CSC 3315

Project - Part 2: High-Level Language Design

Prepared by:

IRAOUI Imane

EL AMRANI Omar

LAMIRI Fatima Zahrae

DRIOUECH Saad

Submitted by team captain:

DRIOUECH Saad

Supervised by:

Dr. Violetta Cavalli Sforza

Spring 2021

Team Dynamics:

As the week before the break was dedicated to working on the second assignment, and the break itself was an opportunity to refuel, we had little time to work on the project itself. Moreover, due to mismatched schedules, many of our members had to make sacrifices(whether that be a bit of sleep or just lunch breaks) for the sake of the overall team. As was explained in the midterm report, we had to make due with the little bit of time we had left and properly manage it. After Saad discussed with you on Teams our two base language options, we started working immediately the day after(Tuesday) on the lexemes. We defined the building blocks of our PL and started working on lexical and syntactic descriptions of the language.

Summary of Tasks:

As we stated earlier, we all discussed the overall design of the language at the start before splitting our tasks. Both Saad Driouech, and Fatima Zharae Laamiri worked on the BNF and regex description of the language. Imane Iraoui wrote the programs, and Omar El Amrani wrote the Language Design Overview, and Requirement Check. We reunited again later to proofread and review all of the different parts, and every single member had different input about what to add in specific parts of the project.

Language Design Overview

When we first set out to make our language, we made it clear from the beginning that our goal is to make a general-purpose imperative PL. As such, there were bound to be similarities between our creation and other similar languages such as C, Visual Basic, and Python. However, to make sure that we are not restrained by our historical C bias, we chose, as stated before, Darija as the base language for our lexeme. Due to the limited scope of the language and not having libraries to support it, the language will inevitably look like a rudimentary version of all the ones listed above with no big defining characteristic that would separate it from the pack. However, it should still be powerful enough to give programmers the tools needed for general-purpose coding.

Using regex, we were able to define core components of our PL, that go as such: user-defined identifiers, numeric literals, character literals, string literals, operators, white space, punctuation, comments, and reserved words. Most of these are self-explanatory (i.e. all the literals, white space, and punctuation), but others deserve more explanation and elaboration. As with the specifications of the project, we had four types of operators, the details of which can be found in the Requirement Check table. The user identifiers can start with an underscore or any letter of the english alphabet(lower case or major case), but you can add numbers after the first letter. As such, a user identifier can look like this: _t880opS.

Continuing on with the language specifications, comments are any text that resides between two “!!”, as we defined using regex. Finally, for the reserved words, we have a handful: hbess | ramz | aadad | tabita | mahed | rejaa | walou | ila | ila ghalat |bda | sali | mouhima. Their english equivalent would be: break | char | int | constant | while | return | void | if | else | begin | end | main. As you could see, these reserved words do not match one to one to any specific imperative language but draw from many (mainly C and Pascal)

This concludes the explanation for the Regex part of our language definition. Now, let's talk about our BNF specifications. The program can optionally start with constant definitions, global variables, and function definitions before entering the main. Once inside a function (main included), we can have statements. A statement can be in the form of variable declarations, which can be either simple or structured. One of our main design choices was to make sure that variables could be declared in the middle of the program, for instance in the middle of compound statements. One of the biggest problems in C in our opinion is the inability to declare variables close to where they're relevant. A statement can also be: assignment, selective, or repetitive. The requirement check goes into more detail about these types of statements. Another type would be function calls, which allow us to make use of nested functions—another previously specified language element.

Moreover, we implemented two built-in functions: "tebaa", and "qra". Tebaa is a similar function to print in many imperative languages. It takes hard coded strings and expressions as input and posts them on the console. Qra, on the other hand, is similar to scanf. It takes user input and assigns it to user identifiers.

As a last note, "rejaa", and "hbess", the equivalent of return and break also fall under the definition of statement. The BNF explains their roles which does not differ from their C counterpart.

Requirement Check

Requirement	Explanation
Constants	Constants can be defined by using the "tabita" special word. The notation would look like this: tabita a 5
Types	In the BNF grammar of our language, we specified both simple and structured types. The simple types can be addressed by &var, and the structured one would be var[var]. Note that the structured types can either be integers (aadad) or characters (ramz).
Variables	A variable can start with any of the latin letters, and underscore, after which, you can add numbers but not symbols. We also do not allow casting, and as such, a variable cannot change its type within the same function.
Functions	Functions can have two types of parameters (ramz or aadad). They are passed by value, as through the specifications of the BNF. Here is an example of a function definition: ramz function_name (ramz var1, aadad var2) bda …. !!Body of the function!! …. sali Here is an example of a function call: function_name(var1, var2) => var3;

