Documentation for Delivery 2

CSCI 465 – Fall 2015

Marshall Mattingly III

12 November 2015

Table of Contents

[1. Scanner 3](#_Toc435082562)

[1.1. Assumptions 3](#_Toc435082563)

[1.1.1. Debug information shall be outputted to stdout and stderr 3](#_Toc435082564)

[1.1.2. Initially, only a subset of Pascal shall be parsed 3](#_Toc435082565)

[1.1.3. Flex / Bison shall be used as counter-points to the hand-made scanner 3](#_Toc435082566)

[1.2. Current Status 3](#_Toc435082567)

[1.2.1. Feature-complete with the previous requirements 3](#_Toc435082568)

[1.2.2. Feature-complete with the current requirements 3](#_Toc435082569)

[1.2.3. Functions, Constants supported 3](#_Toc435082570)

[1.2.4. Multi-line and inline-comments 4](#_Toc435082571)

[1.2.5. Minor warnings and error suggestions for the scanner 4](#_Toc435082572)

[1.3. Future Work 4](#_Toc435082573)

[1.3.1. Include semantic analysis 4](#_Toc435082574)

[1.3.2. Executable code generation 4](#_Toc435082575)

[1.3.3. Completion of Flex implementation 4](#_Toc435082576)

[1.3.4. Add more Pascal functionality 4](#_Toc435082577)

[1.3.5. Enhanced error messages 4](#_Toc435082578)

[1.4. Design 4](#_Toc435082579)

[1.4.1. FSM for the scanner 4](#_Toc435082580)

[1.4.2. Format for scanner 5](#_Toc435082581)

[2. IO module 7](#_Toc435082582)

[2.1. Assumptions 7](#_Toc435082583)

[2.1.1. Standard C library FILE handling 7](#_Toc435082584)

[2.2. Current Status 7](#_Toc435082585)

[2.2.1. Read / Write / Unget 7](#_Toc435082586)

[2.3. Future Work 7](#_Toc435082587)

[2.3.1. Read line-by-line 7](#_Toc435082588)

[3. Symbol Table 8](#_Toc435082589)

[3.1. Assumptions 8](#_Toc435082590)

[3.1.1. Initialized by the scanner 8](#_Toc435082591)

[3.2. Current Status 8](#_Toc435082592)

[3.2.1. Basic functionality 8](#_Toc435082593)

[3.2.2. Scopes 8](#_Toc435082594)

[3.3. Future Work 8](#_Toc435082595)

[3.3.1. Hashtable 8](#_Toc435082596)

# Scanner

The scanner is responsible for the lexical analysis in the program. It takes input from the IO module, character by character, and transforms the input into lexemes (or tokens) for use by the parser.

## Assumptions

This section details any assumptions made about the scanner.

### Debug information shall be outputted to stdout and stderr

Any debugging information will be outputted to stdout (for general information) and stderr (for warnings). If the user would like to store this output, it must be redirected into a file.

### Initially, only a subset of Pascal shall be parsed

Given constraints, only the subset of Pascal defined in our documentation will be provided. However, it is the intention of this author to create a fully-working Pascal compiler.

### Flex / Bison shall be used as counter-points to the hand-made scanner

Flex and Bison, being automated generators for lexical analysis and parsing, respectively, will be used to attempt to verify the hand-written scanner and parser. This shall work as a nice study into the limitations of Flex / Bison vs hand-written compilers.

## Current Status

This section details the current status of the scanner.

### Feature-complete with the previous requirements

The scanner can create lexemes for each feature in the requirements.

### Feature-complete with the current requirements

The parser is recursive in nature with one (1) lookahead value, or LL(1). This means that the semantic analysis will have to be tightly coupled with the parser in the near future to allow for the confirmation of correctness.

#### Full symbol table support

The symbol table is an important element in the compiler, and the parser extends upon the work created by the scanner. The parser is now able to add entries into the symbol table, with context, verify that there are no conflicting symbols, and create multiple levels of symbol tables for different blocks within the program, e.g. functions, procedures, etc.

#### Excellent error messages and printouts

The parser currently prints out a line whenever it enters a new recursive function for verbosity, but this will be toned down on release. Error messages are presented, in full, wherever an error occurs; however, there is no current error correction and all errors halt the program. It is intended to make the error handling more robust in the future, as it was with the scanner.

