Porygon Language
Second Stage: Lexical Analysis

Juliana Mayumi Hosoume

Universidade de Brasília
Departamento de Ciência da Computação
Compilers 2020.1 - A - Prof. Cláudia Nalon
ju.hosoume@gmail.com 18/0048864

Abstract. Processing and pattern recognition performed on a huge amount of data in an optimized manner is a daunting task. In order to surpass these caveats, a programming language based on the C language is presented. This language features a new type to represent a data table. In the first stage of development, the design and motivations for the development of this language were highlighted. In the current stage, the emphasis is on the lexical analysis, therefore the patterns for lexeme recognition and the creation of tokens that will be forwarded to the parser are detailed. Moreover, the token information that will be stored in the symbol table structure is presented.

Keywords: Compiler · Programming Language · Data Science

1 Introduction

Since the early 1960s, programming languages, such as FORTRAN and ALGOL 60, aid programmers in the task of implementing algorithms (BG96). These languages are abstractions, a set of instructions that can be used to express a computation (ALSU07). With the diversification of problems solved by computers, a myriad of programming languages was developed to best suit different domains (Ban14).

Data science is a novel and growing domain of computer science. The increase of computational power has enabled the manipulation and study of a massive amount of data. In this sense, many languages assist and automate data mining and statistical evaluation of data. Python and R are examples of widely used programming languages in this field (SPB$^+$18).

In this work, there is an effort to design a language geared toward data manipulation. This language, named Porygon, should make data efficiently accessible and organized. For that purpose, a table data structure is suggested to augment the C language. To this end, a compiler to translate Porygon to three address code will be constructed.

In the previous and first stages of development of the Porygon language, the design, motivation, grammar, and a summary of the language semantics were introduced. In the current stage, the lexical analysis project of the language is exposed. In the lexical analysis, the source program is read by the scanner and, then, divided by the scanner into lexemes according to defined patterns. Finally, with the gathered information, a stream of tokens is generated and sent to the parser (ALSU07).

The design of the lexical analyzer is key to the performance of the compiler (Wai86). To promote the process of lexical analysis, the Fast Lexical Analyzer (*flex*) (PEM12) software is used. This

tool generates a scanner based on supplied patterns. The patterns are defined by a set of regular expressions and, with the aid of C code, describe the desired scanner to the *flex* program.

In the next section (Section 2), the motivations for the new language decisions will be presented. Following, a brief introduction to Porygon syntax and semantics are shown (Sections 3 and 4). Next, the lexical analyzer is detailed (Section 5). Specifically, the regular expressions, token information, and symbol table structure are described. Furthermore, the error handling of the scanner is defined.

2 Motivation

The C language is a general-purpose, widely used programming language (KR88). Nonetheless, this language only offers structs and arrays as built-in data structures. The provided indexing of arrays is lacking and does not allow easy data accessing. Despite these caveats, C language provides constructs that map to machine instructions and is a well-established language, with many compilers for different hardware.

Tables and arrays are data structures of paramount importance to data manipulation. They facilitate data organization and inspection. The Porygon language is designed to be an extension of the C language, implementing both tables and easy array indexing. Calculation of statistical measures and preprocessing of data, such as noise detecting and feature selection, are manageable when proper a data structure is applied. However, tables are complex in the sense that they mix different data types and pose a challenge to efficiently manage memory and to type check. Besides that, a collection of data should be easily accessed, thus suitable operations for collections should be performed in an efficient manager.

3 Formal Description of the Language

Here, the proposed grammar of the Porygon language is presented. The basic patters were removed from this section, since they are further detailed in Lexical Analysis (Section 5). The grammar was not modified in any meaningful way compared to the previous presentation.

3.1 BNF

The grammar of this language uses the notation presented in Section 2.2.1 of Aho et al. book (ALSU07), in this case terminals are the constants specified in definitions, operator symbols and boldface strings. The nonterminals are lowercase, italic names. Note that in table definition the | is used to assist in the table definition, so in the context of Rule 8, it does not mean an *or*. The symbol ϵ represents the empty string.

