LIMPID

Formal Methods in Computer Science

University of Bari

A.Y. 2021-22

Exam Project: ***Language Interpreter written in Haskell***

Student:

Andrea Montemurro

a.montemurro23@studenti.uniba.it

Re. Number:

747997

Index

Introduction

The goal of this project is first to define a grammar for the IMP language, which we studied in the theory, and then use the Haskell programming language to implement the interpreter for such a language.
Of course, the principle of the functional programming will be used to implement such a project.
The **LIMPID**is an interpreter of the **IMP** language that use the classic syntax of the *C Programming Language*.

As first thing, we want to recognize the command that we defined for the IMP language and possibly add an extension.
The command that we are going to implement in our interpreter will be the following:

- ***skip***: command that performs a jump to the next instruction;
- ***assignment***: command that assigns a value to a variable by calculating it if it is written in the form of an expression;
- ***If-then-else***: conditional control structure;
- ***Do-while***: iterative control structure of the do-while type
- ***While-do***: iterative control structure of the while-do type;
- ***For-times***: iterative control structure of the for type (not already defined in the theory, so this will be implemented an extension);

Moreover, the accepted types will be only Integer and Boolean and we are going to use Dynamic Typization.

So, all this stuff defines our goal, but before going to the implementation, we need to define a **grammar**, that we are going to write in the *Backaus-Naur Form*.

You can also find all the code and the material in the GitHub Repository of the project.

Grammar

```
program ::= <command> | <command> <program>

command ::= <skipcommand> | <assignmentcommand> | <ifcommand> | <whilecommand> |
<dowhilecommand> | <forcommand>

skipcommand ::= skip <semicolon>

assignmentcommand ::= <variable> := (<aexp> | <bexp>) <semicolon> |
<letter> := { <array> } <semicolon>

ifcommand ::= if <space> ( <bexp> ) <space> then { <space> (<program> | <program>
else { <space> <program>) } <semicolon>

whilecommand ::= while <space> ( <bexp> ) <space> { <space> do <space> <program>
<space> } <semicolon>

dowhilecommand ::= do <space> { <space> do <space> <program> <space> } while
<space> ( <bexp> ) <space> <semicolon>

forcommand ::= for <space> <variable> <space> times <space> { <space> <program>
<space> }

array ::= <afactor> | <afactor> <colon> <array>

bexp ::= <bterm> | <bterm> <bexpOp> <bexp>

bterm ::= <bfactor> | ( <bexp> ) | ! <bexp>

bfactor ::= <aexp> | <aexp> <comparisonOp> <aexp> | <variable>

aexpr ::= <aterm> | <aterm> <aexpOp1> <aexp>

aterm ::= <afactor> | <afactor> <aexpOp2> <aterm>

afactor ::= <apositivefactor> | <anegativefactor>

anegativefactor ::= - | <apositivefactor>

apositivefactor ::= <number> | ( <aexp> )

number ::= <positivenumber> | <variable>

positivenumber ::= <digit> | <digit> <positivenumber>

variable ::= <letter> | <letter> <variabile>

semicolon ::= ; | ; <space>

colon ::= , | , <space>

digit ::= 0-9
```

aexpOp1 *::= + | -*

aexpOp2 *::= * | /*

bexpOp *::= & | |*

comparisonOp *::= < | > | = | <= | >= | !=*

letter *::= a-z*

space *::= " "*

Design

Considering the previous defined grammar, the Language Interpreter designed is structured in such a way:

- a **parsing component** that performs the syntactic check of the language;
- an **evaluation component** that can evaluate the correct written code.

Thus, the interpreter reads the source code provided in input twice. First for checking the syntax of the code in input, and then for the evaluation that follows these steps:

1. ***Arithmetic expressions*** *evaluation;*
2. ***Boolean expressions*** *evaluation;*
3. *Assignment of the* ***values*** *to the* ***variables****;*
4. *Evaluation of the condition and selecting the statements to execute in case of i****f-then-else****;*
5. *Evaluation of the condition and iterating the instructions to be executed in the case of* ***while-do****,* ***do-while*** *and* ***for.***

When the interpreter evaluates expressions needs to keep in memory the environment of the variables used to bind them.
So, **LIMPID** evaluates expressions and for each assignment of values to variables stores values in the **environment** in correspondence with the relatives variables. If an interpretation of the variable already exists in the environment, this is replaced. Otherwise, if a variable is within an expression, the interpreter will read its value from the environment and will replace it with the value read.
The **environment represents the portion of memory used by a data program**.

