CS 315

Project Report

Argert Boja 21503525

Ndriçim Rrapi 21500342

Section 3

Language “B#”

Part 1: BNF DESCRIPTION

The BNF description of our language is given below and further explanation will be given in the next sections:

<start> → **begin** <stmt_list> **end**

<stmt_list> → <stmt> | <stmt> **;** <stmt_list>

<stmt> → <bool_var> = <bool_expr>

|<bool_const> = <bool_value>

|<int_var> = <ath_expr>

|<string_var> = <string_expr>

| <mun_stmt>

|<loop_stmt>

|<funct_declare>

<bool_expr> → <bool_val>

| <call_predicate>

| <bool_var>

| <bool_var><connectives><bool_expr>

|<bool_const>

|<int_var><comparison><int_var>

<ath_expr> → <int_var><add> <int_var> | <int_var><sub> <int_var>| <int_var><mult><int_var>| <int_var> <int_var>

<string_expr> → <string_var> | <string_var> <add> <string_var>

<mun_stmt> → <matched_if_stmt> |<unmatched_if_stmt>

<matched_if_stmt> → **in case** (<bool_expr>) <mun_stmt> | <stmt>

<unmatched_if_stmt>→ **in case** (<bool_expr>) <mun_stmt>

| **but in case** (<bool_expr>) <mun_stmt >

| **in case** (<bool_expr>) <matched_if_stmt> **otherwise** <unmatched_if_stmt>

<loop_stmt> → **loop** (<bool_expr>) <stmt>

<funct_declare> → **declare** <predict>(<param_list>) { <stmt_list> ; **return** <bool_value>}

<call_predicate> →<predict>(<arg_list>)

<predict> → **predict** <uppercase> {digit|letter}

<param_list> → <param> | <param>, <param_list>

<param> → **:** <bool_var> **:** | **:** <string_var> **:** | **:** <const_var> **:** | **:** <int_var> **:**

<arg_list> → <arg> | <arg>, <arg_list>

<arg> →**:** <bool_var> | **:** <string_var> | **:** <const_var> | **:** <int_var>

<bool_var> → **booleantype** <id>

<bool_const> → **booleantype const** <id>

<bool_value> → **TRUE** | **FALSE** | **AMBIGUOUS**

<int_var> → **integertype** <id>

<string_var> → **stringtype** <id>

<connectives>→ <and> | <or> | <imply> | <iff> | <ebnb> | <negation>

<comparison> → <beq> | <lt> | <lte> | <gt> | <gte>

<id> → **#**{letter|digit}

2. Detailed Explanation

- **<start>**
  It defines the start of the program.

- **<stmt_list>**
  It defines the list of the statements that will be included in the program.

- **<stmt>**
  It defines a single statement. There are in total 7 different types of statement assignments in our language, which may be: boolean variable assignment, boolean constant assignments, integer variable assignment, string variable assignment, if statements, loop statements, and function declarations. All of the above will be explained in detail in the descriptions below.
- **<bool_var>**
  It defines a boolean variable which may be assigned truth values of expressions multiple times.
  Example:

  ```
  booleantype #bool1 = TRUE;
  booleantype #bool2 = FALSE;
  booleantype #bool3 =  #bool1 & #bool2;
  ```

  As it can be seen from the example **booleantype** corresponds to the type of the variable to be assigned, and it is then followed by our base identifier which should start with the **#** sign and continue with alphanumeric which will define the name of our variable. So, on the third line we show an example of a boolean variable being assigned the truth value of a propositional formula.

- **<bool_const>**
  It defines a constant boolean variable which may be assigned truth values only once.
  Example:

  ```
  booleantype const #constant1        = TRUE;
  booleantype const #anotherConstant = FALSE;
  ```

  As it can be seen from the example **booleantype** corresponds to the type of the constant to be assigned, which then is followed by **const.** Afterward we have yet our base identifier starting with **#** sign and continues with alphanumerics. This base identifier will define the name of our constant. In this example we have assigned true and false values to **constant1** and **anotherConstant** respectively. The boolean values are only assigned once for the constants.

- **<int_var>**
  It defines an integer type variable which may be assigned integers only.
  Example:

  ```
  integertype  #int2 = 18;
  integertype  #int3 = 8;
  ```

#int3 = #int2 + #int3;

As it can be seen from the example **integertype** corresponds to the type of the variable to be assigned, and it is then followed by our base identifier which should start with the **#** sign and continue with alphanumeric, which will define the name of our integer variable. So, on the third line we show an example of an integer variable being reassigned new values as the addition of two integer variables.

- **<string_var>**

  It defines an string type variable which may be assigned string characters only.

  Example:

  stringtype #str2 = "Programming is";
  stringtype #str3 = "fun";
  #str2 = #str2 + #str3;

  As it can be seen from the example **stringtype** corresponds to the type of the variable to be assigned, and it is then followed by our base identifier which should start with the **#** sign and continue with alphanumeric, which in term defines the name of our string variable. So, on the third line we show an example of a string variable being reassigned a new string value as the addition of two string values. The addition operator here means string concatenation.

- **<mun_stmt>**

  The abbreviation means matched or unmatched and it corresponds to the way we have designed our if statements.

