Porygon Language
A Data Manipulation Language
Fourth Stage: Semantic 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 second 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. In the third stage, a syntax analysis for the Porygon language was designed. The abstract syntax tree was constructed in the previous stage and will be annotated and explored in the current stage, the semantic analysis. The foundations to translate a source Porygon code into an intermediate code is done in this stage.

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 first stages of development of the Porygon language, the design, motivation, grammar, and a summary of the language semantics were introduced. In the previous 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 third stage, a syntax analysis was developed. The development was aided by the GNU Bison (CT20) software. The previous lexical analyzer communicates with the syntax analyzer using tokens. In accordance with the tokens matched in the input program, the automaton generated by the bison groups the tokens logically (LJ09). The automaton was constructed based on the defined grammar rules of the language. Simultaneously, the abstract syntax tree was constructed. This structure will be used in the semantic phase.

In the current stage, the abstract syntax tree (AST) is used to provide information about the source code. By traversing the AST, all expressions are type-checked. Besides, information stored in the symbol table is also used to check other semantic errors, as multiple definitions of functions, variable re-declaration, and other errors.

In the next section (Section 2), the motivations for the new language decisions will be presented. Following, a brief introduction to Porygon lexical and syntax are shown (Section 3). The lexical analyzer is detailed (Section 3.1). Specifically, regular expressions and token information are described. Details about the symbol table and abstract syntax tree are presented in Section 3.3. In Section 4, are presented information about the language semantics, including scope definition, parameters passing and implicit type conversion rules. Furthermore, the error handling of the scanner, parser and semantic analysis are defined in Section 5.

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 efficiently performed.

3 Formal Description of the Language

Here, the proposed grammar of the Porygon language is presented. The basic patterns are further detailed in Lexical Analysis (Section 3.1). The grammar and its modifications are shown in Sections 3.2 and 3.3.

3.1 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.

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. If /∗ starts a comment block and a ∗/ is not found, the program considers all text after /∗ until EOF as a comment block. 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:**
  $[\backslash + \backslash -]?\{DIGIT\}+$ RE to identify integer constant
  $[\backslash + \backslash -]?\{DIGIT\} + "."\{DIGIT\}*$ RE to identify float constant

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

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 whitespace, 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.

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 aids 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 indicating the number of line and column was removed, since this information can be obtained from the lex variables.

```
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;
    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 (*these tokens may be excluded in future versions, since they are unreachable variables*). The use of these tokens is specified in Section 5. The tokens types are redefined by the bison generated syntax analyzer output. Therefore, the token type names are maintained, however, the numbers were redefined accordingly.

3.2 BNF

The grammar of this language uses the notation presented in Section 2.2.1 of Aho et al. book (ALSU07). In the following, terminals are the constants specified in the lexical definitions, operator symbols and boldface strings. The nonterminals are denoted using lowercase, italic fonts. 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**
| **void**

---

15. *compoundStmt* → **{ }**
| **{** statementList **}**
16. *statementList* → *statement*
| *statementList statement*
17. *statement* → *varDeclaration* **;**
| *expression* **;**
| *iterationStmt*
| *conditionalStmt*
| *returnStmt* **;**
| **read(ID) ;**
| **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*
    | **(** *expression* **)**
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 17: explicitly specifies the construction of input and output statements. Note that these functions have fixed grammar rules, since they are not going to be handled as common functions of the language.

Rule 18: 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.

Grammatical Changes Minor changes were made in the grammar. All the expected features remain, nonetheless some rules were modified to solve some shift/reduce conflicts. Besides, now the language also includes expression inside parentheses. Following, the main changes in the rules are presented:

Rule 6: Braces were replaced by brackets, since brackets are used to limit the scope.

Rule 14: Functions can have no parameter or a void specification.

Rule 17: Since the read function is hard-coded in the grammar, to avoid conflicts, the function was simplified. Instead of returning a value to a mutable, it modifies a received variable.

Rule 32: Included rule to accept parentheses in order to indicate the precedence of expressions.

3.3 Syntax Analysis

In the syntax analysis, the input program is hierarchically structured. In this stage, it is verified whether the program is composed of valid constructs of the language. The parser communicates directly with the scanner. If the input strings cannot be generated by the grammar, errors are reported. Finally, translation steps and construction of the parser tree are concurrently performed (ALSU07, WSH13).

Symbol Table The symbol table is structured as a hash table. The symbol table has a key-value pair. The key holds the identifier name. The value is a *struct* with the information of name, types, scope, a pointer to tree node, parameters and definition specification. The key is formed of the concatenation of the identifier name and its scope. There is also information on whether the identifier refers to a function or a variable, therefore both are stored in the same symbol table. To construct the hash table, the headers implemented in the uthash are used (HO18).

