Department of Informatica

Formal Methods in Computer Science

Student :
Ghezae Goitom
MAT.761170

Lecturer :
Prof. Giovanni Pani
Dipartimento di Informatica
Università di Bari

GIMP

Index

Introduction

GIMP is an IMP like simple imperative programming language that has been built for the final exam of a course named “Formal Methods In Computer Science”.

Primary data Types:

The primary data types we are going to use in GIMP are Integer and Boolean.

- Integer (Int) : all integer values from negative to positive including zero
- Boolean (Bool) **:** all boolean values which are true and false

When the variables are declared, the user can choose to immediately assign them a value, or to assign a value later on.

Data Structures:

The data Structure we are going to use in GIMP is array

- Array : holds elements of primary data type, for now it’s implemented to hold only Int values.

Control Structure:

Here are the fundamental Control Structure in GIMP

- If (Control) Structure
  - IF ELSE: The else statement is optional and will execute only if the condition in the if statement evaluates to false.
- While (Reputation) Structure
  - WHILE DO: execute a block of statements continuously until the given condition evaluates to false
- Assignment and Declaration Operators
  - Assignment: Assigning values into given variable
  - Declearation: declarations specify the data type of the variable we’re declaring
- Skip: a skip performs a jump, and does nothing

Operators :

Helps us to to perform operation on variables, in GIMP we implemented operators based on the type expressions. We can see detailed information in grammer.

Grammar

The grammar of GIMP is defined by BNF. Backus-Naur notation (shortly BNF) is a formal mathematical way to describe a language, (to describe the syntax of the programming languages).

Preliminary Definitions:

```
<integer> ::= -<natural> | <natural>
<natural> ::= <digit> | <digit> <natural>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<arrIdentifier> := <lower> | <lower> <alphanum>
<elemIdentifier> := <arrIdentifier> | <arrIdentifier> [ <aexp> ]
<alphanum>:= <upper> <alphanum> | <lower> <alphanum> |
<digit><alphanum> | <upper> | <lower> | <digit>
<lower> := a-z <upper> := A-Z
```

Arithmetical Expressions:

```
<aexp> := <aterm> + <aexp> | <aterm> - <aexp> | <aterm>
<aterm> := <afactor> *<term> | <afactor> /<term>| <afactor>
<afactor> := (<aexp>) |-<afactor> | <elemIdentifier> | <integer>
```

Boolean Expressions:

```
<bexp> := <bterm> && <bexp> | <bterm> || <bexp> | <compareTo> |<bterm>
< bterm > := (<bexp>) | !< bterm > | <elemIdentifier> | True | False
< compareTo > := <aexp> < <aexp> | <aexp> <= <aexp> | <aexp> > <aexp> |
<aexp> >= <aexp> | <aexp> == <aexp> | <aexp> != <aexp>
```

Array Expressions:

<aArrexp> := <aArrFactor> ++ < aArrexp > | <aArrFactor>
<aArrFactor> := <aArray> <aTail> | <aArray>
<aTail> := : <aexp> <aTail> | : <aexp>
<aArray> := [<aexp>] | [<aexp> <aSequence>] | <arrIdentifier>
<aSequence> := , <aexp> <aSequence> | , <aexp>

Command Expressions:

<program> := <command> | <command> <program>
<command> := <assignment> | <skip> | <ifThenElse> | <while> | <for> | <doWhile>
<assignment> := <elemIdentifier> = <aexp> ; | <elemIdentifier> = <bexp> ; | <arrIdentifier> = <aArrexp> ; | <arrIdentifier> = <bArrexp> ;
<skip> := skip ;
<ifThenElse> := if (<bexp>) { <program> } |
if (<bexp>) { <program> } else { <program> }
<while> := while (<bexp>) {<program>}
<for>= for (<assignment> <bexp> ; <assignment>) { <program>}
| for (<assignment> <bexp> ;) { <program>} | for (; <bexp> ; <assignment>) { <program>} | for (; <bexp> ;) { <program>}
<doWhile> := do {<program>} while (<bexp>) ;

Parser

A **parser** is a function taking a string and returning a list of (a,string) pairs, where "a" is a value of the parameterized type and "string" is (by convention) the unparsed remainder of the input. The returned list is potentially empty, which signals a failure in parsing.

