Loop

Μετατροπή απλοποιημένης έκδοσης της Pascal σε Loop και διερμηνέας για την Loop, μία γλώσσα ισοδύναμη με τις πρωταρχικά αναδρομικές συναρτήσεις.

A reduced version of Pascal to Loop and an interpreter for Loop, a language equivalent to primitive recursive functions.

Loop Syntax

```
<loop_program> ::= <assignment> ;<loop_program>| <for_loop> <loop_program> | ε
<assignment> ::= <var> := 0 | <var> := 1 | <var> := <var> | <var> := <var> + 1 |
<var> := <var> - 1

<assignment> ::= <var> := <id>(<variable_list>)
<variable_list> ::= <var> , <variable_list> | <var>
<for_loop> ::= for <var> := 0 to <var> do <loop_program> done
<for_loop> ::= for <var> := 1 to <var> do <loop_program> done
```

Comments are lines that start with "--"

Θέματα Σημασιολογίας για τον διερμηνέα της Loop

- Τα προγράμματα που θέλουν input χρησιμοποιούν τις μεταβλητές i1, i2, i3, ...
- Τα προγράμματα που θέλουν να παράξουν αποτέλεσμα γράφουν στην μεταβλητή o1
- Προγράμματα που χρησιμοποιούν άλλα προγράμματα πχ x := add(x, y) πρέπει να βάλουν επιπλέον όρισμα στον interpreter --lib add.loop με το ορισμό του προγράμματος που κλήθηκε.

- Programs that need input use the variables i1, i2, i3, ...
- Programs that write output write to the variable o1
- Programs that get have a definition of the type x := add(x, y) need to provide --lib add.loop with the definition of the add program of the function exactly

Pascal Reduced Syntax for translating to loop

```
    <expr> ::= <simple expression> | <simple expression> <relational operation>
<simple expression>

    <simple expression> ::= <term> | <simple expression> <adding operator> <term>

    <relational operator> ::= = | <> | < | > | <= | >=
    <adding operator> ::= + | -
```

```
    <sign> ::= + | -

    <term> ::= <factor> | <term> <multiplying operator> <factor>

    <multiplying operator> ::= * | div | mod

    <factor> ::= <variable> | <unsigned constant> | ( <expression> )
    <unsigned constant> ::= <digit> <unsigned constant> | ε

    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

    <statement> ::= <assignment statement> | <structured statement>

    <assignment statement> ::= <variable> := <expression>

    <variable> ::= <identifier> | <identifer> [ <expression> ]

    <structured statement> ::= <compound statement> | <if statement> | <case
element> | <for statement>

    <compound statement> ::= begin <statement> {; <statement>} end;

    <if statement> ::= if <expr> then <statement> | if <expr> then <statement>
else <statement>

    <case element> ::= case <expression> of <case list elem> end

    <case list elem> ::= <unsigned constant> : <statement> <case list elem> | ε


    <for statement> ::= for <identifier> := <expression> to <expression> do
<statement>

    <program> ::= program <identifier> <var definitions> begin <statement> {;
<statement>} end.

    <var definitions> ::= var <variable declaration> {; <variable declaration>} |
ε

    <var definition> ::= <identifier> {, <identifier>} : <type>
    <type> ::= integer | array[<range>] of <integer>
    <range> ::= <unsigned constant> ... <unsigned constant>
```

Η διαδικασία της μετατροπής

Θα δείξουμε πως γίνεται η μετατροπή από την περιρισμένη Pascal σε Loop προγράμματα.

Η ισοδυναμία των δύο βασίζεται στο γεγονός
ότι ένα **expression** μπορεί να υπολογιστεί διατρέχοντας το συντακτικό του δέντρο και αποθηκεύοντας προσωρινά δεδομένα σε ξεχωριστές μεταβλητές.

Για παράδειγμα το Plus(Constant 3, Constant 4) θα γίνει

```
x1 := 1;
x1 := x1 + 1;
x1 := x1 + 1;
x2 := 1;
x2 := x2 + 1;
x2 := x2 + 1;
x2 := x2 + 1;
x3 := add(x1, x2)
```

και το αποτέλεσμα του expression βρίσκεται στην μεταβλητή x3.

Οι σταθερές μπορούν να υπολογιστούν και αποδοτικότερα
με διαδοχικούς διπλασιασμούς ή και πολλαπλασιασμούς μικρότερων σταθερών αλλά θα παραμείνουν έτσι λόγω απλότητας.

Τα **if statements** μπορούν να γίνουν ισοδύναμα στην loop θεωρώντας τις τιμές bool ως αριθμούς στο {0, 1}.
Υπολογίζουμε το expression μέσα στο if (εδώ res) και κάνουμε

```
temp := ifnzero(res);
for w := 1 to temp do
    ...body...
done
```

Τα **case statements** μπορούν επίσης να μοντελοποιηθούν μεσω διαδοχικών **if statements** και άρα δεν διαφέρουν από τα **if statements**.

Τέλος το **for statement** είναι ισοδύναμο στην loop αν βρεθεί το εύρος στο οποίο θα γίνει η επανάληψη (δηλαδή το κάτω και το πάνω όριο αποτελούν **expressions**), τοποθετηθεί το κάτω εύρος σε μία μεταβλητή και κάθε φορά προστίθεται σε αυτήν το *loop variable* ώστε να έχουμε αυτή ως τον πραγματικό *loop counter*.