### Functions, Constants supported

Along with the basics, both functions and constants are supported by the symbol table and parser, as can be seen in the example.

### Multi-line and inline-comments

Multi-line and inline comments are accepted with the following form: (\* … \*) and { … }. All comments are simply passed over by the scanner, as if they were whitespace.

### Minor warnings and error suggestions for the scanner

The scanner attempts to be somewhat intelligent with its warnings and errors, provided a line number and suggested issue when encountering an error – i.e. if an identifier starts with a number, the scanner will respond with incorrect number or identifier (since it isn’t entirely sure which has occurred). Warnings are things that may be unintended behavior, such as providing an empty string, since the scanner cannot differentiate between a string and a character.

## Future Work

This section details any future work that needs to be completed on the scanner.

### Include semantic analysis

Since the compiler uses a recursive approach without generating a full parse tree, the semantic analysis must be tightly coupled with the parser.

### Executable code generation

Executable code will be generated.

### Completion of Flex implementation

Currently, the Flex implementation is missing key parts, such as string literal recognition. It must be extended to serve as a true counter-point.

### Add more Pascal functionality

The scanner currently parses a large subset of the Pascal language. Adding tokens and logic to the scanner is relatively trivial, and shall be added in the future. However, adding parsing logic to the parser afterward may be more troublesome.

### Enhanced error messages

While the current error messages provide some detail on suggested fixes, enhancing the scanner to provide even more robust error messages would be ideal.

## Design

This section has all the design elements.

### FSM for the scanner

This is the FSM for the scanner, showing how the scanner produces the tokens.

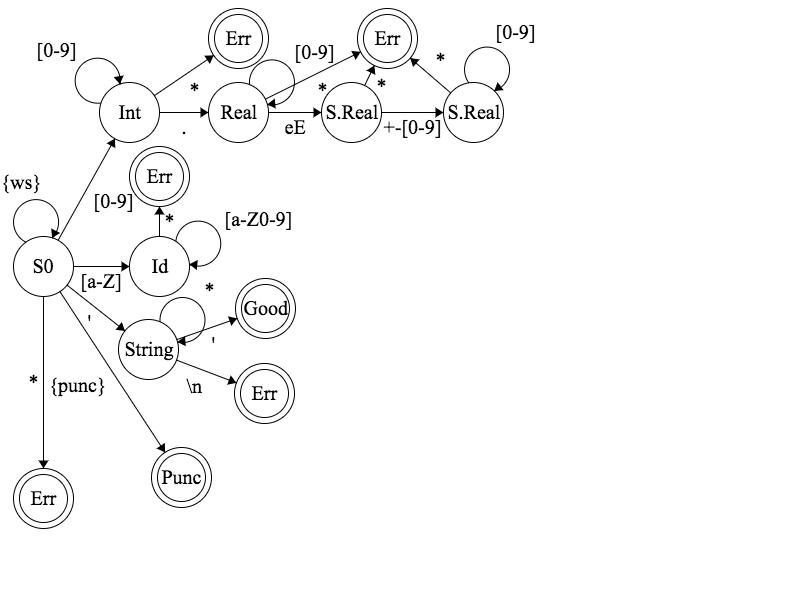


Figure 1: FSM for scanner

### Format for scanner

This section has the format for the scanner, showing how the scanner works.

start : program

program : PROGRAM ID SEMICOLON block PERIOD

block : constant-definition-part

/\*type-definition-part\*/

variable-declaration-part

procedure-and-function-declaration-part

statement-part

constant-definition-part : CONST constant-definition | ε

constant-definition : ID constant-definition-variable EQUALS constant-no-id SEMICOLON constant-definition-recursive

constant-definition-variable : COMMA ID constant-definition-variable | ε

constant-definition-recursive : constant-definition | ε

type-definition-part : TYPE type-definition | ε

type-definition : ID EQUALS type type-definition-recursive

type-definition-recursive : SEMICOLON type-definition | ε

variable-definition-part : VAR variable-declaration | ε

variable-definition : ID variable-definition-variable COLON type SEMICOLON variable-definition-recursive | ε