```
struct st_entry {
    char identifier[64];            /* ID name + Scope */
    char name[32];                  /* ID name */
    enum id_type id_type;           /* Function, Variable or Parameter */
    char type[15];                  /* Type of the entry as a string */
    enum ttype dec_type;            /* Declared type (e.g. integer, float...)
    */
    int scope;                      /* Integer that is related to scope*/
    enum special_var spec_var;      /* Simple, array or table*/
    int size;
    bool defined;                   /* Check if the variable has been defined
     */
    struct params_entry *params;    /* Hold parameter values */


    UT_hash_handle hh; /* Makes the strucutre Hashable */
};
```

Each parameter name and its type are also present in the symbol table. The parameters are stores in another hash table, one for each function (*params*). All the parameters will be displayed as not defined in the symbol table, since the value will only be defined when the function is called. The table data structure is going to be dealt with differently. The table data structure will work like a namespace that groups arrays.

Because the Porygon language proposes the implementation of scopes beyond the global scope, a stack structure is required to store the scopes. The utstack (HO18) header is used to implement the stack. The scanner, by identifying braces, increments indicators of the current scope and push or pop the scopes from the stack accordingly. This leads to some changes in the grammar. The arrays are now identified by brackets, instead of braces.

Abstract Syntax Tree The abstract tree is constructed based on generic nodes. The nodes can have multiple leaves and a root. All the nodes are stored in an auxiliary list of nodes. This list facilitates the organization and deallocation of all the nodes. For now, the constructed tree does not have all the characteristics of an Abstract Syntax Tree, as operators are not grouped, for instance (Section 2.8.2 of (ALSU07)). This issue will be fixed in the next phase.

The scanner creates leaf nodes and hands them to the parser. The leaf nodes contain the information of the terminal tokens of the language. The parser chain the nodes together in agreement with the grammar rules. Each node is composed of information about the constructs of the language. In the semantic stage, all nodes will have synthesized and inherited attributes about types and other values to allow the semantic checking, as the type checking.

4 Semantics

In this section, the definition and the declaration of a table of integers are highlighted. Moreover, scope rules for variables and functions, parameter passing mechanisms and type conversions are presented. In the construction of the compiler, a one-pass approach is planned to be used, by applying the backpatching. This approach is required to aid in the resolution of control flow statements.

The table data structure is the main feature of the Porygon language. 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[] = | ("id", "profit", "debt") : (0, 1, 2); (0.2, 0.9,
    108.3); (2.3, 5.3, 9.6)|;
```

After the 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
}
```

4.1 Scope

The Porygon Language uses a 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 block and all the nested blocks inside, except the ones where the name is redeclared (ALSU07). In the Porygon Language, there is a global scope. Nonetheless, a scope can be defined by a block (compound statements), which, in turn, is delimited by curly braces.

Whenever a curly brace is recognized, a new scope is created. The scopes are managed by a stack as explained in Section 3.3. All the identifiers have a defined scope. Therefore, when an identifier is

referenced, the symbol table is looked up to retrieve its value, based on the current scope (Figure 1).

```
int id1 = 10;
write(id1); /* Writes 10, id1 in the global scope */
void sf(int id1) {
    write(id1);
}
sf(12);  /* Writes 12, id1 in the scope of sf (1) */
```

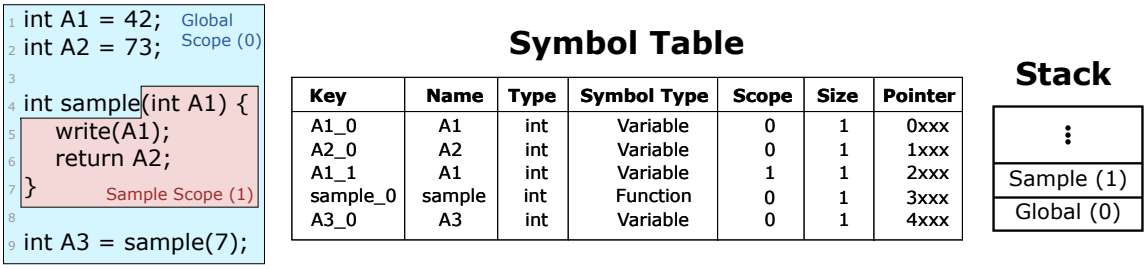

Fig. 1. Example of scope resolution in the Porygon language. In the left side is a sample recognized code. Different colors indicate different scopes. In the right there are the symbol table and stack constructions

4.2 Type Conversion

The proposed language also uses static typing. Therefore, the type associated with a declaration 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 (Figure 2). In other situations, if the types do not match in the type checking phase, a type exception is raised (Section 5.3), even for comparison operators. 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
```

4.3 Parameter Passing

In the design of the Porygon language, considering the current stage, there are no pointers and the parameters are passed as values. Therefore, it only uses a "call-by-value" mechanism. Hence, when a function is called, the actual parameters are bound to the formal parameters by copying evaluating the expression and copying the values (ALSU07).