Για παράδειγμα το

```
for w := 3 to 2*2 do
...;
```

θα γίνει

```
-- Calculate (3)
x0 := 0;
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
-- Calculate 2*2
x1 := 0;
x1 := x1 + 1;
x1 := x1 + 1;
x2 := 0;
x2 := x2 + 1;
x2 := x2 + 1;
x3 := mult(x1, x2);

-- calculate the difference from upper bound to lower bound
x4 := sub(x3, x1);
x4 := x4 + 1;

-- w := 3;
w := x0;
for x5 := 1 to x4 do
    ...
    -- fix w to represent current counter
    w := w + 1;
done
```

Στην μετατροπή δεν θα χρησιμοποιηθούν αρνητικοί αριθμοί παρότι αυτοί μπορούν να κωδικοποιηθούν από τους θετικούς ή να χρησιμοποιηθεί για κάθε xi μία μεταβλητή xip με τιμές στο {0, 1} που καθορίζει το πρόσημο της xi.
Επιπλέον στην μετατροπή θα χρησιμοποιηθούν identifier strings για τις μεταβλητές της loop. Αυτό δεν έχει διαφορά με την χρήση μόνο xi αφού όλες οι identifier strings μπορούν και αυτές να κωδικοποιηθούν από φυσικούς αριθμούς.

Για τα **arrays** της Pascal θα δημιουργήσουμε μία μεταβλητή για κάθε index του array. Πρόσβαση στα arrays θα κάνουμε υπολογίζοντας το εσωτερικό του [] και στην συνέχεια ένα μεγάλο case όπου για κάθε πιθανό index χρησιμοποιείται η σωστή μεταβλητή. Μπορούσαμε και να κωδικοποιήσουμε το array σε μία μεταβλητή χρησιμοποιώντας δυνάμεις πρώτων.

Παράδειγμα μετατροπής όπως παράχθηκε από το πρόγραμμα

`hello.pas`

```
program hello;

var
    x, y : integer;
    z : array[0...10] of integer;
    w : integer;
begin
    x := 12;
    y := x div 2;
    z[x-y] := x + y * 4;
    for w := 1 to 3*3 do
    begin
        x := x * 2;
    end;
    if x > 10 then
        x := 9;
end.
```

Το παραγόμενο loop πρόγραμμα

Παρατηρούμε ακολουθία των 10 ifs που αντιπροσωπεύει τις 10 δυνατές τιμές που μπορεί να γίνει η ανάθεση λόγω index του array z

`hello.loop`

```
w := 0;
x := 0;
y := 0;
z0 := 0;
z1 := 0;
z2 := 0;
z3 := 0;
z4 := 0;
z5 := 0;
z6 := 0;
z7 := 0;
z8 := 0;
z9 := 0;
z10 := 0;
x0 := 0;
x0 := x0 + 1;
```

```
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
x0 := x0 + 1;
x := x0;
x := x;
x1 := 0;
x1 := x1 + 1;
x1 := x1 + 1;
x2 := div(x, x1);
y := x2;
x := x;
y := y;
x3 := 0;
x3 := x3 + 1;
x3 := x3 + 1;
x3 := x3 + 1;
x3 := x3 + 1;
x4 := mult(y, x3);
x5 := add(x, x4);
x := x;
y := y;
x6 := sub(x, y);
x7 := 0;
x6 := x6;
x8 := equals(x7, x6);
x9 := ifnzero(x8);
for x10 := 1 to x9 do
    x5 := x5;
    z0 := x5;
done
x11 := 0;
x11 := x11 + 1;
x6 := x6;
x12 := equals(x11, x6);
x13 := ifnzero(x12);
for x14 := 1 to x13 do
    x5 := x5;
    z1 := x5;
done
x15 := 0;
x15 := x15 + 1;
x15 := x15 + 1;
x6 := x6;
x16 := equals(x15, x6);
x17 := ifnzero(x16);
for x18 := 1 to x17 do
```

```
    x5 := x5;
    z2 := x5;
done
x19 := 0;
x19 := x19 + 1;
x19 := x19 + 1;
x19 := x19 + 1;
x6 := x6;
x20 := equals(x19, x6);
x21 := ifnzero(x20);
for x22 := 1 to x21 do
    x5 := x5;
    z3 := x5;
done
x23 := 0;
x23 := x23 + 1;
x23 := x23 + 1;
x23 := x23 + 1;
x23 := x23 + 1;
x6 := x6;
x24 := equals(x23, x6);
x25 := ifnzero(x24);
for x26 := 1 to x25 do
    x5 := x5;
    z4 := x5;
done
x27 := 0;
x27 := x27 + 1;
x27 := x27 + 1;
x27 := x27 + 1;
x27 := x27 + 1;
x27 := x27 + 1;
x6 := x6;
x28 := equals(x27, x6);
x29 := ifnzero(x28);
for x30 := 1 to x29 do
    x5 := x5;
    z5 := x5;
done
x31 := 0;
x31 := x31 + 1;
x31 := x31 + 1;
x31 := x31 + 1;
x31 := x31 + 1;
x31 := x31 + 1;
x31 := x31 + 1;
x6 := x6;
x32 := equals(x31, x6);
x33 := ifnzero(x32);
for x34 := 1 to x33 do
    x5 := x5;
    z6 := x5;
done
x35 := 0;
```

```
x35 := x35 + 1;
x35 := x35 + 1;
x35 := x35 + 1;
x35 := x35 + 1;
x35 := x35 + 1;
x35 := x35 + 1;
x35 := x35 + 1;
x6 := x6;
x36 := equals(x35, x6);
x37 := ifnzero(x36);
for x38 := 1 to x37 do
    x5 := x5;
    z7 := x5;
done
x39 := 0;
x39 := x39 + 1;
x39 := x39 + 1;
x39 := x39 + 1;
x39 := x39 + 1;
x39 := x39 + 1;
x39 := x39 + 1;
x39 := x39 + 1;
x39 := x39 + 1;
x6 := x6;
x40 := equals(x39, x6);
x41 := ifnzero(x40);
for x42 := 1 to x41 do
    x5 := x5;
    z8 := x5;
done
x43 := 0;
x43 := x43 + 1;
x43 := x43 + 1;
x43 := x43 + 1;
x43 := x43 + 1;
x43 := x43 + 1;
x43 := x43 + 1;
x43 := x43 + 1;
x43 := x43 + 1;
x43 := x43 + 1;
x6 := x6;
x44 := equals(x43, x6);
x45 := ifnzero(x44);
for x46 := 1 to x45 do
    x5 := x5;
    z9 := x5;
done
x47 := 0;
x47 := x47 + 1;
x47 := x47 + 1;
x47 := x47 + 1;
x47 := x47 + 1;
x47 := x47 + 1;
x47 := x47 + 1;
```

```
x47 := x47 + 1;
x47 := x47 + 1;
x47 := x47 + 1;
x47 := x47 + 1;
x6 := x6;
x48 := equals(x47, x6);
x49 := ifnzero(x48);
for x50 := 1 to x49 do
    x5 := x5;
    z10 := x5;
done
x51 := 0;
x51 := x51 + 1;
x52 := 0;
x52 := x52 + 1;
x52 := x52 + 1;
x52 := x52 + 1;
x53 := 0;
x53 := x53 + 1;
x53 := x53 + 1;
x53 := x53 + 1;
x54 := mult(x52, x53);
x55 := sub(x54, x51);
x55 := x55 + 1;
w := x51;
for x56 := 1 to x55 do
    x := x;
    x57 := 0;
    x57 := x57 + 1;
    x57 := x57 + 1;
    x58 := mult(x, x57);
    x := x58;
    w := w + 1;
done
w := w - 1;
x := x;
x59 := 0;
x59 := x59 + 1;
x59 := x59 + 1;
x59 := x59 + 1;
x59 := x59 + 1;
x59 := x59 + 1;
x59 := x59 + 1;
x59 := x59 + 1;
x59 := x59 + 1;
x59 := x59 + 1;
x59 := x59 + 1;
x60 := greater(x, x59);
x61 := ifnzero(x60);
for x62 := 1 to x61 do
    x63 := 0;
    x63 := x63 + 1;
    x63 := x63 + 1;
    x63 := x63 + 1;
```

```
    x63 := x63 + 1;
    x63 := x63 + 1;
    x63 := x63 + 1;
    x63 := x63 + 1;
    x63 := x63 + 1;
    x63 := x63 + 1;
    x := x63;
done
```