	As is clear with the examples, the output is a single result that is of the predefined simple data types, and so are the parameters.
Operators	As with the professor’s instructions, we strayed away from C’s assignment operator and used our own: => For the arithmetic operators we used the following: + | - | / | * | % For the comparison operators we used the following: < | > | = | <= | >= | != For the logical operators we used the following: MACHI | O | WLA, which corresponds to not, and, or.
Expressions	We have two types of expression, arithmetic ones, and logical ones.
Statements	The assignment statement is written as follow: Expression => identifier Example: a + 5 => b For the selection statements, we have two: ila, ila ghalat, which correspond to if, else if. The first one is a 1-way selection while the second is a 2-way one. Example: ila (a < b): …. !!Body of if!! …. : For repetition statements we use the reserved word “mahed”, which is the equivalent of a while-loop. Example: mahed (a < 10): …. !!Body of while!! …. :
Control Structures	The flow of our PL follows the same structure as all the other imperative language, in the sense that it is largely sequential. The only statements that can break that flow are ila and mahed.
Program Structure	The main program is indicated with the keyword “mouhima”, and is limited the same way other functions are limited: bda, sali. Example: mouhima bda … !!body of the main!! … Sali As stated before, the functions are delimited using **bda sali**. In the BNF, we specify that it’s possible to have nested functions. The reason being is that statements can be function calls, and since we have the ability to put statements inside functions, we’re automatically able to have nested functions. In the same spirit as the nested functions, we’re able to declare variables inside compound statements.

Comments	We defined comments using regex. To showcase that some text is a comment, you need to put it between a pair of “!!”.

Lexical Description - using Regular Expressions

regex for user defined identifier [a-zA-Z_][a-zA-Z_0-9]*
regex for numeric literal syntax 0 | (+|-)?[1-9][0-9]*
regex for character literal syntax [a-zA-Z_0-9]
regex for string literal syntax [a-zA-Z_0-9]+
regex for operators (=> | < | > | = | <= | >= | != | + | - | / | * | % | MACHI | O | WLA)
regex for white space ( )
regex for punctuation(; | , | :)
regex for comments (!![a-zA-Z_0-9]*!!)
regex for reserved words (hbess | ramz | aadad | tabita | mahed | rejaa | walou | ila | ila ghalat | bda | sali)

Syntactic description - using BNF

<program> ::= <constants> <declarations>
<constants> ::= tabita identifier aadad; <constantstemp> | ∈
<constantstemp> ::= <constants> | ∈
<declarations> ::= <declaration><declarationstemp>
<declarationstemp>::= <declarations> | ∈
<declaration> ::= <varDeclaration>; | <fctDefinition>
<varDeclaration> ::= <varType> <variables>
<variables> ::= identifier, <variablestemp>
<variablestemp> ::=<variables>| ∈
<varType> ::= <simpleType> | <structuredType>
<simpletype> ::= <Stype> | <address>
<Stype> ::= aadaad | ramz
<address> ::= &<Stype>
<structuredType> ::= aadad[<expression>] | ramz[<expression>]
<fctDefinition> ::= <Stype> identifier (<parameters>) bda <statements> sali | identifier (<parameters>) bda <statements> sali
<parameters> ::= <parameterList> | ∈
<parameterList> ::= <parameter><parametertemp>
<parametertemp> ::= ,<parameterList>| ∈
<parameter> ::= <Stype> identifier
<statements> ::= (<assignmentStmt> | <selectionStmt> | <repetitionStmt>| hbess; |<returnStmt> | <varDeclaration> | <fctCall>) <statementstemp>
<statementstemp> ::= <statement> | ∈
<fctCall> ::= identifier(IDs) => identifier; | identifier(IDs); | <qra>; | <tebaa>;
<IDs> ::= identifier<identifiertemp> | ∈
<identifiertemp> ::= ,<IDs> | ∈
<returnStmt> ::= rajaa <returnedVal>;

<returnedVal> ::= identifier | aadad |∈
<assignmentStmt> ::= <expression> => identifier;
<selectionStmt> ::= ila (<condition>): <statements>: <selectionTemp>
<selectionTemp>::= ila ghalat: <statements>: | ∈
<repetitionStmt> ::= mahed (<condition>): <statements>:
<expression> ::= <expression> <operator> <expression> | <operand> | (<expression>)
<condition>::= <operand> <CompEx> <operand> <conditionTemp>
<conditionTemp>::= <LogEX>(<condition >) | ∈
<operand>::= identifier | aadad
<operator> ::= + | - | / | * | %
<CompEx>::= < | > | <= | >= | = | !=
<logEX> ::= WLA | O | MACHI
<qra> :== qra(<IDs>);
<tebaa> :== tebaa(<combination>);
<combination> ::= “text” <combination> | <expression> <combination> | ∈