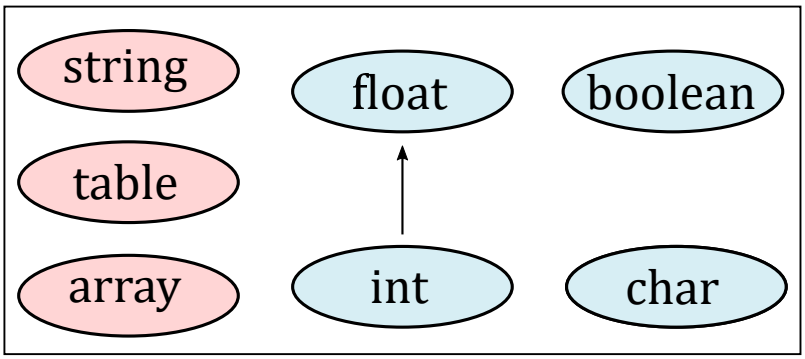

Fig. 2. Types of the Porygon language. In blue are the basic type, whilst in red are more complex structures composed of multiples elements of a defined type. The arrow indicates the implicit conversion of an integer to a float number.

```
/* Sample function with three parameters */
int sample_function (int num1, int num2, string output) {
    num1 = num1 + 3;
    write(output);
    return num1 * num2;
}

/*
    When the function is called, 2 is bound to num1, 3 is bound to num2
    and "hey" is bound to output. All the function input values are
    copied to the location of the formal parameters of the sample function.
*/
int in = 2;
int result = sample_function(in, 1 + 1 + 1, "hey")
/* In this case, in does not change.  */
```

Nonetheless, as the semantic analysis is under development, a "call-by-reference" mechanism is desired to specially handle arrays and tables. In this case, passing these data structures can save memory space. Additionally, since there is no copying, the computation can be faster.

5 Error Handling

5.1 Lexical Errors

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. For now, error tokens appear in the grammar as unused variables. This warning will be fixed in the next phase. 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 3.1 matches.

The *yyerror* function from the parser is used to assist in the notification of lexical errors.

5.2 Syntax Errors

In the parser, the error is handled by the *yyerror* function. All the syntax errors, or, in a simplified explanation, any sequence of tokens that do not match any of the rules, trigger the error handler, which then prints the detailed error in the screen (default output). In some cases, the errors are duplicated, because some errors are printed in the standard output and in the *stderr* concomitantly.

As specified in the Bison manual (CT20), the compiler should try to finish the parsing, even if some failures occur. Thus, in some rules, a Bison *error* marker was included. As when a baseValue is wrongly defined.

```
baseValue
    : LPARENTHESES expression RPARENTHESES          {$$ = $2;}
    | error                                         {++synt_errors;}
    | constant                                      {$$ = $1;}
    | functCall                                     {$$ = $1;}
    | mutable                                       {$$ = $1;}
    ;
```

In most cases, the implemented parser can identify all the errors in the files, nonetheless it may halt and does not proceed to detect all the errors before some of them are fixed.

5.3 Semantic Errors

The semantic errors are identified by the semantic analyzer. Both the symbol table (Section 3.3) and abstract syntax tree (Section 3.3) have crucial information to allow the properly highlight semantic errors. Whenever such an error is found, an error message is printed.

One of the main concerns in the semantic analysis is the type checking. The abstract syntax tree is traversed to identify possible inconsistencies in the defined expressions types. Based on the rules defined in Section 4.2, whenever possible, the types are implicit conversed. However, in some cases, i.e. coercion of a table to a string, a conversion is not feasible, then the semantic analyzer reports the error.

Other errors are centered on identifiers and the symbol table. An identifier cannot be repeated in the symbol table in the same scope. Thus, when a variable or a function is re-declared, an error is thrown. Similarly, when a function has incompatible arguments (*under development*), as a wrong number of arguments or types mismatch, an error is reported. Moreover, when a variable or a function is used before being defined in the current or parent scopes, an error is thrown.