Υλοποίηση σε Haskell

Parser and Interpreter for Loop

```haskell
module Loop where

import System.IO
import System.Environment
import System.Exit
import System.Path
import Data.Either
import Data.Maybe
import Data.Tree
import Data.List
import Control.Monad
import Text.ParserCombinators.Parsec
import qualified Data.Map as Map

-- Loop AST for Parsing
type Variable = String
data AssignmentType = ToVariable Variable
                    | ToZero
                    | ToOne
                    | ToSucc Variable
                    | ToPred Variable
                    | ToProgram String [Variable]
                    deriving (Eq, Show)

data LoopProgram = Assignment Variable AssignmentType
                 | ForLoop (Variable, AssignmentType) Variable LoopProgram
                 | Concatenation LoopProgram LoopProgram
                 deriving (Eq)

toDataTree (Assignment v t) = Node ("Assignment " ++ show v ++ " := " ++ show t)
[]
toDataTree (ForLoop (v, t) end program) = Node ("ForLoop " ++ show v ++ " := " ++
show t ++ "to " ++ show end) [toDataTree program]
toDataTree (Concatenation l r) = Node "" [toDataTree l, toDataTree r]
instance Show LoopProgram where
    -- show x = (drawTree . toDataTree) x ++ "\n\n Printed Program :\n" ++
toProgramStructure [] x
    show = toProgramStructure []

-- PARSER COMBINATORS FOR LOOP PROGRAMS
wsfnb :: GenParser Char st a -> GenParser Char st a
wsfnb t = do
    many space
    res <- t
    many space
    return res
```

```haskell
semicolon :: GenParser Char st Char
semicolon = wsfnb $ char ';'

variable :: GenParser Char st Variable
variable = wsfnb $ do
    x <- letter
    xs <- many digit
    return (x:xs)

assign :: GenParser Char st String
assign = wsfnb $ do
    char ':'
    char '='
    return ":="

assignment :: GenParser Char st LoopProgram
assignment = try oneAssignment
             <|> try zeroAssignment
             <|> try varAssignment
             <|> try succAssignment
             <|> try predAssignment
             <|> programAssignment

oneAssignment :: GenParser Char st LoopProgram
oneAssignment = wsfnb $ do
    target <- variable
    assign
    wsfnb $ char '1'
    semicolon
    return $ Assignment target ToOne

zeroAssignment :: GenParser Char st LoopProgram
zeroAssignment = wsfnb $ do
    target <- variable
    assign
    wsfnb $ char '0'
    semicolon
    return $ Assignment target ToZero

varAssignment :: GenParser Char st LoopProgram
varAssignment = wsfnb $ do
    target <- variable
    assign
    op <- variable
    semicolon
    return $ Assignment target (ToVariable op)

succAssignment :: GenParser Char st LoopProgram
succAssignment = wsfnb $ do
    target <- variable
    assign
    op <- variable
    wsfnb $ char '+'
```

```
    wsfnb $ char '1'
    semicolon
    return $ Assignment target (ToSucc op)

predAssignment :: GenParser Char st LoopProgram
predAssignment = wsfnb $ do
    target <- variable
    assign
    op <- variable
    wsfnb $ char '-'
    wsfnb $ char '1'
    semicolon
    return $ Assignment target (ToPred op)

programAssignment :: GenParser Char st LoopProgram
programAssignment = wsfnb $ do
    target <- variable
    assign
    idbod <- many1 letter
    idtail <- many digit
    let iden = idbod ++ idtail
    wsfnb $ char '('
    vars <- variableList
    wsfnb $ char ')'
    semicolon
    return $ Assignment target (ToProgram iden vars)

-- variable list need to parse "x, y, z, w"
variableList :: GenParser Char st [Variable]
variableList = (try $ wsfnb $  do
    v <- variable
    wsfnb $ char ','
    vs <- variableList
    return (v:vs))
    <|> (do
        v <- variable
        return [v])

number :: GenParser Char st Int
number = wsfnb $ do
    digits <- many1 digit
    return (read digits)

forSchema :: GenParser Char st LoopProgram
forSchema = wsfnb $ do
    wsfnb $ string "for"
    startVar <- variable
    assign
    c <- (try $ wsfnb $ char '0') <|> (wsfnb $ char '1')
    let ass = case c of
                  '0' -> ToZero
                  _   -> ToOne
    wsfnb $ string "to"
    endVar <- variable
```

```
    wsfnb $ string "do"
    p <- loopProgram
    wsfnb $ string "done"
    return $ ForLoop (startVar, ass) endVar p

loopProgram :: GenParser Char st LoopProgram
