Implementing the While Language using Haskell

Ramel Cary B. Jamen
ramelcary.jamen@g.msuiit.edu.ph

CSC153 - Assemblers, Interpreters, and Compilers — May 17, 2024

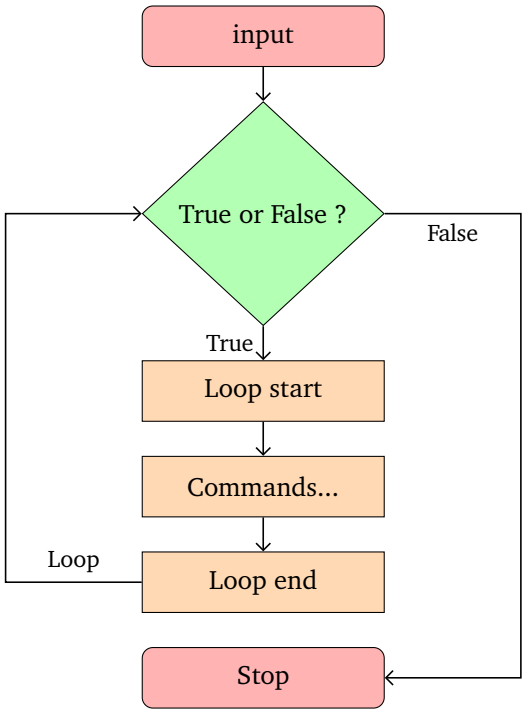
Introduction

The While language, a simple imperative programming language, serves as an excellent educational tool for understanding the fundamentals of programming language design and implementation. In this project, we explore the implementation of the While language using Haskell, a powerful functional programming language known for its expressiveness and strong static typing.

The primary objective of this endeavor is to provide an accessible and instructive example of building a language interpreter in Haskell, showcasing its elegance and conciseness for such tasks. Through this documentation, we aim to delve the process of parsing While programs, evaluating expressions and statements, and executing While code within the Haskell environment.

1 The While Language

The While language is a minimalistic imperative language featuring basic control flow constructs, arithmetic expressions, and boolean operations. Its syntax is designed to be simple yet expressive, making it suitable for introductory programming language courses and exercises.



In the above diagram, the condition "True or False" is a Boolean expression that is evaluated before each iteration of the loop. If the condition is true, the command statements inside the loop will be executed. If the condition is false, the loop will stop. That is how while-do function works.

2 Implementation Overview

Our implementation of the While language interpreter in Haskell follows a modular approach, dividing the implementation into distinct components: parsing, evaluation, and execution. Each component is responsible for handling specific aspects of the language semantics that facilitates clarity of the codebase.

- Parsing: The parser reads the source code and constructs an abstract syntax tree (AST) representing the program's structure. This phase ensures that the input code adheres to the language's grammar.
- Evaluation: The evaluator processes the AST, executing statements and evaluating expressions according to the language's semantics. This phase involves arithmetic operations, variable bindings, and control flow constructs.
- Execution: The final phase executes the evaluated program, performing the necessary computations specified by the While program.

3 Parser Implementation

Functional parsing, a powerful technique for analyzing and processing textual data, forms the backbone of our While language interpreter. In this section, we delve into the implementation details of the parser module, which serves as the foundation for parsing While language programs

3.1 Lexical Analysis

Our parser module is inspired by the functional parsing library described in Chapter 8 of "Programming in Haskell" by Graham Hutton, published by Cambridge University Press in 2007. This library provides the tools necessary for defining parsers using functional programming techniques.

Haskell Code

```
> import Data.Char
> import Control.Monad
> import qualified Control.Applicative as CA
> infixr 5 +++
```

Setting the '+++' operator to be right-associative and with a precedence level of 5. At the core of our parser module lies the Parser monad, defined as follows:

Haskell Code

```
> newtype Parser a  = P (String -> [(a,String)])
>
> instance CA.Applicative Parser where
>    pure           = return
>    (<*>)          = ap
```

Haskell Code

```
> instance Functor Parser where
>    fmap           = liftM

> instance CA.Alternative Parser where
>    (<|>)          = mplus
>    empty          = mzero

> instance Monad Parser where
>    return v       = P (\inp -> [(v,inp)])
>    p >>= f        = P (\inp -> case parse p inp of
>                           []        -> []
>                           [(v,out)] -> parse (f v) out)

> instance MonadPlus Parser where
>    mzero          = P (\inp -> [])
>    p `mplus` q    = P (\inp -> case parse p inp of
>                           []        -> parse q inp
>                           [(v,out)] -> [(v,out)])
```

This monadic structure allows us to compose parsers sequentially, chaining together parsing operations to handle complex grammatical structures.