  So if we are writing two consecutive **if** statements followed by an **else** statement then the second **if** will then match with **else.** This is done in order to avoid further problems in defining which **if** matches to which **else :**

  Example:

  in case (#day -> #word)
  return TRUE;
  in case (' (#day -> #word))
  return FALSE;
  otherwise
  return AMBIGUOUS;

  As it can be seen from the example we denote if statements as **"in case"** and else statements as **"otherwise".** As we said the last "**in case**" on second row will so match with "**otherwise**". (Note that AMBIGUOUS is another type of boolean value we decided to create)

- **<loop_stmt>**

  It defines that this statement will be a loop statement

  Example: (@ means start of a comment)

  until (#input1 >= 0) {
  @testing output
  display "Enter a new number";
  @testing input

```
#input1 = scanin;
}
```

As it can be seen from the given example we denote a loop by **until.** This is followed by open-close round brackets which will contain a boolean logical expression. Afterwards we open-close the curly brackets within which we can write our statements. The given example makes a loop and prompts the user to enter a number and do so until the input number is greater than or equal to 0.

- **<funct_declare>**

  It defines a function declaration

  Example: (@ means start of a comment)

```
@declaration of predictA which is a predicate function with no parameters
declare predictA() {
    stringtype #day = "MONDAY";
    stringtype #word = "DAYOFWEEK";

    in case ( #day -> #word )
        return TRUE;
    but in case ( ' (#day -> #word) )
        return FALSE;
    otherwise
        return AMBIGUOUS;
}
```

  As it can be seen from the given example a function declaration should start with **declare** followed by a space. Afterwards there should be **predict** followed by alphanumerics. As denoted we open-close the circular brackets meaning there are no parameters to pass in this example. And then open-close the curly brackets. Inside the curly brackets we can then write our statements. The method should return boolean value. Note again that in **B#** we also have **AMBIGUOUS** as another type of boolean value alongside **TRUE** and **FALSE.**

- **<bool_expr>**

  It defines a Boolean expression which may be defined in several different ways.

  Example:

```
#bool2 = #bool1 & const #const1;
#bool1 = ( #int1 == #int2 );
#bool2 = predictB (:#str2, :#str3);
```

  As it can be seen from the examples above, a Boolean expression can be expressed as a connective, as a comparison result, as a function call, or as a constant value. More of Boolean expression types can be found in the input file provided in the attachments.

- **<bool_val>**

It defines the value of a Boolean. In B# Programming Language, we have three Boolean values which are TRUE, FALSE and AMBIGUOUS. The first two values are the same as in all other programming languages while the last one is created by us, which is used in cases when it is difficult to define whether a result is true or false.

Example:

```
booleantype #bool1 = TRUE;
booleantype #bool2 = FALSE;
booleantype #bool4 = AMBIGUOUS;
```

- **<call_predicate>**

  It defines a call to a function which returns a truth value. This truth value is assigned to a variable.

  Example:

  ```
  #bool1 = predictA();
  #bool2 = predictB (:#str2, :#str3);
  ```

  As can be seen from the above examples, before each function call, we have used the special word **predict** followed by an uppercase character and then optional alphanumeric characters. Each function call, might or might not have arguments, depending on the function declaration which will be defined below.

- **<connectives>**

  It defines the connectives that we use in our programming language such as AND, OR, IMPLY, NEGATION, IF AND ONLY IF, EITHER BUT NOT BOTH. These connectives in our programming language are defined by the special characters &, |, ->, ‘ , <-> , _v_ respectively.

  Example:

  ```
  #bool2 = #bool1 & const #const1;
  #bool3 = #bool2 | const #const1;
  #bool1 = #bool3 -> const #const1;
  #bool3 = #bool1 <-> const #const2;
  #bool2 = #bool2 _v_ const #const3;
  #bool1 = ' #bool3;
  ```

- **<comparison>**

  It defines the comparison types that we use in B# Language. These comparisons are ==, <=, >= , <, > which have the same meanings as in calculus.

DESCRIPTION OF NON-TRIVIAL TOKENS

As in all the other programming languages, even in our language we have included some tokens which will be used for ***comments, identifiers*** *and* ***reserved words.*** The need for these nontrivial tokens was significant since they affect the readability of a programming language.

In order to define tokens for the comments, we used the special character **@** followed by other characters. In this case, all the characters that follow this special character will be neglected and not treated as tokens. Since the character **@** is not used for any other purpose, there were no constraints in defining this nontrivial token. The role of comments is to help the programmer to increase the readability of the program that has written.

In the case of identifiers, we used the special character **#** followed by a lowercase letter and then alphanumeric characters. These identifiers are used for variable's names and also for constants' names. Even in this case, the usage of a special character in front of the token, makes is easier to identify the token as an identifier, and also increases reliability since this special character will not be used for any other purpose. Furthermore, defining identifiers in this way, also makes it easier to read the code.

Since reserved words are words that will be used only for a special purpose, we were restricted to define these words in a way that they wouldn't be confused with other definitions. Hence, for these reserved words we decided to use no special character in front of them. Some of the reserved words that we use in our language are: **start, until, just in case, otherwise,** etc. The meaning and usage of each of these special words was defined in the section above.

As literals, we tried to use characters which would increase the writability by using characters that express themselves what they are used for. For example, some of the literals in our language are **&** which stands for **AND, ->** which stands for **IMPLY**, etc.

These nontrivial tokens will increase the readability, writability and reliability of **B# Programming Language!**