loopProgram = wsfnb $ do
    p:ps <- many1 (try assignment <|> forSchema)

    return $ foldl Concatenation p ps




-- The actuall interpretation of a compiled LoopProgram
--
-- With the program we have a map<libraryProgram, Syntax of the program>
-- and the result of the computation is map<string, Int> the final values
-- of all variables after calculation
interpret :: LoopProgram -> Map.Map String LoopProgram -> Map.Map Variable Int ->
(Map.Map Variable Int)
-- Change the value of v to the new one according to the assignment
-- if a function call is in the result to add
-- ---> Stop calculation
-- ---> Interpret the library function with arguments those of the function call
interpret a@(Assignment v t) libs prevMap =
    case t of
        ToVariable v' -> valInsert v v' id prevMap
        ToZero        -> Map.insert v 0 prevMap
        ToOne         -> Map.insert v 1 prevMap
        ToSucc v'     -> valInsert v v' (+1) prevMap
        ToPred v'     -> valInsert v v' (\x -> if x == 0 then 0 else x - 1)
prevMap
        ToProgram iden vars -> Map.insert v (fromJust $ Map.lookup "o1"
(independentProgram iden vars)) prevMap
    where
        independentProgram :: String -> [Variable] -> Map.Map Variable Int
        independentProgram iden vars =
            let
                parsedP = Map.lookup iden libs
                func :: Map.Map Variable Int -> (Int, Variable) -> Map.Map
Variable Int
                func prev (index, curr) =
                    Map.insert ("i" ++ show index) (fromJust $ Map.lookup curr
prevMap) prev
            in
                case isNothing parsedP of
                    True -> error ("no " ++ show iden ++ " program in the
library\n all are:\n" ++ show (Map.keys libs))
                    _    -> interpret (fromJust parsedP) libs (foldl func
Map.empty (zip [1..] vars))

        valInsert :: Variable -> Variable -> (Int -> Int) -> Map.Map Variable Int
-> Map.Map Variable Int
```

```
        valInsert v v' f prevMap =
                if v' `Map.notMember` prevMap then error ("unknwon variable " ++
v' ++ " in " ++ show a) else Map.insert v (f $ fromJust $ Map.lookup v' prevMap)
prevMap

-- iterate the interpretation of the body from startval to endval
interpret a@(ForLoop (startvar, startval) endvar innerProgram) libs prevMap =
    let
        m' = interpret (Assignment startvar startval) libs prevMap

        loop :: Variable -> Variable -> Map.Map Variable Int -> Map.Map Variable
Int
        loop start end currMap =
            let
                res1 = Map.lookup start currMap
                res2 = Map.lookup end currMap
            in
                case (res1,res2) of
                    (Nothing, _)     -> error ("you gave me a variable in a loop
that does not exist :" ++ show start)
                    (_, Nothing) -> error ("you gave me a variable in a loop that
does not exist :" ++ show end)
                    _             -> if (fromJust res1) > (fromJust res2)
                                        then currMap
                                        else (loop start end . interpret (Assignment
start (ToSucc start)) libs . interpret innerProgram libs ) currMap

    in
        loop startvar endvar m'

-- Interpret left then interpret right
interpret a@(Concatenation l r) libs prevMap = (interpret r libs . interpret l
libs) prevMap



-- New stuff for the compilation of a program targeting Loop so it can be printed
-- Directly into loop code from the AST of LoopProgram

toProgramStructure' (ToVariable v) = v
toProgramStructure' (ToZero) = "0"
toProgramStructure' (ToOne) = "1"
toProgramStructure' (ToSucc v) = v ++ " + 1"
toProgramStructure' (ToPred v) = v ++ " - 1"
toProgramStructure' (ToProgram p vars) = p ++ "(" ++ intercalate ", " vars ++ ")"

toProgramStructure indent (Assignment v t) = indent ++ v ++ " := " ++
toProgramStructure' t ++ ";\n"
toProgramStructure indent (ForLoop (v, t) end inner) =
    indent ++ "for " ++ v ++ " := " ++ toProgramStructure' t  ++ " to " ++ end ++
" do\n" ++
    (toProgramStructure ("    " ++ indent) inner) ++
    indent ++ "done\n"
toProgramStructure indent (Concatenation l r) =
```

```
      (toProgramStructure indent l) ++
      (toProgramStructure indent r)
```