For the basic parsers, we define it to handle fundamental parsing operations:

- failure: Represents a failed parsing attempt.
- item: Parses a single character from the input string.
- parse: Invokes the parser on the input string.

Haskell Code

```
> failure :: Parser a
> failure = mzero

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

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

The +++ operator allows us to define parsers that can choose between multiple alternatives.

Haskell Code

```
> (+++) :: Parser a -> Parser a -> Parser a
> p +++ q = p `mplus` q
```

We derive more complex parsers from basic ones:

- sat: Parses a character satisfying a given predicate.
- string: Parses a specific string.
- sep0 and sep1: Parses a sequence of commands separated by semicolon.
- ident, nat, int: Parsers for identifiers, natural numbers, and integers respectively.
- space: Parses whitespace characters.

Haskell Code

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

> string :: String -> Parser String
> string []         = return []
> string (x:xs)     = do char x
>                        string xs
>                        return (x:xs)

> many :: Parser a -> Parser [a]
> many p = many1 p +++ return []

> many1 :: Parser a -> Parser [a]
> many1 p = do v <- p
>              vs <- many p
>              return (v:vs)

> ident :: Parser String
> ident = token $ do x <- lower
>                    xs <- many alphanum
>                    return (x:xs)

> nat :: Parser Int
> nat = token $ do xs <- many1 digit
>                  return (read xs)

> int :: Parser Int
> int = token $ (do char '-'
>                   n <- nat
>                   return (-n))
>               +++ nat

> space :: Parser ()
> space = do many (sat isSpace)
>            return ()
```

The token function wraps a parser, ignoring leading and trailing whitespace.

Haskell Code

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

We provide parsers for identifiers and symbols, which are crucial for parsing While language programs.

Haskell Code

```
> identifier :: Parser String
> identifier = token ident

> symbol :: String -> Parser String
> symbol xs = token (string xs)
```

We add a specialized parser 'true' or 'false' to handle parsing of True and False literals in the While language.

Haskell Code

```
> truefalse :: Parser Char
> truefalse = sat isTF
```

We have now covered the basics of lexical analysis or also known as tokenization, that involves breaking down the input string into tokens, which includes identifiers, keywords, operators, literals, and other elements of the language syntax.

3.2 Syntax Analysis

In this section, we can now construct ASTs from the sequence of tokens, by defining parsers for expressions, terms, factors, statements, and programs using combinators to capture the structure of the program for further evaluation.

An arithmetic expression (AExp) in the While language can be represented by a numeric value (Num), a variable (Var), an arithmetic operation between two AExp operands using an arithmetic operator (AOp), or enclosed within parentheses.

$$
\begin{array}{lll}
\langle AExp \rangle & ::= & \langle Num \rangle \\
& | & \langle Var \rangle \\
& | & \langle AExp \rangle\ \langle AOp \rangle\ \langle AExp \rangle \\
& | & \text{`(' } \langle AExp \rangle \text{ `)'} \\
\\
\langle AOp \rangle & ::= & \text{`+'} \mid \text{`-'} \mid \text{`*'} \\
\\
\langle Num \rangle & ::= & \text{`0'} \mid \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \text{`5'} \mid \text{`6'} \mid \\
& & \text{`7'} \mid \text{`8'} \mid \text{`9'} \\
\\
\langle Var \rangle & ::= & \langle String \rangle
\end{array}
$$

On the other hand, boolean expression (BExp) in can be represented by a boolean value (BVal), a negation operation (!) applied to another BExp, a relational operation (ROp) between two AExp operands, a logical operation (LOp) between two BExp operands, or enclosed within parentheses.

