Language name: sparky
File extension: '.spk'
GitHub: https://github.com/ParikshithKedilayaM/SER502-Spring2020-Team7

Design:

1. The purpose of the language

   To help implement algorithms and encode solutions to complex problems.
   We also want to exhibit the ability of declarative languages in implementing imperative languages.

2. Generic stuff that shows what we have thought through while designing:
   - Our language will have Primitive Types Boolean ('and', 'or' & '!'(not)), Integers, Float, and String.
     - boolean (we have true, false, and, or,! (not) as keywords)
     - var keyword supports, 'int','float' & 'string' as data types

   Operations support addition, subtraction, multiplication, and division for Integers and Float.

   - Our language has a way to associate a value with an identifier, hence it will support assignment operators on all supported data types.
     - Identifiers should start with a character from 'a' to 'z' in lower case. The environment supports a string of characters and underscore as identifier names.
   - Our language supports the following conditional constructs:
     - A ternary operator (condition? expression1: expression2). It ends with a semicolon.
     - if-then-endif construct.
     - if-then-else-endif construct.

   - The language must support the following looping structures
     - Traditional for loop.
       - **for(i:=2; i<5; i:= i + 1)-do-endfor;**
       - **for(i:=2; i<5; i++)-do-endfor;**
     - Traditional while loop
       while (boolean) do block endwhile;
     - **'for i in range (2 to 5)-do-endfor'** which will behave the same as **for(i:=2; i<5; i:= i + 1)-do-endfor**.

   - We have a construct display for displaying the identifier values. It will output all the data types supported by the language.
     - display(identifiers+). Where identifiers must be of the same data type.
     - display(expressions+). Will print the evaluated results.
     - display(strings+). Will print a single string or a concatenation of multiple strings.

   - Evaluating Boolean algebra expressions

     - X + Y, when X is true, Y is false, **"+**" acts as **"or"** operator.
     - X * Y, when X is true, Y is false, **"*"** acts as **"and"** operator.
     - Evaluation of parse tree for this will be taken care of in semantic analysis

     Milestone 2 update:
     - This was not implemented.
     - Justification: We implemented this as 'boolean expressions for conditional constructs' because we believed that the above mentioned boolean algebraic expressions would have been redundant.

- The precedence of operators for conditional constructs will be 'not(!)', 'and', 'or' in decreasing order.
- The conditional statement will support '>,<,>=,<=,==,!=' only.

- Concatenation of strings is supported.
  - When two or more identifiers, both should be of type string.

  print (a+b). Where a and b should be declared as variables having values of type string.
  - **print (a +"<text>")**. Where a should be declared as variables having values of type strings. <text> will be a combination of characters from 'a' to 'z' in lowercase or uppercase, spaces, and all special characters.
- In expressions, if there are variables of different data types, the parser will allow it during parse tree generation, but we will not be evaluated during semantic analysis.

3. Declaration and Initialization:
   We have a declaration section followed by commands in a block of code. Commands, in turn, may have multiple blocks of code. Initialization can be done during declaration or in the commands section. We don't use any datatype keywords for declaring variables, instead var keyword supports all datatypes.
   Eg var day := "Monday";
   var days := 30;
   Milestone 2 update:
   Commands will not have blocks inside it since we thought it was redundant. This has been removed from the final code.

4. Commands:
   We have assignment operation, for loop, if-then-else condition, while loop, ternary operator, display.
5. Default values:
   No. If a variable is not declared, an error will be thrown at runtime.
   The counter variable in for loop need not be initialized.
   If a variable is declared without an initial value, it is initialized with the number '0'.
6. Additional features for the future:
   a. Functions – Thoughts so far:
      i. We need to decide from where the function call will happen.
      ii. We need to decide on how to deal with global variables.
      iii. Functions will support primitive data types as input parameters and return values.
      iv. The environment for functions would be represented using a list of lists construct while evaluation which will help define the scope of each function.
      v. The idea is to map each function to a block predicate within a scope i.e. a specific environment as mentioned above.
   b. We plan to add support for variables starting with upper case.
   c. We plan to add support to have code-level comments.

   Milestone 2 update:
   a. The above ideas we not implemented due to time constraints.
   b. Instead we have implemented a list structure to support stack and queue operations.
   c. The design is as follows:
      - List will be initialized using list keyword. Eg list a;
      - We can check if the list is empty or not using list->isEmpty() command which will return true or false accordingly.
      - We have push and pop predicates which can be called to push and pop elements from a list. We have pushFirst(), pushLast(), popFirst(), popLast() predicates which can be used based on requirements. So, the programmer will be given freedom to use list both as a stack and queue.

      Eg. List->pushFirst(5); list->popLast().

7. Tools used:
   Java 8, Prolog, Maven, ‘jpl’ 7.4.0 as a Maven dependency.

Lexical analysis:

- The lexical analyzer has been built using Java.
- The code written by the programmer will be saved in a file with extension ‘.spk’.
- The file will be fed to a Java program that will generate the list of tokens.

Compiler implementation:

- The DCG is built in Prolog.
- There is a Java code in place which uses the ‘jpl’ library to pass the tokens to the DCG and generate the parse tree.

Interpreter implementation:

- The code to evaluate the parse tree is written in Prolog
- There is a Java code in place which uses the ‘jpl’ library to pass the parse tree to the evaluate predicate in Prolog to generate the output for the program.

Sample Program 1:

```
begin
        var teamName := "Team7";
        var teamMembers := 4;
        var emptyInitalize ;
        list listExample ;

        print(teamName);
        print(teamMembers);
        print(emptyInitalize);
        print(listExample);
end
```

Sample Program 2:

```
begin
        var var1 := "Hello";
        var var2 := " World!";

        print(var1 + var2 );
        print("Hello" + var2);
```

```
end
```

Sample Program 3:

```
begin
        var a := 5;
        var b := 6;
        var c := 7;


        print("Test for if");

        if (a == b or b != c)
        then
        print("Inside If");
        else
        print("Inside Else");
        endif;


        print("Test for While");

        while ( a < 10)
        do
                print(a);
                a++;
        endwhile;




        print("Test for traditional for loop");
        for(i := 0; i <10; i++)
        do
                print(i);
```

```
        endfor;

        print("Test for advanced for loop");
        for j in range(10 to 1)
        do
                print(j);
        endfor;



end
```

Sample Program 4:

```
begin
        var tr :="TRUE";
        var fl := "FALSE";
        var x := 8;
        var y := 5;
        var z := 6;
        var t := 6;
        if (x != y and t == z  or z != t and !t == z and true)
        then
           print(tr);
        else
                print(fl);
        endif;
end
```

Sample Program 5:

```
begin

var num1 := 1.1;
var num2 := 2.2;
var str1 := "SER";
var str2 := "502";
var str,add,sub,mul,div;
```

```
add := num1+num2;
sub := num1-num2;
mul := num1*num2;
div := num1/num2;
str := str1+str2;

print("ExpressipnEvaluation");
print(add);
print(sub);
print(mul);
print(div);
print("StringConcatination:");
print("Str1:");
print(str1);
print("Str2:");
print(str2);
print("Str1+Str2:");
print(str);

end
```