Parser and Semantic Analyser of Pascal

```
module ReducedPascal where

import System.IO
import Control.Monad
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Expr
import Text.ParserCombinators.Parsec.Language
import qualified Text.ParserCombinators.Parsec.Token as Token
import qualified Data.Map as Map
import Data.Maybe

-- Datatypes for definining the AST of Reduced Pascal

data Variable = Variable String
              | ArrayIndexedAt String Expression
              deriving (Eq, Show)
data Expression = Equals Expression Expression
                | Different Expression Expression
                | Less Expression Expression
                | LessEq Expression Expression
                | Greater Expression Expression
                | GreaterEq Expression Expression
                | Plus Expression Expression
                | Minus Expression Expression
                | Mult Expression Expression
                | Div Expression Expression
                | Mod Expression Expression
                | Constant Integer
                | Var Variable
                deriving (Eq, Show)

-- This is the datatype to hold a statement
-- A program is basically a statement but it needs to
-- have the variable definition space handled
data Statement = Block [Statement]
               | Assignment Variable Expression
               | If Expression Statement
               | IfElse Expression Statement Statement
               | Case Expression [(Integer, Statement)]
               | For Variable (Expression, Expression) Statement
               -- in the for loop (from-lowerBound, to-UpperBound)
               deriving (Eq, Show)

-- The only two types of variables accepted in this
-- version of pascal
data PascalType = IntegerP
                | ArrayP (Integer, Integer) -- it should have a second argument
here for the types of the variables in the array but is not needed since we only
have Integers and will only allow arrays[integers]
```

```haskell
                   deriving (Eq, Show)

data VarDecl = IntegerVar String
               | ArrayVar String (Integer, Integer)
               deriving (Eq, Show)

-- The definition for the whole program
data PascalProgram = Program {
    name      :: String,      -- Name of the program
    variables :: [VarDecl],   -- The Variables declared at the start
    body      :: [Statement]  -- The main body (begin ... end.)
} deriving (Eq, Show)

type SymbolTable = Map.Map String PascalType


languageDef =
    emptyDef {
        Token.commentStart   = "(*",
        Token.commentEnd     = "*)",
        Token.commentLine    = "//",
        Token.nestedComments = True,
        Token.identStart      = letter,
        Token.identLetter      = alphaNum,
        Token.reservedNames  = [
            "begin",
            "end",
            "if",
            "then",
            "else",
            "case",
            "of",
            "for",
            "to",
            "do",
            "program",
            "var",
            "integer",
            "array"
        ],
        Token.reservedOpNames = [
            "=",
            "<>",
            "<",
            ">",
            "<=",
            ">=",
            "+",
            "-",
            "*",
            "div",
            "mod",
            ":=",
            "[",
```

```
            "]",
            ",",
            ":",
            ".",
            "..."
        ]
    }

lexer = Token.makeTokenParser languageDef

-- All the individual parsers for parsing lexemes
identifier          = Token.identifier     lexer
reserved            = Token.reserved       lexer
reservedOp          = Token.reservedOp     lexer
parens              = Token.parens         lexer
natural             = Token.natural        lexer
semicolon           = Token.semi           lexer
whiteSpace          = Token.whiteSpace     lexer

-- expression parser
expression :: Parser Expression
expression = buildExpressionParser table term
             <?> "expression"

-- base case of expression parser
term :: Parser Expression
term = try (parens expression)
       <|> liftM Constant natural
       <|> try (do {id <- identifier; reservedOp "["; p <- expression ; reservedOp
"]"; return $ Var (ArrayIndexedAt id p);})
       <|> liftM (Var . Variable) identifier
       <?> "simple term failed!"

table = [
            [binary "*" Mult AssocLeft, binary "div" Div AssocLeft, binary "mod"
Mod AssocLeft],
            [binary "+" Plus AssocLeft, binary "-" Minus AssocLeft],
            [binary "=" Equals AssocNone, binary "<>" Different AssocNone, binary
"<" Less AssocNone, binary ">" Greater AssocNone, binary "<=" LessEq AssocNone,
binary ">=" GreaterEq AssocNone ]
    ]

binary name fun assoc = Infix (do{ reservedOp name; return fun }) assoc


-- Now the parsers for Statements to build Statement Syntax Tree
statement :: Parser Statement
statement =     try block
            <|> try assignment
            <|> try ifElse
            <|> try if'
            <|> try case'
            <|> for
            <?> "Probably Something Wrong with opening Statement Here"
```

```haskell
block :: Parser Statement
block = do
        reserved "begin"
        s <- endBy1 statement semicolon
        reserved "end"
        return $ Block s

assignment :: Parser Statement
assignment = do
             v <- variable
             reservedOp ":="
             e <- expression
             return $ Assignment v e
             <?> "Failed to parse Assignment!"

variable :: Parser Variable
variable = try (do {id <- identifier; reservedOp "["; p <- expression ; reservedOp
"]"; return (ArrayIndexedAt id p);})
              <|> liftM Variable identifier
           <?> "failed to parse variable!"