$$
\begin{array}{lll}
\langle BExp \rangle & ::= & \langle BVal \rangle \\
& | & \text{`!' } \langle BExp \rangle \\
& | & \langle AExp \rangle\ \langle ROp \rangle\ \langle AExp \rangle \\
& | & \langle BExp \rangle\ \langle LOp \rangle\ \langle BExp \rangle \\
& | & \text{`(' } \langle BExp \rangle \text{ `)'}
\end{array}
$$

⟨*ROp*⟩ ::= ‘==’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘!=’

⟨*LOp*⟩ ::= ‘&’ | ‘|’

⟨*BVal*⟩ ::= ‘T’ | ‘F’

The While language program structure is defined by a series of statements separated by semicolons (;). Each statement can be a skip statement (Skip), a print statement (Print), an assignment statement (Assignment), a while-do loop statement (WhileDoStatement), or a block statement (Block).

⟨*Commands*⟩ ::= ⟨*Statement*⟩ ‘;’ ⟨*Commands*⟩
| ⟨*Statement*⟩

⟨*Statement*⟩ ::= ⟨*Skip*⟩
| ⟨*Print*⟩
| ⟨*Assignment*⟩
| ⟨*WhileDoStatement*⟩
| ⟨*Block*⟩

⟨*Skip*⟩ ::= ‘skip’

⟨*Print*⟩ ::= ‘print’ ‘(’ ⟨*Identifier*⟩ ‘)’

⟨*Block*⟩ ::= ‘begin’ ⟨*Commands*⟩ ‘end’
| ‘{’ ⟨*Commands*⟩ ‘}’

⟨*Assignment*⟩ ::= ⟨*Identifier*⟩ ‘:=’ ⟨*AExp*⟩

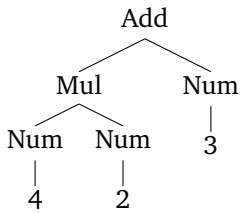
⟨*WhileDoStatement*⟩ ::= ‘while’ ⟨*BExp*⟩ ‘do’ ⟨*Commands*⟩

This grammar provides a formal specification of the syntax and structure of While language expressions and programs, facilitating the understanding and implementation of While language.

3.3 Parsing Program

3.3.1 Arithmetic Expression

Precedence in arithmetic expressions (AExp) determines the order in which operations are performed. For example, in the expression below, "3 + 4 * 2", multiplication has higher precedence than addition, so 4 * 2 is evaluated first, then 3 + 8, resulting in 11.



The resulting AST represents the correct precedence and associativity, note that the AST is evaluated starting from the right.

Haskell Code

```
> expr :: Parser AExp
> expr = do t <- term
>           do symbol "-"; e <- expr; return (A Sub t e)
>             +++ do symbol "+"; e <- expr; return (A Add t e)
>             +++ return t

> term :: Parser AExp
> term = do f <- factor
>           do symbol "*"; t <- term; return (A Mul f t)
>             +++ return f

> factor :: Parser AExp
> factor = do symbol "("
>             e <- expr; symbol ")"; return e
>             +++ number
>             +++ var

> number :: Parser AExp
> number = do x <- natural; return (Num x)

> var :: Parser AExp
> var = do v <- identifier; return (Var v)

Input: "3 + 4 * 2"
Result: Add (Num 3) ( Mul (Num 4) (Num 2))
```

The expr parser handles addition and subtraction operations. It first parses a term and then checks for either a addition or subtraction symbol, followed by another expression (expr). This recursive structure ensures that expressions are evaluated from right to left, maintaining the correct precedence.

The term parser deals with multiplication operations. Similar to expr, it first parses a factor, then checks for a multiplication symbol, and recursively parses another term if found. This structure ensures that multiplication operations are evaluated before addition and subtraction.

The factor parser handles factors in expressions, which can be either parenthesized expressions, numbers, or variables. It first checks for a left parenthesis symbol, then recursively parses an expression, and finally checks for a right parenthesis symbol. If no parenthesis are found, it attempts to parse a number or a variable.

3.3.2 Boolean Expression

In Boolean expressions (BExp), precedence determines the order in which logical operations and relational comparisons are performed. For instance, in the expression "TRUE & FALSE | TRUE", the & operation is performed before the | operation because & has higher precedence.