It might have made more sense to define parser as a type alias for the function, but types can't be made into instances of typeclasses; therefore, we use newtype with a dummy constructor named P.

```
parse : : Parse a -> String -> [(a, String)]
parse (P p) inp = p inp
```

Parser as a Functor

We'll start by making parser an instance of functor:

```
instance Functor Parser where
  -- fmap :: (a -> b) -> Parser a -> Parser b
  fmap g p = P (\inp -> case parse p inp of
                          []        -> []
                          [(v,out)] -> [(g v,out)])
```

While it's not obvious why a functor instance for parser is useful in its own right, it's actually required to make parser into an applicative, and also when combining parsers using applicative style.

Parser as an Applicative

Parser Combinator is a popular technique for constructing complex parsers from simpler parsers, by means of higher-order functions. In Haskell, one of the ways in which parsers can be elegantly combined is using applicative style. Here's the Applicative instance for **GIMP** Parser.

```
instance Applicative Parser where
  -- pure :: a -> Parser a
  pure v = P (\inp -> [(v,inp)])

  -- <*> :: Parser (a -> b) -> Parser a -> Parser b
  pg <*> px = P (\inp -> case parse pg inp of
                          []         -> []
                          [(g,out)] -> parse (fmap g px) out)
```

Parser as a Monad

Parsers can also be expressed and combined using monadic style. Here's the Monad instance for Parser:

```
instance Monad Parser where
  -- return :: a -> Parser a
  return = pure

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

The Alternative Typeclass

Alternative is a subclass of Applicative which describes choice out of multiple options, by exposing a binary operation < | >.

We will define our instance so that a < | > b produces the result of the first parser if it succeeded, with the second as a fallback. The Alternative typeclass also provides us with the nifty some and many functions, which will allow us to match the same parser repeated 1+ or 0+ times respectively.

Parsing functions:

The Parser is carried on with the definition of the *item* and *satisfy* functions.

Item function:

Item function is used to parse each character of a string, taking the char as input and giving back a parser for it.

```haskell
item :: Parser Char
item =
    P (\input -> case input of
        [] -> []
        (x : xs) -> [(x, xs)]) –
```

Satisfy Function:

Satisfy Function uses item function and if the char read is null returns empty, otherwise item function applied to the char

```haskell
satisfy :: (Char -> Bool) -> Parser Char
sat p =
    do
      x <- item
      if p x then return x else empty
```

Token Function:

Token function is used to detect numbers, identifiers and commands. Takes a parser as an input and returns the same parser but without space.

```
token :: Parser a -> Parser a
token p =
    do
        aSpace
        v <- p
        aSpace
        return v
```

Expression parser:

Now we are going to see the parser for the expressions defined in GIMP.

Aexp: Arithmetic Expression Parser

Aexp is the name of the parser that evaluate arithmetic expressions built up from natural numbers using addition, multiplication, division, subtraction,power and parentheses. In arithmetic expressions division and multiplication have higher priority than addition and subtraction and the operators associate to the right.

```
-- ARITHMETIC EVALUATION --
aExp  :: Parser ArithExpr
aExp = do chain aTerm op
    where
      op =
          (do symbol "+"; return Add)
          <|> do symbol "-"; return Sub

aTerm :: Parser ArithExpr
aTerm = do chain aFactor op
    where
      op =
        (do symbol "*"; return Mul)
        <|> (do symbol "/"; return Div)
        <|> do symbol "**"; return Power


aFactor :: Parser ArithExpr
aFactor = do
    (Constant <$> integer)
        <|> do
            i <- identifier
            do
                symbol "["
                n <- aExp
                symbol "]"
                return (ArrVariable i n)
                <|> return (ArithVariable i)
        <|> do
            symbol "("
            a <- aExp
            symbol ")"
            return a
```

Bexp: Boolean Expression Parser

Similar to aexp, a parser that evaluate Boolean expression is called bexp. The functioning is analogous to aexp: for each Boolean expression exists a unique derivation Tree. In this case the boolean operators <, >, <=, >=, ==,!= and the terminals True, False, Not are defined for GIMP.

Command Expression Parsers

The commands supported in the language are Assignment, Declare, Skip, If Else, While Do. Starting from those commands we define a parser