Short program:

This program computes average of 3 numbers:

```
mouhima()
bda
aadad grade1, grade2, grade3, sum,avg;

qra (grade1);
qra (grade2);
qra (grade3);

grade1 + grade2 + grade3 => sum;
sum / 3 => avg;
tebaa (avg);
sali
```

Sample input:

```
97
95
90
```

Sample output:

```
94
```

Medium program:

This program finds, given an array, if there are any of subsets of its elements such that their sum is even. It outputs -1 if there is no such subset, otherwise it outputs the number of elements in the subset obtained and it prints the positions of the elements of such subset. If there are multiple subsets, it prints an example.

```
mouhima()
 bda
 !!declaring variables needed!!
 !!m: number of elements in the given array!!
 !!temp: variable that holds the number of subsets!!

 aadad m, arr[100],temp, j, var;
 -1 => temp;
 0 => j;
 qra (m);

 ila (m > 1):
!!reading the content of the array!!
     mahed (j < m):
     qra (arr[j]);
     j + 1 => j;
     :

     0 => j;
     mahed (j < m):
!! if an element of the array is even, then we choose a subset
consisting of that only element and print 1 as well as its
position.!!
          ila (arr[j] % 2 = 0):
            tebaa(1);
            tebaa (j+1);
            hbess;
          :
     !! initialize temp to 0 in the first iteration
          ila (temp = -1):
             j => temp;
          :
     !! if there is no even element in the first two elements
     of the array, then we display the position of these odd
     elements since their sum is even!!
          ila ghalat:
            tebaa (2);
            tebaa (temp+1);
            tebaa (j+1);
            hbess;
```

```
                    :
               :
          :
          !!case where the array only has one element!!

                ila ghalat:
                    qra (var);
         !! if that one element is even we print 1 (position of the
         element) and 1 (number of elements)!!
                 ila (var%2 = 0):
                       tebaa (1);
                       tebaa(1);
                 :
               !! if the only element is odd, then we print -1 to signal
               that there is no subset!!
                   ila ghalat:
                       tebaa (-1 );
                   :
               j+1 => j;

               :
         sali
```

Sample input:

```
      5
      300 100 1 2 3
```

Sample output:

```
      1
      1
```

Long program:

This program finds the maximum common divisor possible of p elements in the array. It outputs the maximum GCD.

```
mouhima()
bda
aadad n, i, x, j, k, p, o;
aadad arr[10000];

0 => o;

  qra (x); !!number of elements in the array!!
  qra (p); !!number of elements with maximum possible gcd!!
  o + 1 => o;
  0 => j;

 !! reading the content of the array!!
  mahed (j < x) :
    qra (arr[j]);
     j + 1 => j;
  :
  !! finding the maximum number in the array!!
  aadad high = arr[0];
     0 => 0;
  mahed (i < x) :
    if (arr[i]>high)
    arr[i] => high;
     i + 1 => i;
  :
     !! if the number of elements in the required subset is 1, then
     the highest gcd is the highest number since it divides itself!!
  ila (p = 1) :
      tebaa (high);
  :
     !!array to keep track of the count of numbers that each divisor
     divides!!
  aadad div [high+1];
     j=0;
     !! initializing the array to 0!!
  mahed (j < high) :
    0 => div[j];
     j + 1 => j;
  :
     0 => j;
  mahed (j<x) :
     1 => k;
     !! finding the number of divisors of each element !!
    mahed (k <= arr[j]) :
     !! if a divisor of an element is found, then the count is
     incremented!!
     !! div[i] is the number of elements that i can divide!!
         ila (arr[j] % k = 0) :
           div[k] + 1 => div[k];
        :
     k + 1 => k;
    :
     j + 1 => j;
  :
   0 => k;

  aadad arr2 [high+1];
     0 => i;
     !! copying all divisors that can divide p elements into another
     array to find their max!!
  mahed (i < high) :
    ila (div[i] = p) :
      i => arr2[k];
      k + 1 => k;
    :
     i + 1 => i;
  :
     !! finding the highest number that can divide p elements!!
  aadad max2;
  arr2[0] => max2;
  0 => i;
  mahed(i < high) :
    ila (arr2[i] > max2):
    arr2[i] => max2;
    :
     i + 1 => i;
  :
  ila(p != 1) :
    tebaa (max2) ;
  :
```

```
 sali
```

<u>Sample input:</u>

```
      5 2
      300 100 1 2 3
```

<u>Sample output:</u>

```
      100
```