Haskell Code

```
> bexp :: Parser BExp
> bexp = (do t <- bterm; symbol "|"; e <- bexp;
>             return (L Or t e))
>             +++ bterm

> bterm :: Parser BExp
> bterm = (do f <- bfactor; symbol "&"; t <- bterm;
>             return (L And f t))
>             +++ bfactor

> bfactor :: Parser BExp
> bfactor = nbval +++ bval

> bval :: Parser BExp
> bval = bpar +++ tval +++ rexp

> nbval :: Parser BExp
> nbval = do symbol "!"; f <- bfactor; return (Not f)

> tval :: Parser BExp
> tval = (do symbol "F"; return (BVal False))
>           +++ (do symbol "T"; return (BVal True))

> bpar :: Parser BExp
> bpar = do symbol "("; e <- bexp; symbol ")"; return e

> rexp :: Parser BExp
> rexp = eq +++ lt +++ gt +++ lteq +++ gteq +++ noteq

> eq :: Parser BExp
> eq = do l1 <- expr; symbol "=="; l2 <- expr;
>             return (R Equ l1 l2)

> lt :: Parser BExp
> lt = do l1 <- expr; symbol "<"; l2 <- expr;
>             return (R Lt l1 l2)

> gt :: Parser BExp
> gt = do l1 <- expr; symbol ">"; l2 <- expr;
>             return (R Gt l1 l2)

> lteq :: Parser BExp
> lteq = do l1 <- expr; symbol "<="; l2 <- expr;
>             return (R Lte l1 l2)

> gteq :: Parser BExp
> gteq = do l1 <- expr; symbol ">="; l2 <- expr;
>             return (R Gte l1 l2)

> noteq :: Parser BExp
> noteq = do l1 <- expr; symbol "!="; l2 <- expr;
>             return (R Ne l1 l2)

Input: "T & F | T"
Result: Or (And (BVal True) (BVal False)) (BVal True)
```

The "bexp" parser handles logical OR operations. It first parses a "bterm", followed by a symbol for logical OR ("|"), and then recursively parses another "bexp". This structure ensures that logical OR operations are evaluated after logical AND operations, reflecting the precedence rules where logical AND has higher precedence than logical OR.

The bterm parser, on the other hand, deals with logical AND operations. Similar to "bexp", it first parses a "bfactor", then checks for a symbol for logical AND ("&"), and recursively parses another "bterm" if found. This structure ensures that logical AND operations are evaluated before logical OR operations.

The "bfactor" parser handles the factors in boolean expressions, which can be negated boolean values, boolean values themselves (either TRUE or FALSE), or relational expressions. If a negation symbol (!) is found, it recursively parses another "bfactor". If a boolean value is found (T or F), it constructs a boolean value. If none of these are found, it attempts to parse a relational expression.

The relational expressions handle comparisons between arithmetic expressions. These comparisons include equality (==), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), and not equal to (!=). Each relational parser first parses two arithmetic expressions and constructs the corresponding relational operation.

3.3.3 While Program

We have now a framework for Arithmetic Expression and Boolean Expression, based on the structure that is given in the syntax analysis, we may proceed to the While language with a factorial program as an example:

Haskell Code

```
> data Com
>     = Skip
>     | Print String
>     | Assign String AExp
>     | WhileDo BExp Com
>     | Coms [Com]
>     deriving Show

> sep1 :: Parser Com -> Parser a -> Parser [Com]
> sep1 p sep = do x <- p; xs <- many (sep >> p); return (x : xs)

> sep0 :: Parser Com -> Parser sep -> Parser [Com]
> sep0 p sep = sep1 p sep +++ return []

> coms = do lst <- (sep0 comparse (symbol ";"));
>     return (if length lst == 1 then head lst else Coms lst)

> comparse :: Parser Com
> comparse = skip +++ block +++ cblock +++
>            assign +++ printcom +++ wdloop

> skip :: Parser Com
> skip = do symbol "skip"; return Skip
```