Evaluation

In this project the ***Eager Evaluation Strategy*** has been adopted, classic of IMP languages. Therefore, **LIMPID** will evaluate the expressions to be associated with the variables as soon as it encounters the related assignment statements. In the case of composition, in which the evaluation of one expression requires the result of another, the interpreter will use the **call-by-value**.

Implementation

For the i**mplementation of the environment** it has been adopted a list of pairs containing the name of the variable and its value. This environment has been implemented as a type:

type Env = [(String,String)]

The variable names allowed by the language are made up of multiple text characters, so the first value of the pair is of type String (for arrays a single text character and an [index]).

The second value of the pair is of type String because the (calculated) value associated with the variable will be physically replaced in subsequent statements in the source code, and the code is a string.

The String type can be considered as the type of data encoding used in memory for the value of the variable, represented in this case by the Env.

The Env environment must be passed from function to function for each character of the code that is read and be available for any evaluation or assignment, otherwise it will be lost.

type Parser a = Env -> String -> [(Env,a,String)]

Env and String are the types of parameters provided in input, Env is the environment (initially empty) and String is the textual input.
The triple represents the result of the function composed by the Env itself, possibly modified by adding elements if an assignment has been read, from the value returned by the parser in case of correctly recognized input, and the textual input not yet analysed or discarded.

Function implementations are listed below:

item

It will analyse the first character, the environment is the input and returned as output without making changes.

```
item :: Parser Char
item = \env inp -> case inp of
                    []     -> []
                    (x:xs) -> [(env,x,xs)]
```

parserReturn

It will replace the elements a and Env of the triple with those received in input as a parameter and it will return a Parser a with the environment modified following assignments.
• v: parserized portion of code, or the result of an evaluation
• newenv: is the new environment

```
parserReturn  :: Env -> a -> Parser a
parserReturn newenv v = \env inp -> [(newenv, v,inp)]
```

failure

It will stop parsing by returning an empty list.

```
failure :: Parser a
failure  = \env inp -> []
```

+++

Disjunction between parsers, it executes p, if it fails it executes q. The input and the env are transferred from one parser to the other.

```
(+++)  :: Parser a -> Parser a -> Parser a
p +++ q = \env inp -> case p env inp of
                    []         -> parse q env inp
                    [(env, v,out)] -> [(env, v,out)]
```

parse

Runs a parser.

```
parse :: Parser a -> Env -> String -> [(Env, a,String)]
parse p env inp = p env inp
```

>>>=

The sequence operator executes p and then the parser returned by the function f. The env is transferred from one function to another.

```
(>>>=) :: Parser a -> (Env -> a -> Parser b ) -> Parser
p >>>= f = \env inp -> case parse p env inp of
                       [] -> []
                       [(env, v, out)] -> parse (f env v) env out
```

Environment manipulation

Functions for the manipulation of the environment, i.e. the binding of a variable within a block of code, the insertion and modification of a variable:

SetEnv

The environment changes, setEnv adds a new variable name-value pair or changes its value (if the variable already exists in the environment).
• v: variable name
• a: variable value

```
setEnv :: String -> String -> Env -> Env
setEnv v a []  = [(v,a)]
setEnv v a (e:es)
              | (fst e)==v      = [(v,a)] ++ es
              | otherwise       = e:(setEnv v a es)
```

GetEnv

The environment is extracted from the tuples.

```
getEnv :: [(Env, a, String)] -> Env
getEnv []              = []
getEnv [([], _, _)]    = []
getEnv [(x, _, _)]     = x
```

Getmemory

The environment is extracted and is wrote in String form.

```
getMemory :: [(Env, a, String)] -> String
getMemory []              = []
getMemory [([], _, _)]    = []
getMemory [(x:xs, c, s)] = fst x ++ "=>" ++ snd x ++ " " ++ (getMemory [(xs,c,s)])
```

Bind

It interprets the variables (using replace) within the expression passed as a parameter.

• eg: environment of variables
• *xs: expression*

```
bind ::  Env -> String -> String
bind [] xs = xs
```

```haskell
bind es [] = []
bind (e:es) xs = bind es (replace e xs)
```

Replace

It replaces the name of the variable with its value.