```haskell
command :: Parser Command
command =
  arithDeclare
    <|> boolDeclare
    <|> arrDeclare
    <|> arithAssign
    <|> boolAssign
    <|> arrAssign
    <|> ifElse
    <|> whiledo
    <|> skip

program :: Parser [Command]
program =
  do many command

arithDeclare :: Parser Command
arithDeclare =
  do
    symbol "int"
    i <- identifier
    symbol "="
    r <- ArithDeclare i <$> aExp
    symbol ";"
    return r

boolDeclare :: Parser Command
boolDeclare =
  do
    symbol "bool"
    i <- identifier
    symbol "="
    r <- BoolDeclare i <$> bExp
    symbol ";"
    return r

arrDeclare  :: Parser Command
arrDeclare  =
  do
    symbol "arr"
    i <- identifier
    symbol "["
    j <- aExp
    symbol "]"
    symbol ";"
    return (ArrDeclare i j)
```

```
arithAssign :: Parser Command
arithAssign =
  do
    i <- identifier
    symbol "="
    r <- ArithAssign i <$> aExp
    symbol ";"
    return r

boolAssign  :: Parser Command
boolAssign  =
  do
    i <- identifier
    symbol "="
    r <- BoolAssign  i <$> bExp
    symbol ";"
    return r

arrAssign  :: Parser Command
arrAssign  =
  do
    i <- identifier
    do            -- id [4]= 455;    assegno in posizione 4 il numero 455
      symbol "["
      j <- aExp
      symbol "]"
      symbol "="
      r <- ArrAssign  i j <$> aExp
      symbol ";"
      return r
      <|>
        do
          symbol "="
          symbol "["
          i' <- aExp
          i'' <- many (do symbol ","; aExp)
          symbol "]"
          symbol ";"
          return (ArrFullAssign i (Array (i':i'')))
      <|>
        do
          symbol "="
          x <- identifier
          symbol ";"
          return (ArrFullAssign i (ArrayVariable x))
```

```haskell
skip  :: Parser Command
skip  =
  do
    symbol "skip"
    symbol ";"
    return Skip

ifElse :: Parser Command
ifElse =
  do
    symbol "if"
    symbol "("
    b <- bExp
    symbol ")"
    symbol "{"
    thenP <- program
    symbol "}"
    do
      symbol "else"
      symbol "{"
      elseP <- program
      symbol "}"
      return (IfElse b thenP elseP)
      <|> do
        return (IfElse b thenP [Skip])
```

```haskell
whiledo :: Parser Command
whiledo =
  do
    symbol "whiledo"
    symbol "("
    b <- bExp
    symbol ")"
    symbol "{"
    p <- program
    symbol "}"
    return (Whiledo b p)
```

ENVIRONMENT

The environment can be seen as a memory and therefore must be kept up to date: the instruction/command that modify the state of the environment are the assignment.

In GIMP environment is defined as a list of Variables, where each one is a tuple*(name, value)*.

```haskell
type Env = [Variable]
```

and variable is defined as a type and holds two elements name and value. They show the name and actual value of the variable like their name saying.

```haskell
data Variable = Variable {name  :: String,
                          value :: Type } deriving Show
```

Arithmetic Expressions Evaluation

The evaluation of Arithmetic Expression, called ArithExprEval, takes in environment and an arithmetic expression as input and returning maybe Int type, which is a monad that can be Just value or nothing.

```
--ARITHMETIC EXPRESSION EVALUATION--

arithExprEval:: Env -> ArithExpr -> Maybe Int

arithExprEval env (Constant i) = Just i

arithExprEval env (ArithVariable i) =
        case searchVariable env i of
                Just (IntType v)-> Just v
                Just _ -> error "type mismatch"
                Nothing -> error "undeclared variable"

arithExprEval env (ArrVariable s i) =
        case searchVariable env s of
                Just (ArrayType a)-> Just (readElemArray a j)
                        where Just j = arithExprEval env i
                Just _ -> error "type mismatch"
                Nothing -> error "undeclared variable"



arithExprEval env (Add a b) = pure (+) <*> (arithExprEval env a) <*> (arithExprEval
 env b)