Haskell Code

```
> assign :: Parser Com
> assign = do id <- identifier; symbol ":="; e <- expr;
>          return (Assign id e)

> printcom :: Parser Com
> printcom = do symbol "print"; symbol "(";
>          v <- identifier; symbol ")"; return (Print v)

> block :: Parser Com
> block = do symbol "begin"; t <- coms; symbol "end";
>          return (t) +++ comparse

> cblock :: Parser Com
> cblock = do symbol "{"; t <- coms; symbol "}";
>          return (t) +++ comparse

> wdloop :: Parser Com
> wdloop = do symbol "while"; b <- bexp; symbol "do";
>          ct <- comparse; return (WhileDo b ct)

Input: "x := 10; y := 1; while x > 0
        do { y := y * x; x := x - 1 }; print(y)"

Output:
Coms [Assign "x" (Num 10),Assign "y" (Num 1),
WhileDo (R Gt (Var "x") (Num 0)) (Coms [Assign "y"
(A Mul (Var "y") (Var "x")), Assign "x" (A Sub (Var "x")
(Num 1))]),Print "y"]
```

The coms parser is responsible for parsing sequences of commands separated by semicolons in the While language. It takes a string input, separates the commands by semicolons, and then analyzes each command to create a structured AST according to the grammar rules of the language.

3.3.4 Additional

We can also extend the Statement to support the following (Implemented in Source code):

$$
\begin{aligned}
\langle \textit{Statement} \rangle \quad &::= \ \ldots \\
&\mid \ \langle \textit{IfStatement} \rangle \\
&\mid \ \langle \textit{DoWhileStatement} \rangle \\
&\mid \ \langle \textit{RepeatUntilStatement} \rangle \\
&\mid \ \langle \textit{ForLoopStatement} \rangle
\end{aligned}
$$

$$
\langle \textit{IfStatement} \rangle \quad ::= \ \text{`}\texttt{if}\text{'} \ \langle \textit{BExp} \rangle \ \text{`}\texttt{then}\text{'} \ \langle \textit{Commands} \rangle \ \text{`}\texttt{else}\text{'} \ \langle \textit{Commands} \rangle
$$

$$
\langle \textit{DoWhileStatement} \rangle ::= \ \text{`}\texttt{do}\text{'} \ \langle \textit{Commands} \rangle \ \text{`}\texttt{while}\text{'} \ \langle \textit{BExp} \rangle
$$

$$
\langle \textit{RepeatUntilStatement} \rangle ::= \ \text{`}\texttt{repeat}\text{'} \ \langle \textit{Commands} \rangle \ \text{`}\texttt{until}\text{'} \ \langle \textit{BExp} \rangle
$$

$$
\langle \textit{ForLoopStatement} \rangle ::= \ \text{`}\texttt{for}\text{'} \ \text{`}(\text{'} \ \langle \textit{Assignment} \rangle \ \text{`};\text{'} \ \langle \textit{BExp} \rangle \ \text{`};\text{'} \ \langle \textit{Assignment} \rangle \ \text{`})\text{'} \ \text{`}\texttt{do}\text{'} \ \langle \textit{Commands} \rangle
$$

4 Translation and Evaluation

Haskell Code

```
> type Loc = String
> data Ins = PUSH Int | ADD | MUL | SUB
>          | TRUE | FALSE | EQU | GRT | LST
>          | GTE | LTE | AND | OR | NOT
>          | PRINT Loc | LOAD Loc | STORE Loc | NOOP
>          | BRANCH (Code, Code) | LOOP (Code, Code)
>          deriving Show
> type Code = [Ins]
```

These type definitions and data constructors form the core of the abstract machine's instruction set. "Loc" represents variable names or memory locations, "Ins" represents the possible instructions that can manipulate the stack or storage, and "Code" is a sequence of these instructions that constitutes a program.

In this section, we define the translation of arithmetic and boolean expressions as well as commands from the While language into an abstract machine code. The translation functions take expressions or commands and produce a list of instructions that the abstract machine will execute.

The evaluator component on the other hand, interprets the parsed ASTs of While programs, executing statements and evaluating expressions according to the language semantics. By traversing the AST recursively, the evaluator computes the results of expressions and performs the corresponding actions specified by the statements.