- v: is the interpretation of the single variable through the tuple (variable name, value)
- xs: is the expression

```haskell
replace :: (String,String) -> String -> String
replace v [] = []
replace v xs
   | (fst v) `isPrefixOf` xs = (snd v) ++ replace v (drop (length (fst v)) xs)
   | otherwise = (xs!!0) : replace v (drop 1 xs)
```

Constructs

Functions that interpret assignment statements, if-then-else, while-do, do-while and for.

Variable

Variable allows you to have as the name of the variable, a string of any length.
In case of array, only one char can be used for the name.

```haskell
variable    :: Parser String
variable    = sat isLetter >>>= \env c ->
                                (
                                  char '[' >>>= \env op ->
                                    variable >>>= \env v ->
                                      char ']' >>>= \env cl ->
                                        parserReturn env ([c] ++ "[" ++ bind env v ++ "]")
                                )
                                +++
                                (
                                  char '[' >>>= \env op ->
                                    parsenumber >>>= \env num ->
                                      char ']' >>>= \env cl ->
                                        parserReturn env ([c] ++ "[" ++ num ++ "]")
                                )
                                +++
                                (
                                    variable >>>= \env f ->
                                        parserReturn env ([c] ++ f)
                                )
                                +++
                                parserReturn env [c]
```

Assignment

```
parseassignmentCommand :: Parser String
parseassignmentCommand = variable >>>= \env v ->
                          char ':' >>>= \env _ ->
                            char '=' >>>= \env _ ->
                              (
                                parsebexpr >>>= \env b ->
                                  semicolon >>>= \env s ->
                                    parserReturn env (v ++ ":=" ++ b ++ s)
                              )
                              +++
                              (
                                parseaexpr >>>= \env a ->
                                  semicolon >>>= \env s ->
                                    parserReturn env (v ++ ":=" ++ a ++ s)
                              )
                              +++
                              ( --parse the array explicit declaration
                                char '{' >>>= \env opg ->
                                  parsearray >>>= \env arr ->
                                    char '}' >>>= \env cpg ->
                                      semicolon >>>= \env s ->
                                        parserReturn env (v ++ ":=" ++ [opg] ++ arr ++ [cpg] ++ s)
                              )
```

This function recognizes the assignment command, in particular the variable and the assigned value (integer or Boolean), and also the assigment to an element of an array. The value is the result of the evaluation of aexpr or bexpr for an assigment to a variable, while for the assignment to a variable of an array it is call myarray. The core of the function lies in the parserReturn in which the env resulting from the setEnv function is provided as a parameter,
SetEnv is required to associate the value b (bexpr expression), a (aexpr expression) and array (saveArray) to the variable v (which represents the key with which to save the expression) in the env environment. The same for associate the value of afactor expression.
aexpr and bexpr in calculating the expression may encounter a variable, its value will be read from the environment using the bind function.

Saving the elements of an Array in the environment

```
arrayType ::  Parser [Int]
arrayType  = afactor  >>>= \env n ->
                (
                   colon >>>= \env c ->
                     arrayType >>>= \env f ->
                       parserReturn env ([n] ++ f)
                  )
                   +++
                   parserReturn env [n]


saveArray :: Env -> String -> [Int] -> Env
saveArray env var list = foldl (\e v -> setEnv (fst v) (snd v) e) env l
                      where l = zipWith (\val index ->
```

```haskell
                                        (var ++ "[" ++ (show index) ++ "]", show (val) )) list [0..]
```

The functions for saving arrays are arrayType and saveArray. The first, myarray returns an [int] list from the parsing of the elements, taken as a parameter by savearray with the env and the variable, to save the single elements in the env i.e. y:={2,5}; -> y[0]:=2; y[1]:=5;