ifElse :: Parser Statement
ifElse = do
         reserved "if"
         condition <- expression
         reserved "then"
         ifpart <- statement
         reserved "else"
         elsepart <- statement
         return $ IfElse condition ifpart elsepart

if' :: Parser Statement
if' = do
      reserved "if"
      condition <- expression
      reserved "then"
      ifpart <- statement
      return $ If condition ifpart

case' :: Parser Statement
case' = do
        reserved "case"
        arg <- expression
        reserved "of"
        l <- sepBy1 caseListElem semicolon
        reserved "end"
        return $ Case arg l

caseListElem :: Parser (Integer, Statement)
caseListElem = do
               n <- natural
               reservedOp ":"
               body <- statement
```

```
                  return (n, body)

for :: Parser Statement
for = do
        reserved "for"
        Assignment loopVariable lowerBound <- assignment
        reserved "to"
        upperBound <- expression
        reserved "do"
        body <- statement
        return $ For loopVariable (lowerBound, upperBound) body



program :: Parser PascalProgram
program = do
            reserved "program"
            n <- identifier
            semicolon
            reserved "var"
            vs <- endBy1 varDef semicolon
            -- now for the main body
            Block ss <- block
            reservedOp "."
            return Program {
                name = n,
                variables = concat vs,
                body = ss
            }


varDef :: Parser [VarDecl]
varDef = do
           ids <- sepBy1 identifier (reservedOp ",")
           reservedOp ":"
           func <- type'
           return $ map func ids

-- This just gathers which type it is and returns
-- the correct constructor to apply to the names
type' :: Parser (String -> VarDecl)
type' =     (try $ reserved "integer" >> return IntegerVar)
        <|> do
             reserved "array"
             reservedOp "["
             from <- natural
             reservedOp "..."
             to <- natural
             reservedOp "]"
             reserved "of"
             reserved "integer"
             return $ flip ArrayVar (from, to)
```

```
sem :: PascalProgram -> Either SymbolTable String
sem p =
    let
        st = getSymbolTable p
        programStatements = body p
    in
        case sem' (Right (Block programStatements)) st of
            Nothing  -> Left st
            Just err -> Right err

getSymbolTable :: PascalProgram -> SymbolTable
getSymbolTable p = foldl func Map.empty (variables p)
    where
        func prev (IntegerVar v) | isNothing $ Map.lookup v prev = Map.insert v
IntegerP prev
                                 | otherwise                     = error $
varString v ++ "already defined"
        func prev (ArrayVar v bounds) | isNothing $ Map.lookup v prev = Map.insert
v (ArrayP bounds) prev
                                      | otherwise                     = error $
varString v ++ "already defined"


sem' :: Either Expression Statement -> SymbolTable -> Maybe String
sem' (Left (Constant c)) st = Nothing
sem' (Left (Var (Variable v))) st | v `Map.notMember` st = Just $ varString v ++
"is not previously defined"
                                  | otherwise            =
                                      case Map.lookup v st of
                                          Just (ArrayP _) -> Just $ varString v ++
"is defined as an array and cannot be used in expressions"
                                          Just (IntegerP) -> Nothing
                                          Nothing         -> Just "should not have
gotten here"

sem' (Left (Var (ArrayIndexedAt v e))) st =
    case Map.lookup v st of
        Just (ArrayP (_, _)) -> Nothing
        Just _               -> Just $ varString v ++ "is not an array to be
indexed"
        Nothing              -> Just $ varString v ++ "is not previously
defined"

sem' (Left (Equals l r))    st = helper [Left l, Left r] st
sem' (Left (Different l r)) st = helper [Left l, Left r] st
sem' (Left (Less l r))      st = helper [Left l, Left r] st
sem' (Left (LessEq l r))    st = helper [Left l, Left r] st
sem' (Left (Greater l r))   st = helper [Left l, Left r] st
sem' (Left (GreaterEq l r)) st = helper [Left l, Left r] st
sem' (Left (Plus l r))      st = helper [Left l, Left r] st
sem' (Left (Minus l r))     st = helper [Left l, Left r] st
sem' (Left (Mult l r))      st = helper [Left l, Left r] st
sem' (Left (Div l r))       st = helper [Left l, Left r] st
sem' (Left (Mod l r))       st = helper [Left l, Left r] st
```

```haskell
-- Starting to semantically analyse statements
sem' (Right (Block stms)) st = helper (map Right stms) st

sem' (Right (Assignment v e)) st = helper [Left $ Var v, Left e] st

sem' (Right (If e s)) st = helper [Left e, Right s] st
sem' (Right (IfElse e s s')) st = helper [Left e, Right s, Right s'] st
sem' (Right (Case e xs)) st = helper (Left e : map (Right . snd) xs) st
sem' (Right (For v (lbound, hbound) stmt)) st = helper [Right (Assignment v
lbound), Left hbound, Right stmt] st

helper l st =
    case (catMaybes . map (flip sem' st)) l of
        []    -> Nothing
        l   -> Just $ unlines l

varString v = "Variable '" ++ v ++ "' "

parseProgram :: IO String -> IO (Either (PascalProgram, SymbolTable) (Either
ParseError String) )
parseProgram inp = do
                   s <- inp
                   let p = parse program [] s
                   case p of
                       Left err -> return (Right $ Left err)
                       Right p  -> case sem p of
                                       Left st   -> return $ Left (p, st)
                                       Right err -> return $ Right $ Right err
```