arithExprEval env (Sub a b) = pure (-
) <*> (arithExprEval env a) <*> (arithExprEval env b)

arithExprEval env (Mul a b) = pure (*) <*> (arithExprEval env a) <*> (arithExprEval
 env b)

arithExprEval env (Div a b) = pure (div) <*> (arithExprEval env a) <*> (arithExprEv
al env b)

arithExprEval env (Power a b) = pure (^) <*> (arithExprEval env a) <*> (arithExprEv
al env b)
```

Boolean Expression Evaluation

For the evaluation of Boolean expression, it’s almost the same as Arithmetic Expression, but obviously the operators are the ones that give a Maybe Bool variable as result, which can assume Just or Nothing as values.

```
-- BOOLEAN EXPRESSION EVALUATION

boolExprEval :: Env -> BoolExpr -> Maybe Bool

boolExprEval env (Boolean b) = Just b

boolExprEval env (BoolVariable s)=
        case searchVariable env s of
                Just (BoolType v) -> Just v
                Just _ -> error "type mismatch"
                Nothing -> error "undeclared variable"

boolExprEval env (Lt a b) = pure (<) <*> (arithExprEval env a) <*> (arithExprEval e
nv b)

boolExprEval env (Gt a b) = pure (>) <*> (arithExprEval env a) <*> (arithExprEval e
nv b)

boolExprEval env (Eq a b) = pure (==) <*> (arithExprEval env a) <*> (arithExprEval
env b)

boolExprEval env (Neq a b) = pure (/=) <*> (arithExprEval env a) <*> (arithExprEval
 env b)

boolExprEval env (Lte a b) = pure (<=) <*> (arithExprEval env a) <*> (arithExprEval
 env b)

boolExprEval env (Gte a b) = pure (>=) <*> (arithExprEval env a) <*> (arithExprEval
 env b)

boolExprEval env (And a b) = pure (&&) <*> (boolExprEval env a) <*> (boolExprEval e
nv b)

boolExprEval env (Or a b) = pure (||) <*> (boolExprEval env a) <*> (boolExprEval en
v b)

boolExprEval env (Not a) = not <$> boolExprEval env a
```

Array Expression Evaluations

In Array Expression Evaluations are implemented the functions to detect Array Type. It was necessary to implement this new type because, to add the possibility to assign values to Arrays a list of values, this implemented new function returned an Array Type, so it was decided to add these kind of expressions.

```
-- ARRAY EXPRESSION EVALUATION

arrExprEval :: Env -> ArrayExpr -> Maybe [Int]
arrExprEval e (Array a) = if hasFailed
                                    then Nothing
                                    else Just $ map (\v -> case v of Just x -> x) r
                                      where hasFailed = or $ map (\v -> case v of
                                                  Nothing -> True
                                                  Just x -> False) r
                                            r = map (\exp -> arithExprEval e exp) a
arrExprEval e (ArrayVariable v) =
  case searchVariable e v of
    Just (ArrayType a) -> Just a
    Nothing -> error "Variable not found"
```

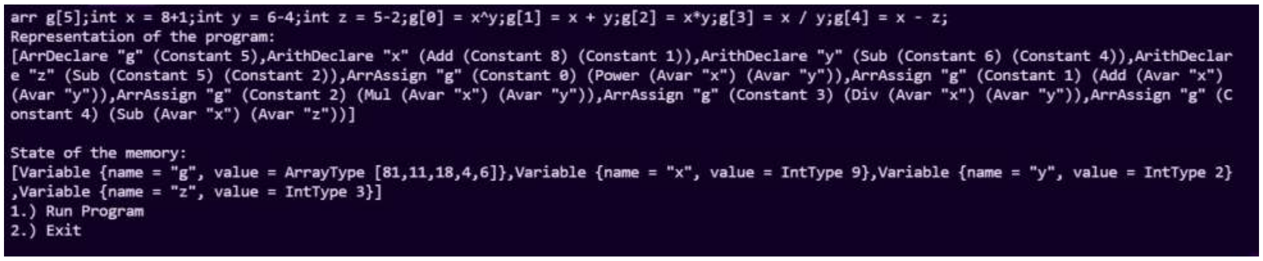
How to run GIMP:

All the directive to run the GIMP interpreter are present in Main.hs, running using “runghc” will let you run the file.

1. Navigate to the project directory
2. Run main.hs using this command “runghc Main.hs”
3. You will be presented with two options “Run program” and “Exit”
4. After typing “1” for “Run program”
5. Type your test code in single line
6. Hit enter and you will get the result

Test Cases:

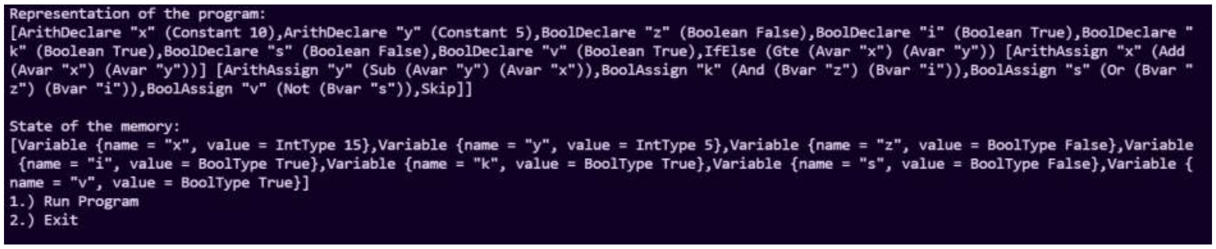
```
■ Test case one
arr g[5];
int x = 8+1;
int y = 3-1-6;
int z = 5-2;
g[0] = x^y;
g[1] = x + y;
g[2] = x*y;
g[3] = x / y;
g[4] = x - z;
```