Following the aexpr and bexpr definitions:

```haskell
aterm                  :: Parser Int
aterm                  = afactor >>>= \env f ->
                                (
                                  char '*' >>>= \env _ ->
                                    aterm >>>= \env t ->
                                      parserReturn env (f * t)
                                )
                                +++
                                (
                                  char '/' >>>= \env _ ->
                                    aterm >>>= \env t ->
                                      parserReturn env (fromIntegral (div f t))
                                )
                                +++
                                parserReturn env f


aexpr                   :: Parser Int
aexpr                   = aterm >>>= \env t ->
                                (
                                  char '+' >>>= \env _ ->
                                    aexpr >>>= \env e ->
                                      parserReturn env (t + e)
                                )
                                +++
                                (
                                  char '-' >>>= \env _ ->
                                    aexpr >>>= \env e ->
                                      parserReturn env (t - e)
                                )
                                +++
                                parserReturn env t

bterm :: Parser Bool
bterm = ((trueKeyword +++ falseKeyword) >>>= \env b1 -> parserReturn env (read b1 :: Bool))
       +++
        (bfactor >>>= \env b2 -> parserReturn env b2)
        +++
(char '(' >>>= \env _ -> bexpr >>>= \env b3 -> char ')' >>>= \env _ -> parserReturn env b3)
       +++
        (char '!' >>>= \env _ -> bexpr >>>= \env b4 -> parserReturn env (not b4))
```

```
bexpr               :: Parser Bool
bexpr               = bterm >>>= \env b1 ->
                            (
                              char '&' >>>= \env _ ->
                                bexpr >>>= \env b2 ->
                                  parserReturn env (b1 && b2)
                            )
                            +++
                            (
                              char '|' >>>= \env _ ->
                                bexpr >>>= \env b2 ->
                                  parserReturn env (b1 || b2)
                            )
                            +++
                            parserReturn env b1
```

ParseNumber

Number is the function that contains variables or integers and can in turn represents integers.

```
parsenumber :: Parser String
parsenumber = parsepositivenumber  +++ (variable >>>= \env v -> parserReturn env v
```

If-then-else

In the if-then-else statement, condition b is evaluated through bexpr. In case of true value only p1 will be evaluated, while p2 will only be parsed syntactically vice versa in the false case (program reads and evaluates, parseprogram reads without evaluating).

```
ifCommand          :: Parser String
ifCommand          = ifKeyword >>>= \env i ->
                       openPar >>>= \env op ->
                         bexpr >>>= \env b ->
                           closePar >>>= \env cp ->
                             openPargraf >>>= \env t ->
                               if b
                                 then
                                   program >>>= \envTrue p1 ->
                                     (
                                       elseKeyword >>>= \env e ->
                                         parseprogram >>>= \env p2 ->
                                           closePargraf >>>= \env ei ->
                                             semicolon >>>= \env s ->
                                               parserReturn envTrue (i ++ op ++ (show b) ++ cp ++ t ++ p1 ++ e
++ p2 ++ ei ++ s)
                                     )
                                     +++ --whithout else branch
```

```
                                    (
                                      closePargraf >>>= \env ei ->
                                        semicolon >>>= \env s ->
                                          parserReturn envTrue (i ++ op ++ (show b) ++ cp ++ t ++ p1 ++ ei ++
s)
                                    )
                                else
                                  parseprogram >>>= \env p1 ->
                                    (
                                      elseKeyword >>>= \env e ->
                                        program >>>= \envFalse p2 ->
                                          closePargraf >>>= \env ei ->
                                            semicolon >>>= \env s ->
                                              parserReturn envFalse (i ++ op ++ (show b) ++ cp ++ t  ++ p1 ++
e ++ p2 ++ ei ++ s)
                                    )
                                    +++
                                    (
                                      closePargraf >>>= \env ei ->
                                        semicolon >>>= \env s ->
                                          parserReturn env (i ++ op ++ (show b) ++ cp ++ t ++ p1 ++ ei ++ s)
                                    )
```

Do-while

```
dowhileCommand  :: Parser String
dowhileCommand  = doKeyword >>>= \env d ->
                          openPargraf >>>= \env opg ->
                            parseprogram >>>= \env p ->
                              closePargraf >>>= \env cpg ->
                                whileKeyword >>>= \env w ->
                                  openPar >>>= \env op ->
                                    parsebexpr >>>= \env b ->
                                      closePar >>>= \env cp ->
                                        semicolon >>>= \env s ->
                                          parserReturn (getEnv (parse program env p)) p >>>=
\envw _
                                            if (getCode (parse bexpr env b))
                                              then
                                                parserReturn (getEnv (parse program envw (d ++
opg ++ p ++ cpg ++ w ++ op ++ b ++ cp ++ s))) (d ++ opg ++ p ++ cpg ++ w ++ op ++ b ++ cp ++ s)
                                              else
                                                parserReturn env (d ++ opg ++ p ++ cpg ++ w ++ op
++ b ++ cp ++ s)
```

In the do-while statement the condition b is not evaluated immediately but is only read (by parsebexpr). Once the code is syntactic checked, the program is executed for the first iteration and the bexpr is evaluated at each iteration.
The block of instructions p is initially read (parseprogram) and then executed at least one time regardless of the condition.

