

Software Design Document

Michael Farghali

February 9, 2016

Introduction

This is a preliminary Software Design Document (SSD) for my mini-Pascal compiler that I will be working on over the next two semesters. When completed it should accept a text file which represents a mini pascal language. The text file should then be able to be converted to assembly language if its syntax is correct according to our production rules.

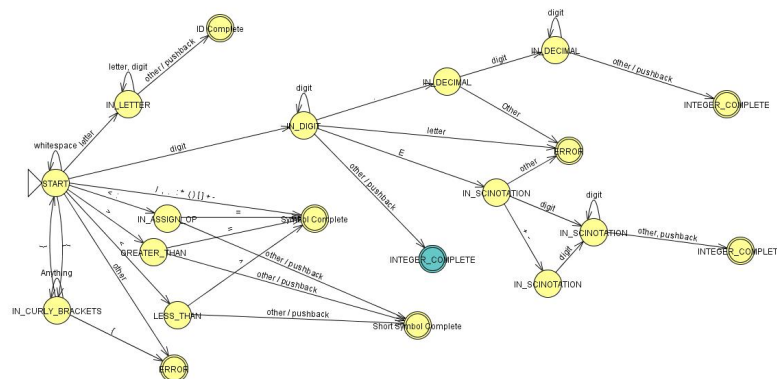
Scanner

The Scanner class should read in a file and process it character by character. It is based on a Deterministic Finite Automata, Appendix A, and the given grammars. The Scanner reads in a file and attempts to match each string to a given keyword, symbol, or number. It returns a valid Token if the string is valid in the language or it will return false. See Appendix B for list of valid Tokens.

Parser

The Parser class processes a text file token by token which are given by the Scanner class. It uses the grammar rules listed in Appendix C to match the tokens against expected tokens. It will eventually create a parse tree but for now it only checks that the production rules are followed.

Appendix A Deterministic Finite Automata



Appendix B List of Keywords

Symbols: $\cdot, - + */()[] <<=>>=:$

Keywords: div, mod, and, program, id, var, array, of, num, integer, real, function, procedure, begin, end, if, then, else, while, do, not

Appendix C Grammar Rules

Production Rules

<i>program</i> ->	program id ; <i>declarations</i> <i>subprogram_declarations</i> <i>compound_statement</i> .
<i>identifier_list</i> ->	id id , identifier_list
<i>declarations</i> ->	var identifier_list : type ; declarations λ
<i>type</i> ->	<i>standard_type</i> array [num : num] of standard_type
<i>standard_type</i> ->	integer real
<i>subprogram_declarations</i> ->	<i>subprogram_declaration ;</i> <i>subprogram_declarations</i> λ
<i>subprogram_declaration</i> ->	<i>subprogram_head</i> <i>declarations</i> <i>subprogram_declarations</i> <i>compound_statement</i>
<i>subprogram_head</i> ->	function id arguments : standard_type ; procedure id arguments ;
<i>arguments</i> ->	(parameter_list) λ
<i>parameter_list</i> ->	<i>identifier_list : type</i> <i>identifier_list : type ; parameter_list</i>
<i>compound_statement</i> ->	begin optional_statements end
<i>optional_statements</i> ->	<i>statement_list</i> 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*

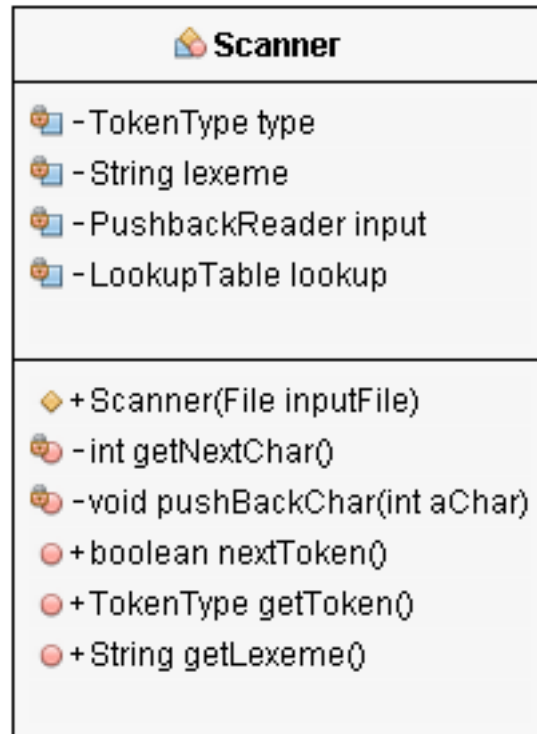
sign -> **+** |
 -

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 **:=.**





Appendix D UML Diagrams

Parser

-  - Scanner scanner
-  - TokenType currentToken

-  +Parser(String filename)
-  +void match(TokenType expectedToken)
-  +void program()
-  +void identifier_list()
-  +void declarations()
-  +void type()
-  +void standard_type()
-  +void subprogram_declarations()
-  +void subprogram_declaration()
-  +void subprogram_head()
-  +void arguments()
-  +void parameter_list()
-  +void compound_statement()
-  +void optional_statements()
-  +void statement_list()
-  +void statement()
-  +void variable()
-  +void expression()
-  +void simple_expression()
-  +void simple_part()
-  +void term()
-  +void term_part()
-  +void factor()
-  +void expression_list()
-  +void sign()
-  +void mulop()
-  +void error()

 SymbolTable
 ~Hashtable table
<ul style="list-style-type: none"> + boolean addProgramName(String name) + boolean addFunctionName(String name) + boolean addProcName(String name) + boolean addVarName(String name) + boolean addArrayName(String name) + boolean isProgName(String name) + boolean isFuncName(String name) + boolean isProcName(String name) + boolean isVarName(String name) + boolean isArrayName(String name)