in the grammar:

program → ***program id ;***
declarations
subprogram_declarations
compound_statement
.

This is of course the first thing that must be in a Pascal file. The first token must be program, followed by an id and a semicolon. This is checked in the *program* method in the recognizer. As these are bold in the grammar, they represent tokens. The following non-bold rules are other non-terminals. Because of this the *program* method calls the *declarations* method, which will in turn match tokens *var* and then call *identifier_list*, which in turn calls other methods. After moving back up the stack of calls from *declarations* program calls *subprogram_declarations* and repeats the process before moving on to *compound_statement*. The program then matches the . or the PERIOD token. This is the parsing of the entire file, as everything is contained between the *program* and the PERIOD and parsed further down the call stack. The recognizer checks to make sure that there are no issues in parsing and then prints a success message. It does not yet assign any meaning to the syntax, this will be handled by the actual parser. If there is an error found while parsing the file, it shows the line it encountered the error, and what type of error occurred. A full list of these errors and error codes can be found in Appendix B.

5 Parser

The extension of the recognizer discussed in the previous section, works as a recursive descent parser. As the lexical tokens were previously recognized, they are now parsed and assigned meaning. This means that the parser starts at Program, then recursively goes over the remainig parts of the program. During the declarations part of the parsing, symbols are added to the Symbol Table. The information added here is what kind of declaration it is, var, array, procedure, or function, the type of value or return value, and its lexical token. It also handles the various scopes used by different methods. This is used to ensure variables have been declared. No instantiation is required to use a variable, but values default to zero. As of this moment, functions, procedures and arrays are recognized, and to some extent put into the Symbol Table. For the variables, both integer and real, the Symbol Table is also used for type checking. In addition to the Symbol Table, the Parser also builds a Syntax Tree. The Syntax Tree is an internal representation of the Pascal code. The tree is built by creating a node at each of the methods corresponding to the non-terminals of the grammar. The functions of these two components are not discussed in their own section as they are simple structures used by the Parser to place data in. Once this internal representation of the code is generated by the Parser, this tree is handed off to the Code Generator.

6 Code Generator

Once a Syntax Tree has been created and fully filled, it is given to the Code Generator. The Code Generator walks this tree, starting with the program, then evaluating the children: the declarations nodes, the main, and main's statements. As this tree is walked assembly code is generated and appended to a string, which is written to the a file. The MIPS code that is generated is the version of MIPS that is used with the qtSPIM application.

7 Appendices

7.1 Appendix A: Grammar

This is the grammar for the Pascal language as specified for this version of the compiler.

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
subprogram_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)

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 -> **+** |
-

Lexical Conventions

1. Comments are surrounded by **{** and **}**. They may not contain a {. 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 **or**.

5. The **mulop**'s are ***, **/**, **div**, **mod**, and **and**.

6. The lexeme for token **assignop** is **:=**.