The pair b, p consists of a copy of the do-while block that the interpreter will use as a parameter for the recursive execution of the do-while itself.
At the end of the doWhileCommand function we note that the expression b is evaluated, it will depend on the execution of the recursive call to program to give rise to the iteration, or the termination of the function with return of the result. And then if the condition is True, LIMPID executes the program and then a while with the same commands and the new environment:

```
parserReturn (getEnv (parse program envw (d ++ opg ++ p ++ cpg ++ w ++ op ++ b ++ cp ++ s))) (d
++ opg ++ p ++ cpg ++ w ++ op ++ b ++ cp ++ s)
```

If the condition is False we return the new env (obtained with the result of the first evaluation of the program), but the program is not executed again in this branch because we have done it before the evaluation of the condition (do-while logic):

```
parserReturn env (d ++ opg ++ p ++ cpg ++ w ++ op ++ b ++ cp ++ s)
```

While-do

```
whileCommand        :: Parser String
whileCommand        = whileKeyword >>>= \env w ->
                        openPar >>>= \env op ->
                          parsebexpr >>>= \env b ->
                            closePar >>>= \env cp ->
                              openPargraf >>>= \env ogr ->
                                doKeyword >>>= \env t1 ->
                                  parseprogram >>>= \env p ->
                                    closePargraf >>>= \env cgr ->
                                      semicolon >>>= \env s ->
                                        if (getCode (parse bexpr env b))
                                          then
                                            parserReturn (getEnv (parse program env p)) p >>>= \envw _
                                              parserReturn (getEnv (parse program envw (w ++ op ++ b ++ cp ++
ogr ++ t1 ++ p ++ cgr ++ s))) (w ++ op ++ b ++ cp ++ ogr ++ t1 ++ p ++ cgr ++ s)
                                          else
                                            parserReturn env (w ++ op ++ b ++ cp ++ ogr ++ t1 ++ p ++ cgr ++
s)
```

The while-do statement is like the do-while and we first note that condition b is not evaluated immediately but is only read (by parsebexpr). As in the case of the do-while statement this is necessary because the condition will have to be evaluated at each iteration. If this were evaluated immediately almost certainly ending in an infinite loop.
But in this case the block of instructions p it's initially read (by parseprogram), and subsequently executed.
When b is True these two lines of code are executed:

```
parserReturn (getEnv (parse program envw (d ++ opg ++ p ++ cpg ++ w ++ op ++ b ++ cp ++ s))) (d
++ opg ++ p ++ cpg ++ w ++ op ++ b ++ cp ++ s)
```

The first executes the block of instructions p contained in the while, and modifies the environment; the second iterates the loop passing as input a new while with the same condition b and the same block of instructions p.

Otherwise, when b is False the program is only parsed syntactically, and it is not evaluated:

```
parserReturn env (d ++ opg ++ p ++ cpg ++ w ++ op ++ b ++ cp ++ s)
```

For

```
forCommand :: Parser String
forCommand = forKeyword >>>= \env f ->
              variable >>>= \env v ->
                timesKeyword >>>= \env t ->
                    openPargraf >>>= \env op ->
                      parseprogram >>>= \env p ->
                        repeatNTimes env p v >>>= \env r ->
                          closePargraf >>>= \env cp ->
                            semicolon >>>= \env s ->
                            parserReturn env (f ++ v ++ t ++ op ++ r ++ cp ++ s)
```

In the for statement we note that LIMPID first evaluates the value of the variable that represents the number of iterations of the program that will be execute, then parses the program (parseprogram) without its evaluation and computes the function **repeatNTimes**.

```
repeatNTimes :: Env -> String -> String  -> Parser String
repeatNTimes env p v = if bind env v <= "0" then
                              parserReturn env p
                          else
                            parserReturn env (v ++ ":=" ++ v ++ "-" ++ "1" ++ ";") >>>= \env dec-
>  parserReturn  (getEnv (parse program env dec)) dec  >>>= \envdec _ ->
                                  parserReturn (getEnv (parse program envdec p)) p >>>= \envf _ -
>
                                    repeatNTimes envf p v
```

RepeatNTimes accepts as parameters the env with the variable for the iteration evaluated and the program parsed but not executed.
So, if there are not iteration to be executed (the variable is zero or a negative number), then LIMPID leaves the program only parsed and not evaluated.
Otherwise, the variable is decremented, and the program is executed at each step of the iteration.
So, the environment obtained is passed to the recursive call of repeatNTimes that iterates until it reachs the value 0 and goes in the Then branch.