When a program does not provide a main function declaration, an error is reported. Besides, if there is a type mismatch between the function declared type and its returned type, an error is evoked. In the case of multiple syntactic errors, some semantic errors may not be handled properly, since entries may not be inserted correctly on the symbol table. Besides, some semantic errors are difficult to be assigned specifically to a line, since context is required.

1. **Type Mismatch**: Only numerical values can be coerced. The use of a type in the context of a different type is considered a semantic error. This error is identified not only on expressions, but also on function calls (argument type mismatch) and function returns (each return statement is evaluated individually).
2. **Simple Variable Called as Function**: A declared variable cannot be called as a function.
3. **Function Used as Variable**: A function cannot be used as means of state definition. Thus, a value can only be obtained from a function through calls.
4. **Function Redefinition**: A function cannot be redefined in the same scope, therefore two different functions cannot have the same name in a given scope.
5. **Function Call Before Definition**: A function cannot be called before being defined in the current or in parent scopes.
6. **Variable Re-declaration**: A variable cannot be re-declared in the same scope.
7. **Variable Use Before Declaration**: A variable cannot be used before being declared in the current or in parent scopes.
8. **Variable Use Before Definition**: The value of a variable cannot be retrieved before being defined.
9. **Function call without specifying all arguments**: A function cannot be called if any of its parameters is not defined by an argument.
10. **Array indexing without using integer**: An array cannot be indexed by any other type than an integer.
11. **If-statement without Boolean conditional**: An if-statement needs a boolean expression in the conditional part.

6 Testing

Multiple tests were done to check the correctness of the lexical, syntax and semantic analyzers. In the *test* directory provided along with the program, there are 14 different test files, four without lexical, syntactic or semantic errors and ten with different types of errors that will be specified bellow. Files that start with *extra* are contributions from other students that evaluated this work. The parser will try to continue evaluation, even if some errors are found. Nonetheless, the presence of lexical and syntax errors can trigger some semantic errors, as the *syntax tree* is not correctly constructed or the symbol table is not updated.

Files *extra1_correct.prg*, *extra2_correct.prg*, *t1_correct.prg* and *t2_correct.prg* should not report any code errors. Following are highlighted the lines that contain errors in the incorrect codes.

extra1_incorrect.prg Syntax error in Line 21, instead of ";", there is a ":". The file presents an illegal character "?" in Line 23 (lexical error). Some semantic errors are thrown because of lexical and syntax errors. The same goes for the other incorrect files.

extra2_incorrect.prg In Line 8 there is a syntax error, lack of the right-hand side of the rem operator. The file has two other examples of illegal characters. Line 14 has a "@" symbol and Line 24 has a "#" symbol. The language does not contain both strings.

t1_incorrect.prg More illegal characters are presented. In Line 5, there are "@" and $ symbols.

t2_incorrect.prg In Line 5, the constant char has more than one character. Also, in Line 8, the identifier has more than the maximum length. In Line 13 the type is misspelled, thus there is a syntax error.

```
1  /* This is a sample incorrect (lexically) input Porygon File */
2
3  int main(void) {
4      /*Incorrect char constant (bigger than one character) in line 5 col 14
       */
5      char k = 'some very big char';
6      float ________;
7      /* Incorrect long definition of an identifier in line 8 col 9*/
8      int
       thisisnotlegalintheClanguagebecauseitistolongandshouldnotbeconsidered
9  anidentifier;
10
11 }
12
13 float someFunct(flaot a, float b) {
14     float something = a +b;
15     return something;
16 }
```

t3_incorrect.prg This file has an incorrect number of parentheses in Line 2, Column 10.

t4_incorrect.prg Duplicated symbol of sum is a syntax error, found in Line 2, Column 8.

t5_incorrect.prg This file does not have the program required structure, so a syntax error is thrown in Line 1. Besides, since the main function is not defined, at the end of the program, a semantic error is thrown indicating the absence of this function.

t6_incorrect.prg File is missing a semi-colon in Line 3, nonetheless this error is identified only in Line 7, when an unexpected symbol is found.

t7_incorrect.prg This file and the subsequent ones are used to check semantic errors, so there are no lexical or syntax errors.
Line 13: Re-declaration of variable.
Line 18: Redefinition of function *int_func*. This error is only caught in the end of the function declaration statement.
Line 21: Indexing without using an integer.
Lines 25 to 32, and Line 41: Type Mismatch.
Line 35: Use of function as a variable (not being properly called).
Line 38: Calling of a variable.
Line 41: Operator "!" can only be applied to a boolean.
Line 54: Missing function return.