```
arr g[5];int x = 8+1;int y = 6-4;int z = 5-2;g[0] = x^y;g[1] = x + y;g[2] = x*y;g[3] = x / y;g[4] = x - z;
Representation of the program:
[ArrDeclare "g" (Constant 5),ArithDeclare "x" (Add (Constant 8) (Constant 1)),ArithDeclare "y" (Sub (Constant 6) (Constant 4)),ArithDeclar
e "z" (Sub (Constant 5) (Constant 2)),ArrAssign "g" (Constant 0) (Power (Avar "x") (Avar "y")),ArrAssign "g" (Constant 1) (Add (Avar "x")
(Avar "y")),ArrAssign "g" (Constant 2) (Mul (Avar "x") (Avar "y")),ArrAssign "g" (Constant 3) (Div (Avar "x") (Avar "y")),ArrAssign "g" (C
onstant 4) (Sub (Avar "x") (Avar "z"))]

State of the memory:
[Variable {name = "g", value = ArrayType [81,11,18,4,6]},Variable {name = "x", value = IntType 9},Variable {name = "y", value = IntType 2}
,Variable {name = "z", value = IntType 3}]
1.) Run Program
2.) Exit
```

```
■ Test case two
arr g[5];
int x = 10;
int y = 5;
bool z = False;
bool i = True;
bool k = True;
bool s = False;
bool v = True;
if (x >= y){
    x = x + y;
}else{
    y = y - x;
    k = z And i;
    s = z Or i;
    v = Not s;
    skip;
}
```



```
Representation of the program:
[ArithDeclare "x" (Constant 10),ArithDeclare "y" (Constant 5),BoolDeclare "z" (Boolean False),BoolDeclare "i" (Boolean True),BoolDeclare "
k" (Boolean True),BoolDeclare "s" (Boolean False),BoolDeclare "v" (Boolean True),IfElse (Gte (Avar "x") (Avar "y")) [ArithAssign "x" (Add
(Avar "x") (Avar "y"))] [ArithAssign "y" (Sub (Avar "y") (Avar "x")),BoolAssign "k" (And (Bvar "z") (Bvar "i")),BoolAssign "s" (Or (Bvar "
z") (Bvar "i")),BoolAssign "v" (Not (Bvar "s")),Skip]]

State of the memory:
[Variable {name = "x", value = IntType 15},Variable {name = "y", value = IntType 5},Variable {name = "z", value = BoolType False},Variable
 {name = "i", value = BoolType True},Variable {name = "k", value = BoolType True},Variable {name = "s", value = BoolType False},Variable {
name = "v", value = BoolType True}]
1.) Run Program
2.) Exit
```