variable-definition-variable : COMMA ID variable-definition-variable | ε

variable-definition-recursive : variable-definition | ε

type : simple-type /\*| array-type\*/

array-type : ARRAY LBRACK index-type RBRACK OF simple-type

index-type : ID | index-range

index-constant : sign INTVAL | CHARVAL | sign constant-name

index-range : index-constant DOTDOT index-constant

simple-type : STRING | INTEGER | REAL | CHAR

constant-name : ID

sign : ADD | MINUS | ε

procedure-and-function-definition-part : procedure-declaration SEMICOLON | function-declaration SEMICOLON | ε

procedure-declaration : PROCEDURE ID

LPAREN formal-parameters RPAREN SEMICOLON

block

procedure-and-function-definition-part

formal-parameters : ID formal-parameters-variable COLON type

formal-parameters-variable : COMMA ID formal-parameters-variable | ε

function-declaration : FUNCTION ID

LPAREN formal-parameters RPAREN

COLON type SEMICOLON

block

procedure-and-function-declaration-part

statement-part : compound-statement

compount-statement : BEGIN statement statement-recursive SEMICOLON END

statement : simple-statement | structured-statement

statement-recursive : SEMICOLON statement statement-recursive | ε

simple-statement : assignment-statement | procedure-statement | application | read-statement | write-statement

assignment-statement : variable ASSIGN expression

procedure-statement : ID

application : ID LPAREN expression application-recursive RPAREN

application-recursive : COMMA expression application-recursive | ε

read-statement : READ read-statement-part | READLN read-statement-part

read-statement-part : LPAREN ID read-statement-recursive RPAREN

read-statement-recursive : COMMA ID read-statement-recursive | ε

write-statement : WRITE write-statement-part | WRITELN write-statement-part

write-statement-part : LPAREN expression write-statement-recursive RPAREN

write-statement-recursive : COMMA expression write-statement-recursive | ε

structured-statement : compound-statement | if-statement | while-statement | for-statement

if-statement : IF expression THEN statement if-statement-else

if-statement-else : ELSE statement | ε

while-statement : WHILE expression DO statement

for-statement : FOR ID ASSIGN expression for-statement-to expression DO statement

for-statement-to : TO | DOWNTO

expression : simple-expression expression-relational

simple-expression : sign term expression-add

expression-add : add-term term expression-add | ε

add-term : ADD | MINUS | OR

term : factor term-mult

term-mult : mult-term factor term-mult | ε

mult-term : MULT | IDIV | DIV | AND

factor : application | variable | constant | NOT factor

expression-relational : relational-operator simple-expression | ε

relational-operator : EQ | NEQ | LT | LTE | GTE | GT

variable : ID | ID LBRACK expression RBRACK

paramenter-identifier : ID

constant : constant-no-id | sign constant-identifier

constant-no-id : constant-number | sign constant-identifier | CHARVAL | STRINGVAL

constant-number : sign INTEGERNO | sign REALNO

constant-identifier : ID

# IO module

The IO module is responsible for reading files and writing output. This is called by the scanner to get the characters required for lexical analysis.

## Assumptions

This section details the assumptions for the IO module.

### Standard C library FILE handling

The IO module, to remain as cross-platform as possible, uses the basic C-library functionality for FILE handling.

## Current Status

This section details the current status of the IO module.

### Read / Write / Unget

Currently, the IO module can simply read a character from a FILE, write to the stderr FILE, and unget a character back into a FILE stream.

## Future Work

This section details the future work for the IO module.

### Read line-by-line

Currently, characters are read in individually. However, it would be a nice feature to have the IO module load in entire lines, storing them into a buffer, and providing the characters for the scanner from the buffer. This will allow lines to be printed before lexemes, rather than afterward.

# Symbol Table

The symbol table is responsible for keeping track of all lexemes and their values in the program, which varies by scope.

## Assumptions

This section details the assumptions for the symbol table.

### Initialized by the scanner

The symbol table requires the parser for context; therefore, only the keywords are initialized into the symbol table by the scanner.

## Current Status

The section details the current status of the symbol table

### Basic functionality

Currently, the symbol table has basic linked-list style functionality, with support for multiple scopes. However, this should be changed to a hash table for the final product.

### Scopes

Scopes have full support in this deliverable, changing between blocks.

## Future Work

This section details the future work for the symbol table.

### Hashtable

The symbol table uses link-listed to store values, which is inefficient compared to a hashtable. For the next deliverable, the symbol table will use a hashtable instead.