4.1 Translation

4.1.1 Arithmetic Expression

The translation of arithmetic expressions (AExp) is straightforward. Each type of arithmetic expression is mapped to a corresponding sequence of instructions.

Haskell Code

```
Translate an arithmetic expression to abstract machine code

> tr_a :: AExp -> Code
> tr_a (Num n)        = [PUSH n]
> tr_a (Var x)        = [LOAD x]
> tr_a (A Add a1 a2)  = tr_a a2 ++ tr_a a1 ++ [ADD]
> tr_a (A Sub a1 a2)  = tr_a a2 ++ tr_a a1 ++ [SUB]
> tr_a (A Mul a1 a2)  = tr_a a2 ++ tr_a a1 ++ [MUL]

Input: "Add (Num 3) (Mul (Num 4) (Num 2))"

Output: [PUSH 2,PUSH 4,MUL,PUSH 3,ADD]
```

4.1.2 Boolean Expression

Boolean expressions (BExp) are translated similarly, with each type of boolean operation corresponding to specific instructions.

Haskell Code

```
Translate a boolean expression to abstract machine code

> tr_b :: BExp -> Code
> tr_b (BVal True)    = [TRUE]
> tr_b (BVal False)   = [FALSE]
> tr_b (Not l1)       = tr_b l1 ++ [NOT]
> tr_b (R Equ l1 l2)  = tr_a l2 ++ tr_a l1 ++ [EQU]
> tr_b (R Gt l1 l2)   = tr_a l2 ++ tr_a l1 ++ [GRT]
> tr_b (R Lt l1 l2)   = tr_a l2 ++ tr_a l1 ++ [LST]
> tr_b (R Gte l1 l2)  = tr_a l2 ++ tr_a l1 ++ [GTE]
> tr_b (R Lte l1 l2)  = tr_a l2 ++ tr_a l1 ++ [LTE]
> tr_b (L And l1 l2)  = tr_b l1 ++ tr_b l2 ++ [AND]
> tr_b (L Or l1 l2)   = tr_b l1 ++ tr_b l2 ++ [OR]

Input: "Or (And (Gt (Var "x") (Num 1)) (BVal False)) (BVal True)"

Output: [PUSH 1,LOAD "x",GRT,FALSE,AND,TRUE,OR]
```

4.1.3 Translating Commands

The translation of commands (Coms) includes various types of statements in the While language, each translating to specific sequences of instructions.

Haskell Code

```
Translate a commands to abstract machine code

> tr_c :: Com -> Code
> tr_c (Coms (c:cs))  =  tr_c c ++ if length cs == 1 then
>                        tr_c (head cs) else tr_c > (Coms cs)
> tr_c (Skip)         = [NOOP]
> tr_c (Print v)      = [PRINT v]
> tr_c (Assign m v)   = tr_a v ++ [STORE m]
> tr_c (If b c1 c2)   = tr_b b ++ [BRANCH(tr_c c1, tr_c c2)]
> tr_c (WhileDo b c)  = [LOOP (tr_b b, tr_c c)]
> tr_c (DoWhile c b)  = tr_c (Coms (c : [WhileDo b c]))
> tr_c (RepUntil c b) = tr_c (Coms (c : [WhileDo (Not b) c]))
> tr_c (ForLoop (c1, b, c2) c3) =
>           tr_c (Coms (c1 : [WhileDo b (Coms (c3 : [c2]))]))

Input: Coms [Assign "x" (Num 2), WhileDo (R Gt (Var "x") (Num 0))
            (Coms [Assign "x" (A Sub (Var "x") (Num 1)), Print "x"
            ])]

Output: [PUSH 2,STORE "x",LOOP ([PUSH 0,LOAD "x",GRT],
        [PUSH 1,LOAD "x",SUB,STORE "x",PRINT "x"])]
```

4.2 Evaluation

The evaluation of the compiled code is performed by an abstract machine that processes the instructions step by step. The machine state consists of the code to execute, a stack, storage, and a debug log.