t8_incorrect.prg This file verifies if the returns are correctly checked. In Line 5, the return of a float does not match the defined type of the function. Meanwhile, in Line 6 there is a correct

type casting of int to a float.

t9_incorrect.prg In this file, function declaration and call are tested.
Line 13: Wrong number of arguments.
Line 15: Type coercion of the argument (not an error).
Line 21: Wrong number of arguments.
Line 22: Wrong number of arguments.
Line 26: Argument type mismatch.
Line 27: Argument type mismatch.

t10_incorrect.prg This file checks a function composed of multiple return statements. Two return functions in Lines 7 and 9 do not match the function declared type. However, since the parser only identifies the function type in the end of the declaration, both the errors are only thrown at Line 14.

t11_incorrect.prg Line 10: Wrong number of arguments.
Line 16: Conditional expression of an if-statement is not a boolean.
Line 17: Not void function does not define a return value.

7 Compilation and Execution

In order to make a full compilation of the scanner and parser, the flex and bison should be installed. A Makefile was made to aid in the compilation process. More information can be found in the *README.md* file. All flags needed to generate warnings are already provided. To make a full compilation of the program:

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

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

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

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

Or from a pipe:

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

The operating system used to develop and test the implementation was Ubuntu 20.04, with gcc 9.3, flex 2.6.4 and bison 3.5.1.

8 Implementation Details

Besides the implementation of the patterns and tokens shown in Section 3.1 and Section 3.1, 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 lexemes in the source code, a function was implemented to update the current line and column as the lexemes are matched.

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

The stack, symbol table and AST need to be allocated and deallocated. So a group of functions were created.

```
struct st_entry *symbol_table = NULL;
struct node_list *ast_tree_list = NULL;
struct tree_node *ast_root = NULL;

scope_stack *sp_stack = NULL;

/* ... */
ast_tree_list = initialize_list();
/* ... */

free_st();
free_list(ast_tree_list);
free_stack();
```

For the implementation of the semantic analyzer many functions were implemented, especially to type check. The next function presented receives a node and, accordingly to its node type (language construct), verifies if the node has valid types.

```
void check_type(struct tree_node *node);
```

Meanwhile, to check the return types of a function, a recursive traverse of the leafs of a function declaration node was implemented in the following function.

```
bool check_return(struct tree_node *node, enum ttype func_type);
```

To store information about the parameters of a function a hash table was used (HO18). This hash is stored in another hash table, the symbol table. The structure and auxiliary functions are presented here.

```
struct params_entry {
    int param_indx;
    enum ttype dec_type;
    char name[32];

    UT_hash_handle hh; /* Makes the strucutre Hashable */
};

struct params_entry *find_param(struct params_entry **func_params, int
    param_indx);
void add_param(struct params_entry **func_params,
               int param_indx,
               enum ttype dec_type,
               const char *name
           );
void force_add_param(struct params_entry **func_params,
               int param_indx,
```

```
17                  enum ttype dec_type,
18                  const char *name
19              );
20 void free_params(struct params_entry **func_params);
21 void print_params(struct params_entry *func_params);
22 int num_params(struct params_entry **func_params);
```

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

8.1 Challenges

One of the main challenges faced in the current stage is the difficulty to visualize all the required information to correctly create the intermediate code. To translate successfully and accurately, auxiliary information of the source code needs to be stored in structures as the AST and symbol table beforehand, in an efficient and accessible manner.

Unfortunately, some erroneous decisions in the previous stages as the implementation of a concrete syntax tree instead of a proper AST and the use of lexical error tokens demand a large number of changes in many modules. Thus, these issues will only be solved in the next phase of the compiler. Besides, some features as showing a warning when an array is out-of-bounds indexed could not be fully implemented yet.

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.

[CT20] Robert Corbett and GNU Project Team. Bison manual. https://www.gnu.org/software/bison/manual/, 2020.

[HO18] Troy D. Hanson and Arthur O'Dwyer. uthash User Guide. http://troydhanson.github.io/uthash/userguide.html#_what_can_it_do, 2018.

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

[LJ09] John Levine and Levine John. *Flex amp; Bison*. O'Reilly Media, Inc., 1st edition, 2009.

[PEM12] Vern Paxson, Will Estes, and John Millaway. Lexical Analysis with Flex Manual. http://courses.softlab.ntua.gr/compilers/2013a/flex.pdf, 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] William McCastline Waite. The cost of lexical analysis. *Software: Practice and Experience*, 16(5):473–488, May 1986.

[WSH13] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 1st edition, 2013.