1. *declarationList* → *declarationList declaration*
   | *declaration*
2. *declaration* → *varDeclaration* **;**
   | *functDeclaration*

---

3. *varDeclaration* → *varSimpleDeclaration*
   | *varSimpleDeclaration* = *logicalOrExp*
   | *arrayDeclaration*
   | *arrayDeclaration* = *arrayDefinition*
   | *tableDeclaration*
   | *tableDeclaration* = *tableDefinition*

4. *varSimpleDeclaration* → *typeSpecifier* **ID**
5. *arrayDeclaration* → *typeSpecifier* **ID** [ ]
6. *arrayDefinition* → { *constList* }
7. *tableDeclaration* → **table** *typeSpecifier* **ID**[ ]
8. *tableDefinition* → |(*stringList*) : *columnContent* |
9. *constList* → *constant*
   | *constList* , *constant*
10. *stringList* → STRINGCONST
    | *stringList*, STRINGCONST
11. *columnContent* → (*constList*)
    | *columnContent* ; (*constList*)

---

12. *functDeclaration* → *typeSpecifier* **ID** (*parameterList*) *compoundStmt*
    | *typeSpecifier* **ID** ( ) *compoundStmt*
13. *parameterList* → *parameterDeclaration*
    | *parameterList* , *parameterDeclaration*
14. *parameterDeclaration* → *typeSpecifier* **ID**

---

15. *compoundStmt* → **{ }**
    | **{** statementList **}**
16. *statementList* → *statement*
    | *statementList statement*
17. *statement* → *varDeclaration* **;**
    | *expression* **;**
    | *iterationStmt*
    | *conditionalStmt*
    | *returnStmt* **;**
    | **ID** = **read() ;**
    | **write(** *baseValue* **) ;**
18. *iterationStmt* → **while (** *expression* **)** *compoundStmt*
    | **for** (*typeSpecifier* **ID**) **in ID** *compoundStmt*
19. *conditionalStmt* → *ifSmtm elseStmt*
20. *ifStmt* → **if (** *expression* **)** *compoundStmt*
21. *elseStmt* → **else** *compoundStmt* | ϵ
22. *returnStmt* → **return**
    | **return** *expression*

---

23. *expression* → *logicalOrExp*
    | *mutable* = *logicalOrExp*
24. *logicalOrExp* → *logicalAndExp*
    | *logicalOrExp* || *logicalAndExp*
25. *logicalAndExp* → *equalityExp*
    | *logicalAndExp* && *equalityExp*
26. *equalityExp* → *relationExp*
    | *equalityExp* == *relationExp*
    | *equalityExp* ! = *relationExp*
27. *relationExp* → *sumExp*
    | *relationExp* > *sumExp*
    | *relationExp* < *sumExp*
    | *relationExp* >= *sumExp*
    | *relationExp* <= *sumExp*

28. *sumExp* → *multExp*
    | *sumExp* + *mulExp*
    | *sumExp* − *mulExp*
29. *multExp* → *unaryExp*
    | *mulExp* ∗ *unaryExp*
    | *mulExp* / *unaryExp*
    | *mulExp* % *unaryExp*
30. *unaryExp* → *baseValue*
    | !*unaryExp*
    | -*unaryExp*
31. *mutable* → **ID**
    | **ID** [*expression*]
    | **ID** [*expression* **:** *expression*]
    | **ID** [*expression* **:** *expression* **:** expression]
32. *baseValue* → *constant*
    | *functCall*
    | *mutable*
33. *functCall*→ **ID (***args* **)**
34. *args* → *argList* | ϵ
35. *argsList* → *expression*, *argList* | *expression*
36. *constant* → INTCONST
    | FLOATCONST
    | BOOLEANCONST
    | CHARCONST
    | STRINGCONST
37. *typeSpecifier* → **char** | **int** | **float** | **bool** | **string** | **void**

Rule 1: start symbol of the program.

Rules 7, 8, 9 and 31: main differences from Porygon to the C language. These rules are associated with tables and indexing. *One issue of the previous report was the lack of explanation for the notation of the table definition. In this case, this notation is used to properly identify the definition of a table type. Since this type has two main components, the labels and the data, the use of a simple array notation does allow the distinction between the two structures for the definition.*

Rule 19: suggested syntax of a loop through elements of an array.