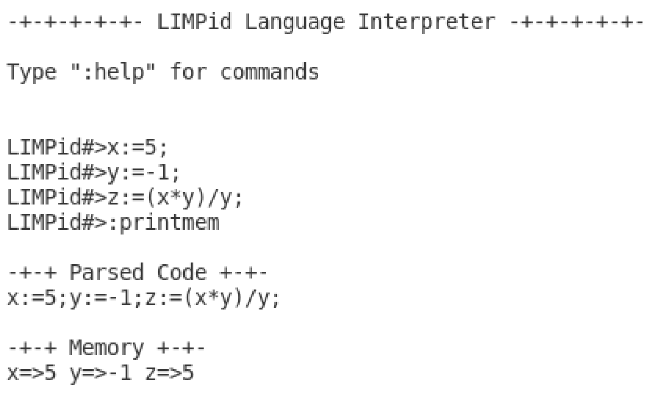
Execution LIMPID

LIMPID has a command-line interface in which you can write your own program instruction by instruction and check its execution, that is the state of the memory (the environment) at any time, with the command: mem.
For each line of code written, LIMPID will first execute a syntactic check, in fact calling only the ***parseprogram*** function, and in case of success it will be subsequently executed call; in case of a syntactic error, an error message will be returned.

Available commands are listed below:

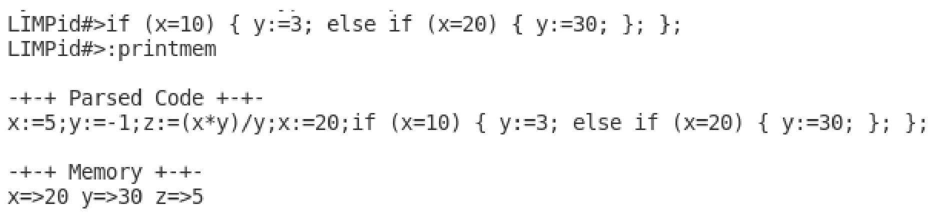
1. **: printmem** prints on screen the source code entered and parsed, and the memory status
2. **: syntax** prints the grammar of the language on the screen
3. **: help** prints the command help on the screen
4. **: quit** stops the application

Tests cases



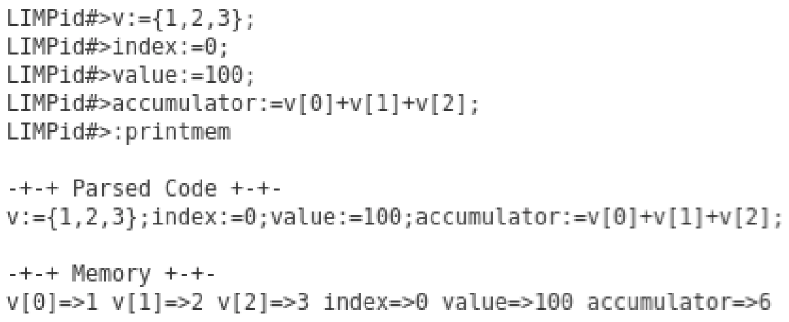
```
-+-+-+-+-+- LIMPid Language Interpreter -+-+-+-+-+-

Type ":help" for commands


LIMPid#>x:=5;
LIMPid#>y:=-1;
LIMPid#>z:=(x*y)/y;
LIMPid#>:printmem

-+-+ Parsed Code +-+-
x:=5;y:=-1;z:=(x*y)/y;

-+-+ Memory +-+-
x=>5 y=>-1 z=>5
```

Figure 1 Integer assignment and evaluation of arithmetic expr.