Pascal To Loop Logic

```haskell
module RPtoLoop where

import qualified ReducedPascal as P
import qualified Loop as L
import qualified Data.Map as Map


-- Operators to help not writing a lot of stuff
-- ASSIGNMENT TO FUNCTION CALL
(<%) :: L.Variable -> (String, [L.Variable]) -> L.LoopProgram
var <% (fname, args) = L.Assignment var (L.ToProgram fname args)

-- ASSIGNMENT TO SOMETHING
(<==) :: L.Variable -> L.AssignmentType -> L.LoopProgram
var <== t = L.Assignment var t

-- INFIX CONCATENATION
(=>>) :: L.LoopProgram -> L.LoopProgram -> L.LoopProgram
l =>> r = L.Concatenation l r
infixl 0 =>>

-- GET THE NEW VARIABLE OF AN INTEGER
decode :: Show a => a -> String
decode x = ("x" ++ show x)

-- END OF OPERATORS

-- THIS DEFINES A DATATYPE FOR CALCULATION
-- CALCULATIONS CAN BE COMBINED AND APPLIED THUS
-- THE APPLICATIVE AND MONAD DEFINITIONS

-- Αυτό γίνεται γιατί κατά την διάρκεια της μετατροπής
-- ενός pascal expression χρειαζόμαστε ξεχωριστές μεταβλητές συνέχεια
-- για ενδοιάμεσα απότελέσματα
--
-- Το UniqueCalculation φορντίζει να περνιέται ένας ακέραιος μεταξύ των
-- calculation μέσω του οποίου δημιουργούνται νέες καινούργιες μεταβλητές

-- Το UniqueCalculation είναι Monad και από αυτό μπορεί να χρησιμοποιηθεί το
ειδικό
-- do-syntax της haskell για τον συνδυασμό μικρότερων calculations σε μεγαλύτερα
και πιο σύνθετα

-- A Pascal Program will be defined as a UniqueCalculation of a LoopProgram
newtype UniqueCaclulation a = Calculation {
    runCalculation :: Integer -> (a, Integer)
}

instance Functor (UniqueCaclulation) where
```

```
    fmap f (Calculation g) = Calculation $ \i ->
        let
            (res, next) = g i
        in
            (f res, next)

instance Applicative (UniqueCaclulation) where
    -- pure :: a -> f a
    pure x = Calculation $ \i -> (x, i)

    (Calculation f) <*> (Calculation p) = Calculation $ \i ->
        let
            (res, next) = f i
            (res', next') = p next
        in
            (res res', next')

instance Monad (UniqueCaclulation) where
    return = pure
    (Calculation p) >>= (f) =
        Calculation $ \i ->
            let
                (res, next) = p i
                Calculation g = f res
            in
                g next

-- Grab an Integer for creating a new unique variable and change the state to the
next Integer
next :: UniqueCaclulation Integer
next = Calculation $ \i -> (i, i+1)

-- Calculate the increment of a calculation
incCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable])  -> UniqueCaclulation
(L.LoopProgram, [L.Variable])
incCalc p = do
            (res, rest) <- p
            let f = head rest
            return $ (res =>> f <== L.ToSucc (f), [f])

-- Calculate 0
calcZero :: UniqueCaclulation (L.LoopProgram, [L.Variable])
calcZero = do
    var <- decode <$> next
    return (var <== L.ToZero, [var])

-- Calculate a constant n
constantCalc :: Integer -> UniqueCaclulation (L.LoopProgram, [L.Variable])
constantCalc 0 = calcZero
constantCalc n = incCalc (constantCalc (n-1))

-- Calculate a series of calculations
calculateAll :: [UniqueCaclulation (L.LoopProgram, [L.Variable])] ->
UniqueCaclulation (L.LoopProgram, [L.Variable])
```

```haskell
calculateAll [x] = x
calculateAll (x:xs) = do
    (res, vars) <- x
    (res', vars') <- calculateAll xs
    return (res =>> res', vars ++ vars')

-- Pass to a function results of a list of computations
funCallCalc :: String -> [UniqueCaclulation (L.LoopProgram, [L.Variable])] ->
UniqueCaclulation (L.LoopProgram, [L.Variable])
funCallCalc s vs = do
    (res, resvars) <- calculateAll vs
    var <- decode <$> next
    return (res =>> var <% (s, resvars), [var])


-- Caclualte if two calculations produce equal results
equalsCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable]) -> UniqueCaclulation (L.LoopProgram, [L.Variable])
equalsCalc l r = funCallCalc "equals" [l, r]

-- Caclualte if two calculations produce different results
differentCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) ->
UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable])
differentCalc l r = do
    (res, vars) <- equalsCalc l r
    var <- decode <$> next
    return (res =>> var <% ("ifnzero", vars), [var])

-- Caclualte if the first calculation is smaller than the second
lessCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable]) -> UniqueCaclulation (L.LoopProgram, [L.Variable])
lessCalc l r = funCallCalc "less" [l, r]

-- Caclualte if the first calculation is smaller or equals to the second
lessEqCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable]) -> UniqueCaclulation (L.LoopProgram, [L.Variable])
lessEqCalc l r = do
    (res, rvar) <- l
    (res', rvar') <- r
    let var = head rvar
    let var' = head rvar'
    var1 <- decode <$> next
    var2 <- decode <$> next
    var3 <- decode <$> next
    return (res =>> res' =>> var1 <% ("less", [var, var']) =>> var2 <% ("equals",
[var, var']) =>> var3 <% ("or", [var1, var2]), [var3])

-- Caclualte if the first calculation is greater than the second
greaterCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) ->
UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable])
greaterCalc l r = funCallCalc "greater" [l, r]
```

```
-- Caclualte if the first calculation is greater or equals to the second
greaterEqCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) ->
UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable])
greaterEqCalc l r = do
    (res, rvar) <- l
    (res', rvar') <- r
    let var = head rvar
    let var' = head rvar'
    var1 <- decode <$> next
    var2 <- decode <$> next
    var3 <- decode <$> next
    return (res =>> res' =>> var1 <% ("greater", [var, var']) =>> var2 <%
("equals", [var, var']) =>> var3 <% ("or", [var1, var2]), [var3])

-- Plus of two calculations
plusCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable]) -> UniqueCaclulation (L.LoopProgram, [L.Variable])
plusCalc l r = funCallCalc "add" [l, r]

-- Minus of two calculations ...
minusCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable]) -> UniqueCaclulation (L.LoopProgram, [L.Variable])
minusCalc l r = funCallCalc "sub" [l, r]

multCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable]) -> UniqueCaclulation (L.LoopProgram, [L.Variable])
multCalc l r = funCallCalc "mult" [l, r]

divCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable]) -> UniqueCaclulation (L.LoopProgram, [L.Variable])
