**Language name**: sparky

**File extension**: ‘.spk’

**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.

1. 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.
  + - int
    - float
    - string
    - boolean (we have true, false, and, or, not as keywords)
    - var - supports above data types

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

* Our language will have a way to associate a value with an identifier, hence it will support assignment operators on all supported data types.
  + - Identifiers will be represented by a single character from ‘a’ to ‘z’. If the environment supports, we will make it a string of alphanumeric characters.
* Our language will support the following conditional constructs:
  + A ternary operator (condition? expression1: expression2). It ends with a semicolon.
  + if-then-else 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)**
    - Traditional while loop **while (expr == expr) do block endwhile   
      while (not boolean) do block endwhile  
      while (boolean) do block endwhile**
    - ‘for i in range (2 to 5)’ which will behave the same as **for (i = 2; i < 5; i = i + 1)**.
    - The conditional statement will support ‘>,<,>=,<=,==,!=’ only. The ‘and, or’ symbols in the DCG rules will be taken care of automatically while evaluation.
  + We will have a construct **display** for displaying the identifier values.
    - It will output all the data types supported by the language. display(identifiers+).
    - Display - returns string or output of expressions after evaluation.
  + 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
  + Concatenation of strings will be supported → Evaluation will be taken care of in semantic analysis.
  + In expressions, if there are variables of different data types, the parser will allow it during parse tree generation, but we will invalidate that in semantic analysis.

1. 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.

1. Commands:

We have assignment operation, for loop, if-then-else condition, while loop, ternary operator, display.

1. Default values:

No. If a variable is not initialized, an error will be thrown at runtime.

1. Additional features for the future:
   1. Functions – Thoughts so far:
      1. We need to decide from where the function call will happen.
      2. We need to decide on how to deal with global variables.
      3. Functions will support primitive data types as input parameters and return values.
      4. The environment for functions would be represented using a list of lists construct while evaluation which will help define the scope of each function.
      5. The idea is to map each function to a block predicate within a scope i.e. a specific environment as mentioned above.
   2. We plan to add support for variables starting with upper case.
   3. We plan to add support to have code-level comments.
2. 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.

**Grammar:**

digit --> [X], {number(X)} .

alphaNumeric --> [X],{atom(X), X \= true, X \= false}.

identifier --> alphaNumeric .

endLine --> [;].

endPeriod --> [.].

equal --> [=].

var --> [var].

begin --> [begin].

end --> [end].

program --> block,endPeriod.

block --> begin, declrList,commandList,end.

/\*

\* Declaration Parsing

\*/

declrList --> declR, endLine,declrList.

declrList --> declR, endLine.

declR --> var, identifier,equal,digit.

declR --> var, identifier,equal,["'"],alphaNumeric,["'"].

declR --> var, identifier,equal,booleanI.

declR --> var, identifierList.

identifierList --> identifier.

identifierList --> identifier,[','], identifierList.

/\*

\* Commands Parsing

\*/

commandList --> commandI,endLine,commandList.

commandList --> commandI,endLine.

commandI --> display.

commandI --> commandInitialize.

commandI --> ifEval.

commandI --> forEval.

commandI --> whileEval.

commandI --> identifier ,[:,=], ternaryEval.

commandInitialize --> identifier,[:,=],expr.

commandInitialize --> identifier,[+,+].

commandInitialize --> identifier,[-,-].

ifEval -->[if],['('],booleanComb,[')'],[then],commandList, [else], commandList, [endif].

ifEval -->[if],['('],booleanComb,[')'],[then],commandList, [endif].

ternaryEval --> booleanComb,[?],expr,[:],expr.

forEval --> [for],['('],commandInitialize,endLine,booleanComb,endLine,commandInitialize,[')'],[do],commandList,[endfor].

forEval --> [for],identifier,[in],[range],['('],digit, [to],digit,[')'].

whileEval --> [while],['('],booleanComb,[')'],[do],commandList,[endwhile].

/\*

\* Boolean Parsing

\*/

booleanComb --> booleanI.

booleanComb --> boolean.

booleanI --> [true].

booleanI --> [false].

boolean --> [!],booleanComb.

boolean --> expr,equal,equal,expr.

boolean --> expr,[!],equal,expr.

boolean --> expr,conditional,expr.

conditional --> [>].

conditional --> [<].

conditional --> [>,=].

conditional --> [<,=].

conditional --> [and].

conditional --> [or].

:-table expr/2, term/2 .

/\*

\* Expression Parsing

\*/

expr --> expr, [-], term.

expr --> expr, [+], term.

expr --> term.

term --> term, [\*], factor.

term --> term, [/], factor.

term --> factor.

factor --> ['('],expr,[')'].

factor --> digit.

factor --> identifier.

factor --> booleanI.

factor --> alphaNumeric.

display --> [display], ['('],expr,[')'].

**Sample Program**:

begin var x = 5; var xyz = 10; var abcdef; var y = true; display(x+5); if(x+y==y+z) then x++; else y++; endif; for(i:=1;i<10;i++) do if(x+y==y+z) then x++; else y++; endif; endfor; while(x==y) do if(x+y==y+z) then x++; else y++;

endif;

endwhile;

if (true)

then

while(x==y)

do if(x+y==y+z)

then x++;

else y++;

endif;

endwhile;

endif;

display((x+y)\*(x-y)/(x+x+x+x\*x\*x\*x-x-x-x-x));

end.