Haskell Code

```
> type ValZB      = Z Int | B Bool deriving Show
> type Stack      = [ValZB]
> type Binding    = (Loc, Int)
> type Storage    = [Binding]
> type Debug      = [Binding]
> type MState     = (Code, Stack, Storage, Debug)

Initialize the abstract machine state

> amInit :: Code -> MState
> amInit xs = (xs, [], [], [])

Input: [PUSH 5,STORE "x",LOOP ([PUSH 0,LOAD "x",GRT],
       [PUSH 1,LOAD "x",SUB,STORE "x",PRINT "x"])]

Output: ([PUSH 5,STORE "x",LOOP ([PUSH 0,LOAD "x",GRT],
        [PUSH 1,LOAD "x",SUB,STORE "x",PRINT "x"])],[],[],[])
```

4.2.1 Evaluation Functions

Evaluation Function (amEval), this function performs a single step of evaluation, updating the machine state based on the current instruction.

Haskell Code

```
> amEval :: MState -> MState
> amEval (PUSH n:cs, e, s, d) = (cs, Z n:e, s, d)
> amEval (ADD:cs, Z n1: Z n2:e, s, d) = (cs, Z (n1 + n2):e, s, d)
> amEval (SUB:cs, Z n1: Z n2:e, s, d) = (cs, Z (n1 - n2):e, s, d)
> amEval (MUL:cs, Z n1: Z n2:e, s, d) = (cs, Z (n1 * n2):e, s, d)
> amEval (TRUE:cs, e, s, d) = (cs, B True:e, s, d)
> amEval (FALSE:cs, e, s, d) = (cs, B False:e, s, d)
> amEval (EQU:cs, Z n1: Z n2:e, s, d) = (cs, B (n1 == n2):e, s, d)
> amEval (GRT:cs, Z n1: Z n2:e, s, d) = (cs, B (n1 > n2):e, s, d)
> amEval (LST:cs, Z n1: Z n2:e, s, d) = (cs, B (n1 < n2):e, s, d)
> amEval (LTE:cs, Z n1: Z n2:e, s, d) = (cs, B (n1 <= n2):e, s, d)
> amEval (GTE:cs, Z n1: Z n2:e, s, d) = (cs, B (n1 >= n2):e, s, d)
> amEval (AND:cs, B b1: B b2:e, s, d) = (cs, B (b1 && b2):e, s, d)
> amEval (OR:cs, B b1: B b2:e, s, d) = (cs, B (b1 || b2):e, s, d)
> amEval (NOT:cs, B b:e, s, d) = (cs, B (not b):e, s, d)
> amEval (LOAD x:cs, e, s, d) = (cs, Z (valueOf x s):e, s, d)
> amEval (STORE x:cs, Z z:e, s, d) = (cs, e, update s x z, d)
```

Haskell Code

```
> amEval (NOOP:cs, e, s, d) = (cs, e, s, d)
> amEval (BRANCH (c1, c2):cs, B b:e, s, d) =
>       if b then (c1 ++ cs, e, s, d) else (c2 ++ cs, e, s, d)
> amEval (LOOP(c1, c2):cs, b, s, d) =
>       (c1 ++ BRANCH(c2 ++ [LOOP(c1, c2)],[NOOP]):cs, b, s, d)
> amEval (PRINT x:cs, e, s, d) = (cs, e, s, (x, valueOf x s):d)
> amEval (c:cs, e, s, d) = (cs, e, s, d)
> amEval ([], e, s, d) = ([], e, s, d)
```

This "eval" function repeatedly applies amEval until the code list is empty, representing the completion of the program. It ensures the program is fully executed.

Haskell Code

```
eval :: MState -> MState
eval ([], e, s, d) = ([], e, s, d)
eval state = eval (amEval state)
```

4.2.2 Utility Functions for Storage and Debug

The "valueOf" function retrieves the value associated with a given location in the storage. If the location is not found, it raises an error.

Haskell Code

```
> valueOf :: Loc -> Storage -> Int
> valueOf loc [] = error ("Unused input " ++ loc)
> valueOf loc ((key, val):rest)
>     | loc == key = val
>     | otherwise = valueOf loc rest
```

This "update" function updates the value associated with a given location in the storage. If the location is not found, it adds a new binding.