Rule 31: when type checking this rule, the expressions used for indexing should only have INTCONST as the computed value.

4 Semantics

In this section, the definition and the declaration of a table of integers are highlighted. In the context presented, tables should associate a column name with an array. This construction is useful especially to prepare training data for machine learning algorithms.

```
table float balance[];
balance = | ("id", "profit", "debt") : (0, 1, 2); (0.2, 0.9, 108.3); (2.3,
    5.3, 9.6)|;
```

After proper assignment, the information in the table should be easily accessed. Therefore, an array of the type defined is returned when the table is indexed by the column name.

```
balance["profit"]; // Should return  [0.2, 0.9, 108.3], an array of floats
```

One other key feature is a new way of array indexing. Since tables return arrays, a slice of the table can be efficiently obtained, for that, the beginning index, end index and steps (increment) are needed.

```
int values[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
values[3:7:2];
// Should return [4, 6], beginning = 3, end = 7 and steps = 2
```

Frequently, performing operations through all elements of an array is necessary. To that end, a command loop that iterates over an array is a great asset.

```
int values[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int value) in values {
    write(value); // For each iteration should print from 1 to 9
}
```

The Porygon Language uses static scope. In other words, as in the C language, the scope of functions and variables depends on the location of the declarations and can be determined at compile time. In this case, the scope of a declaration is its own block and all the nested blocks inside, except the ones where the name is redeclared (ALSU07). In the Porygon Language, there is the global scope. Nonetheless, a scope can be defined by a block (compound statements), which, in turn, is delimited by curly braces.

The proposed language also uses static typing. Therefore, the type of a declaring is known at compile-time, so the type of a variable or function is defined in its declaration. The language goes towards a strongly typed one. In this sense, only **int** to **float** implicit conversions, or coercions, are performed in the basic arithmetic expressions. In other situations, if the types do not match in the type checking phase, a type exception is returned. The language does not provide explicit conversion, so it does not have *cast* operations.

```
int integer_value = 50;
float float_value = 2.5;
integer_value + float_value; //Should return 52.5 of type float
integer_value / float_value; // Should return 20.0 of type float
```

5 Lexical Analysis

The *flex* program needs an input file that defines rules. Each rule is an association of a regular expression, or pattern, and an action (PEM12). Next, regular expressions and auxiliary definitions are presented. Followed by the token definition with a brief description of the symbol table design. Finally, the scanner error handling and testing are described.

5.1 Patterns

In the algorithm of the Porygon language scanner, after the match of a pattern, the current line is updated and then the token type of the pattern is returned, accompanied by the information

provided by the *flex* in the *yyleng* (length of lexeme) and the *yytext* (characters of the lexeme) global variables. Two names are defined to assist in pattern formation.

```
DIGIT  [0-9]
LETTER  [a-zA-Z]
```

Besides the simple definitions of digits and letters accepted, two starting conditions, a functionality of the *flex*, are stated. These are used to define the beginning of comments in the language. All comments are ignored by the scanner. The /∗ and ∗/ mark the beginning and the end of block comments, which can span multiple lines. In turn, // marks the beginning of line comments that end with \n. Both comments are based on the C language standard (KR88).

```
%x COMMENTLINE
%x COMMENTBLOCK
```

The patterns to define the comments are then expressed in terms of the starting conditions. For simplicity, the function to update the column and line is not shown in the codes here.

```
"/*"                    { BEGIN(COMMENTBLOCK); }
<COMMENTBLOCK>"*/"      { BEGIN(INITIAL); }
<COMMENTBLOCK>\n        { ; }
<COMMENTBLOCK>.         { ; }

"//"                    { BEGIN(COMMENTLINE); }
<COMMENTLINE>\n         { BEGIN(INITIAL); }
<COMMENTLINE>.          { ; }
```

For the eight keywords in the language, the regular expressions are easily defined. The same is true for the seven variable types, eleven syntax symbols and fourteen operators.