divCalc l r = funCallCalc "div" [l, r]

modCalc :: UniqueCaclulation (L.LoopProgram, [L.Variable]) -> UniqueCaclulation
(L.LoopProgram, [L.Variable]) -> UniqueCaclulation (L.LoopProgram, [L.Variable])
modCalc l r = funCallCalc "mod" [l, r]


-- This generates
programCalc :: (P.PascalProgram, P.SymbolTable) -> UniqueCaclulation L.LoopProgram
programCalc (p, symtbl) = do
    let header = (foldl1 (=>>) . map f . Map.assocs) symtbl
    res <- (statementCalc . P.Block . P.body) p
    return (header =>> res)

    where
        f (v, P.IntegerP) = v <== L.ToZero
        f (n, P.ArrayP (low, high))  | low == high = f ((n ++ show low),
P.IntegerP)
                                     | otherwise   = f ((n ++ show low),
P.IntegerP) =>> f (n, P.ArrayP(low+1, high))

        expressionCalc :: P.Expression -> UniqueCaclulation (L.LoopProgram,
[L.Variable])
```

```
        expressionCalc (P.Equals l r) = equalsCalc (expressionCalc l)
(expressionCalc r)
        expressionCalc (P.Different l r) = differentCalc (expressionCalc l)
(expressionCalc r)
        expressionCalc (P.Less l r) = lessCalc (expressionCalc l) (expressionCalc
r)
        expressionCalc (P.LessEq l r) = lessEqCalc (expressionCalc l)
(expressionCalc r)
        expressionCalc (P.Greater l r) = greaterCalc (expressionCalc l)
(expressionCalc r)
        expressionCalc (P.GreaterEq l r) = greaterEqCalc (expressionCalc l)
(expressionCalc r)
        expressionCalc (P.Plus l r) = plusCalc (expressionCalc l) (expressionCalc
r)
        expressionCalc (P.Minus l r) = minusCalc (expressionCalc l)
(expressionCalc r)
        expressionCalc (P.Mult l r) = multCalc (expressionCalc l) (expressionCalc
r)
        expressionCalc (P.Div l r) = divCalc (expressionCalc l) (expressionCalc r)
        expressionCalc (P.Mod l r) =  modCalc (expressionCalc l) (expressionCalc
r)
        expressionCalc (P.Constant i) = constantCalc i
        expressionCalc (P.Var v) = variableCalc v

        -- Now compiling statements to LoopPrograms
        statementCalc :: P.Statement -> UniqueCaclulation L.LoopProgram
        statementCalc (P.Block [stmt]) = statementCalc stmt
        statementCalc (P.Block (st:sts)) = do
            res <- statementCalc st
            res' <- statementCalc (P.Block sts)
            return (res =>> res')

        statementCalc (P.Assignment (P.Variable v) e) = do
            (res', var') <- expressionCalc e
            let var = head var'
            return (res' =>> v <== L.ToVariable var)

        statementCalc (P.Assignment (P.ArrayIndexedAt v e) e') = do
            let Just ( P.ArrayP (low, high)) = Map.lookup v symtbl
            (res, resvar) <- expressionCalc e'
            let var = head resvar
            stsres <- statementCalc (P.Case e [(i, P.Assignment (P.Variable (v ++
show i)) (P.Var (P.Variable var))) | i <- [low..high]])
            return (res =>> stsres)


        statementCalc (P.If e stmt) = do
            (res, vars) <- funCallCalc "ifnzero" [expressionCalc e]
            let var = head vars
            nvar <- decode <$> next
            res' <- statementCalc stmt
            return (res =>> L.ForLoop (nvar, L.ToOne) var res')

        statementCalc (P.IfElse e stmt stmt') = do
```

```
            -- calculate the codition
            (res, vars) <- expressionCalc e
            let var = head vars
            -- get two new variables to hold e && not e
            truevar <- decode <$> next
            falsevar <- decode <$> next
            let trueScale = truevar <% ("ifnzero", [var])
            let falseScale = falsevar <% ("ifzero", [var])

            resif <- statementCalc (P.If (P.Var (P.Variable truevar)) stmt)
            reselse <- statementCalc (P.If (P.Var (P.Variable falsevar)) stmt')
            return (res =>> trueScale =>> falseScale =>> resif =>> reselse)

        -- Case Expression [(Integer, Statement)]
        statementCalc (P.Case e cases) = do
            (res, resvar) <- expressionCalc e
            let var = head resvar
            casesres <- calculateCases var cases
            return (res =>> casesres)
            where
                calculateCases :: L.Variable -> [(Integer, P.Statement)] ->
UniqueCaclulation L.LoopProgram
                calculateCases v [(i, stmt)] = statementCalc (P.If (P.Equals
(P.Constant i) (P.Var (P.Variable v))) stmt)
                calculateCases v ((i, stmt):cs) = do
                    bodyres <- statementCalc (P.If (P.Equals (P.Constant i) (P.Var
(P.Variable v))) stmt)
                    res <- calculateCases v cs
                    return (bodyres =>> res)


        -- For Variable (Expression, Expression) Statement
        statementCalc (P.For (P.Variable v) (lowerBound, upperBound) stmt) = do
            (resl, resvarl) <- expressionCalc lowerBound
            let varl = head resvarl
            (resu, resvaru) <- expressionCalc upperBound
            let varh = head resvaru
            -- calculate the difference because all loop forLoops have to start
            -- at 0 or 1
            diffvar <- decode <$> next
            -- calculate the number the forLoop will be executed here we add one
            -- since in pascal the for loop runs if the loopvariable is equal to
the
            -- upper bound
            let difference = diffvar <% ("sub", [varh, varl]) =>> diffvar <==
(L.ToSucc diffvar)
            loopvar <- decode <$> next
            body <- statementCalc stmt
            let actualBody = body =>> v <== (L.ToSucc v)
            return (resl =>> resu =>> difference =>> (v <== L.ToVariable varl) =>>
L.ForLoop (loopvar, L.ToOne) diffvar actualBody =>> v <== L.ToPred v)

        -- This is the harder part since we have to access an array
        variableCalc :: P.Variable -> UniqueCaclulation (L.LoopProgram,
```

```
[L.Variable])
        variableCalc (P.Variable v) = do
            return (v <== L.ToVariable v, [v])

        variableCalc (P.ArrayIndexedAt v e) = do
            -- This will never fail to match because the semantic analysis
            -- will be prior to compiling to loop
            let Just ( P.ArrayP (low, high)) = Map.lookup v symtbl
            nvar <- decode <$> next
            res <- statementCalc (P.Case e [(i, P.Assignment (P.Variable nvar)
(P.Var (P.Variable (v ++ show i)))) | i <- [low..high]])
            return (res, [nvar])
```