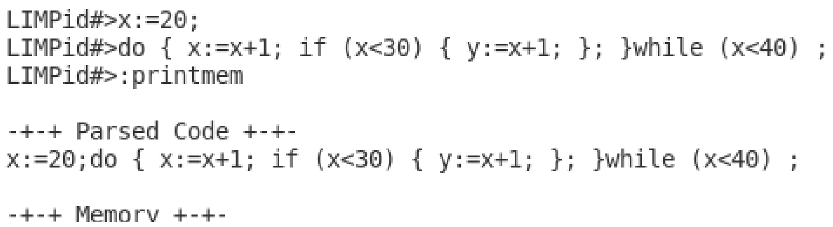
```
LIMPid#>if (x=10) { y:=3; else if (x=20) { y:=30; }; };
LIMPid#>:printmem

-+-+ Parsed Code +-+-
x:=5;y:=-1;z:=(x*y)/y;x:=20;if (x=10) { y:=3; else if (x=20) { y:=30; }; };

-+-+ Memory +-+-
x=>20 y=>30 z=>5
```

Figure 2 If-then-else with nested if-then-else in the false branch



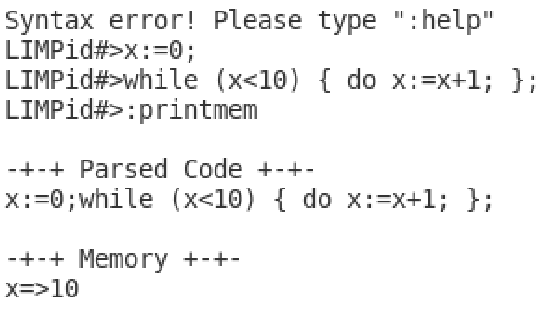
```
LIMPid#>v:={1,2,3};
LIMPid#>index:=0;
LIMPid#>value:=100;
LIMPid#>accumulator:=v[0]+v[1]+v[2];
LIMPid#>:printmem

-+-+ Parsed Code +-+-
v:={1,2,3};index:=0;value:=100;accumulator:=v[0]+v[1]+v[2];

-+-+ Memory +-+-
v[0]=>1 v[1]=>2 v[2]=>3 index=>0 value=>100 accumulator=>6
```

Figure 3 Other stuff with array



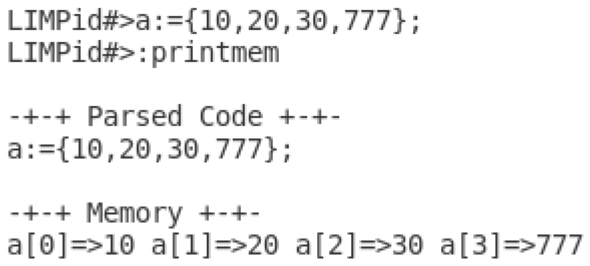
```
LIMPid#>x:=20;
LIMPid#>do { x:=x+1; if (x<30) { y:=x+1; }; }while (x<40) ;
LIMPid#>:printmem

-+-+ Parsed Code +-+-
x:=20;do { x:=x+1; if (x<30) { y:=x+1; }; }while (x<40) ;

-+-+ Memory +-+-
```

Figure 4 Do-while with nested if-then-else



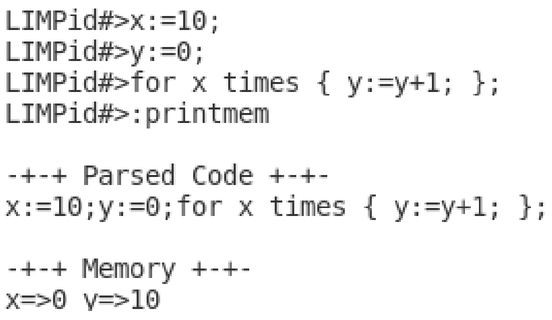
```
Syntax error! Please type ":help"
LIMPid#>x:=0;
LIMPid#>while (x<10) { do x:=x+1; };
LIMPid#>:printmem

-+-+ Parsed Code +-+-
x:=0;while (x<10) { do x:=x+1; };

-+-+ Memory +-+-
x=>10
```

Figure 5 While-Do



```
LIMPid#>a:={10,20,30,777};
LIMPid#>:printmem

-+-+ Parsed Code +-+-
a:={10,20,30,777};

-+-+ Memory +-+-
a[0]=>10 a[1]=>20 a[2]=>30 a[3]=>777
```

Figure 6 Array declaration



```
LIMPid#>x:=10;
LIMPid#>y:=0;
LIMPid#>for x times { y:=y+1; };
LIMPid#>:printmem

-+-+ Parsed Code +-+-
x:=10;y:=0;for x times { y:=y+1; };

-+-+ Memory +-+-
x=>0 y=>10
```

Figure 7 For-Times loop