- **Keywords:**
  "*while*" "*for*" "*in*" "*if*" "*else*" "*return*" "*read*" "*write*"
- **Variable Types:**
  "*char*" "*string*" "*int*" "*float*" "*table*" "*bool*" "*void*"
- **Operators:**
  "+" "−" "∗" "/" "%" "!" "<" "<=" ">" ">=" "!=" "==" "||" "&&"
- **Symbols:**
  "{" "}" "[" "]" "(" ")" ":" ":" ";" "|" "=" ","

The regular expressions of some constants are more complex, except for the boolean constants.

- **Boolean Constants:**
  "*true*" "*false*"
- **Number Constants:**
  $[\+\-]?\{DIGIT\}+$ RE to identify integer constant
  $[\+\-]?\{DIGIT\}+"."\{DIGIT\}*$ RE to identify float constant

- **Char Constants:**
  '(.)' RE to identify char constant
  \"(.)\" RE to identify string constant

For the char constants, some improvements can be made. It may be necessary to check certain characters like \ followed by " or /. The length of constant chars will be assessed, more details in Section 5.3.

The identifier follows the C standard. Any identifier should contain only characters, digits and underscore. An identifier cannot begin with a digit and cannot contain whitespace. The Porygon language is case sensitive. Thus, *num*, *indx*, *value*, *meanProfit*, *_cases* are all examples of valid identifiers.

– **Identifiers:**
$(\{LETTER\}|"\_")(\{LETTER\}|\{DIGIT\}|"\_")*$

Lastly, there are two rules. One for recognizing the whithespace, including space, newline and tabs, which are all ignored. The other one matches all the possible lexemes. This rule is useful to define a proper action for unknown lexemes.

– **Break line, tabs and whitespace:**
$[\quad \backslash n \backslash t \backslash f \backslash v \backslash r]$
– **Default Rule:**
.

The pattern order of definition is important. If two or more matches are found of the same length, the pattern that comes first is the selected one. Therefore, constants, identifiers and the default rule are the last ones listed. All these patterns were directly implemented in the patterns section of a *.l* flex input file.

5.2 Tokens

The design proposed for the Porygon language delegates the management of tokens to the parser. Whilst the scanner generates the tokens based on the tokenization process, the parser inserts them on the symbol table, since the parser is more informed than the scanner about the identifiers (ALSU07). In this context, the scanner creates token instances composed of token type, attribute value and the token location in the source file.

The token type specifies the token and aid the parser in the syntax checking process. The attribute value defines exactly the lexeme found. In the case of the Porygon language, the created token has its value as a text that will be handled further in the compilation process. The location is the number of the line and the column in which the lexeme was found in the source code.

```
struct token create_token(enum token_type tok_type, const char * value, int
    line, int col) {
    struct token tok;
    tok.tok_type = tok_type;
    tok.att_value = value;
    tok.line = line;
    tok.column = col;
    return tok;
}
```

In order to indicate all the different matches to the parser, a total of 48 token types are needed. Additionally, three more token types are specified to report errors found by the scanner. The use of these tokens is specified in Section 5.3.

The symbol table is planned to be structured as a hash table. Because the Porygon language proposes the implementation of scopes beyond the global scope, a chained symbol table structure may be required. The symbol table will have a key-value pair. The key will hold the identifier name. The value will be a *struct* with the information of name, type, location and attribute value. To implement the hash table, the headers implemented in the uthash will be used (HO18).

5.3 Error Handling

In the proposed language, the scanner only identifies lexical errors. These are passed to the parser as error tokens. Hence, the scanning does not stop on an error. There are three different types of lexical errors:

1. **ERR_INVALID_ID**: Token type returned when the identifier is longer than 32 characters. In the C language, for internal identifiers, i.e. not used in external linkage, at least the first 31 characters are significant, depending on the implementation (KR88).
2. **ERR_INVALID_CHARCONST**: Token type returned when the char constant, for instance $'a123'$ or $'here'$, are larger than one character.
3. **ERR_UNKNOWN_TOKEN**: Token type returned when none of the regular expressions presented in Section 5.1 matches.

All the error tokens contain the information of the line and column, so further in the compiler the error can be handled correctly. An uncertainty is whether to inform the user of the lexical error in the scanner or group the errors in the parser. One benefit of letting the parser output the error is to maintain the order of printing. So, if the parser finds a syntactic error in line one, an invalid token in line five will not be warned before that.

5.4 Testing

Multiple tests will be done to check the correctness of the lexical analyzer. The following code is a valid example in the Porygon language.

```
/* This is a sample correct (lexically) input Porygon File */

int sample_function(void) {
    return 93 + 72;
}

float another_function(float p1, float p2, char c1 /* comment */) {
    return p1 % p2;
}

bool third_func(bool b) {
    bool var = ((7 == 2) > 9);
    var = var && true;
    return ((1 - 9) <= 10 ) || !false;
}

int main(void) {
    /* Do some computations */
```

```
    int one_var = 3;
    int one = 10 / 10;
    float another;
    float result;
    another = 7.98; /* This comment should not appear */

    bool ttrue = !false;

    string input = read();

    write("Running implicit conversion.");

    if (ttrue) {
        result = another - one_var;
    } else {
        result = one_var + another;
    }


    return 1;
}
```

Contrarily, the next code has errors. In line 4, the constant char has more than one character. Also, in line 6, the identifier has more than the maximum length. In line 7, there is an unrecognized @ symbol.

```
/* This is a sample incorrect (lexically) input Porygon File */

int main(void) {
    /*Incorrect char constant (bigger than one character) in line 5 col 14*/
    char k = 'some very big char';
    float ________;
    /* Incorrect long definition of an identifier in line 8 col 9*/
    int thisisnotlegalintheClanguagebecauseitistolongandshouldnotbeconsidered
anidentifier;

    /* Incorrect @ at and $ at line 12 col 9 and col 15*/
    int @a = 3$;
}

float someFunct(flaot a, float b) {
    float something = a +b;
    return something;
}
```

5.5 Compilation and Execution

In order to make a full compilation of the scanner, the flex should be installed. A Makefile was made to aid in the compilation process. More information can be found in the *READM.md* file. To make a full compilation of the scanner:

```
$ make clean_all; make flex; make;
```

In a successful compilation, a *scanner* executable will be created in the *bin* directory.

The input file of the scanner can be passed as an argument:

```
$ ./bin/scanner tests/t2_correct.prg
```

Or from a pipe:

```
$ ./bin/scanner < tests/t1_correct.prg
```

The operating system used to developt and test the implementation was Ubuntu 20.04, with gcc 9.3 and flex 2.6.4.

5.6 Implementation Details

Besides the implementation of the patterns and tokens shown Section 5.1 and Section 5.2, some other functions were created. One function was created to help debuging, since the *enum* type is not easily printed out.

```
1 void print_token(struct token tok);
```

To keep track of the location of the lexems in the source code, a function was made to update the current line and column as the lexems are matched.

```
1 void update_pos(void);
```

Main function of the scanner. This function make the tokenization based on the flex module.

```
1 void run_scanner(void);
```

Further information can be found on comments on the source code.

Bibliography

[ALSU07] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, editors. *Compilers: principles, techniques, & tools*. Pearson/Addison Wesley, Boston, 2nd edition, 2007. OCLC: ocm70775643.

[Ban14] Arvind Kumar Bansal. *Introduction to programming languages*. Chapman and Hall/CRC, 1st edition, 2014. OCLC: 929769460.

[BG96] Thomas J. Bergin and Richard G. Gibson, editors. *History of programming languages II*. ACM Press ; Addison-Wesley Pub. Co, New York : Reading, Mass, 1st edition, 1996. Meeting Name: History of Programming Languages Conference.

[HO18] Troy D. Hanson and Arthur O'Dwyer. uthash User Guide, 2018.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall, Englewood Cliffs, N.J, 2nd edition, 1988.

[PEM12] Vern Paxson, Will Estes, and John Millaway. Lexical Analysis with Flex Manual, 2012.

[SPB+18] Yoav Shoham, Raymond Perrault, Erik Brynjolfsson, Jack Clark, James Manyika, Juan Carlos Niebles, Terah Lyons, John Etchemendy, and Barbara Grosz. The AI Index 2018 Annual Report. Technical report, AI Index Steering Committee, December 2018.

[Wai86] W. M. Waite. The cost of lexical analysis. *Software: Practice and Experience*, 16(5):473–488, May 1986.