Haskell Code

```
> update :: Storage -> Loc -> Int -> Storage
> update [] loc newVal = [(loc, newVal)]
> update ((key, val):rest) loc newVal
>     | loc == key = (loc, newVal) : rest
>     | otherwise = (key, val) : update rest loc newVal
```

5 Execution and Testing

5.1 Interface Functions

These interface functions collectively provide a comprehensive framework for compiling, parsing, and executing programs written in the While language. They enable users to convert high-level program descriptions into executable machine code, run these programs, and observe their behavior through debugging output.

5.1.1 parseAExpr

Parses an arithmetic expression from a string.

Haskell Code

```
> parseAExpr :: String -> AExp
> parseAExpr xs = case (parse expr xs) of
>     [(n,[])] -> n
>     [(_,out)] -> error ("Unused input " ++ out)
>     [] -> error "Invalid input"

Example usage:

> parseAExpr "x + 3 * (y - 2)"

Output: A Add (Var "x") (A Mul (Num 3) (A Sub (Var "y") (Num 2)))
```

5.1.2 parseBExpr

Parses a boolean expression from a string.

Haskell Code

```
> parseBExpr :: String -> BExp
> parseBExpr rstr = case parse bexp rstr of
>     [(n,"")] -> n
>     [(_, out)] -> error ("Unused input " ++ out)
>     [] -> error "Invalid input"

Example usage:

> parseBExpr "x < 5 & (y >= 3 | z != 4)"

Output: L And (R Lt (Var "x") (Num 5)) (L Or (R Gte (Var "y")
(Num 3)) (R Ne (Var "z") (Num 4)))
```

5.1.3 parse_com

Parses a single command from a string.

Haskell Code

```
> parse_com :: String -> Com
> parse_com rstr = case parse seqcom rstr of
>     [(n,"")] -> n
>     [(_, out)] -> error ("Unused input " ++ out)
>     [] -> error "Invalid input"

Example usage:

> parse_com "x := 5; print(x)"

Output: Seq (Assign "x" (Num 5)) (Print "x")
```

5.1.4 parse_coms

Parses multiple commands from a string.

Haskell Code

```
> parse_coms :: String -> Com
> parse_coms rstr = case parse coms rstr of
>     [(n,"")] -> n
>     [(_, out)] -> error ("Unused input " ++ out)
>     [] -> error "Invalid input"

Example usage:

> parse_coms "x := 5; y := 3; z := x * y"

Output:  Coms [Assign "x" (Num 5), Assign "y" (Num 3),
Assign "z" (A Mul (Var "x") (Var "y"))]
```

5.1.5 compile

Compiles a given program string into a list of Abstract Machine instructions.

Haskell Code

```
> compile :: String -> Code
> compile = tr_c . parse_coms

Example usage:

> compile "x := 5; y := 3; z := x * y"

Output: [PUSH 5, STORE "x", PUSH 3, STORE "y", LOAD "y",
LOAD "x", MUL, STORE "z"]
```

5.1.6 amInit

Initializes the Abstract Machine state with a given list of instructions.

Haskell Code

```
> amInit :: Code -> MState
> amInit xs = (xs, [], [], [])

Example usage:

> amInit [PUSH 5, STORE "x"]

Output: ([PUSH 5, STORE "x"], [], [], [])
```

5.1.7 eval

Evaluates the Abstract Machine instructions from an initial state to a final state.

Haskell Code

```
> eval :: MState -> MState
> eval ([], e, s, d) = ([], e, s, d)
> eval state = eval (amEval state)

Example usage:

> eval ([PUSH 5, PUSH 3, ADD, STORE "x"], [], [], [])

Output: ([], [], [("x", 8)], [])
```

5.1.8 amExec

Executes a given program string and returns the final state of the Abstract Machine.

Haskell Code

```
> amExec :: String -> MState
> amExec prog = eval (amInit (compile prog))

Example usage:

> amExec "x := 5; y := 3; z := x + y"
Output: ([], [], [("x",5), ("y",3), ("z",8)], [])
```

5.1.9 amGetVal

Retrieves the value of a variable after executing a given program string.

Haskell Code

```
> amGetVal :: String -> String -> Int
> amGetVal var prog = result where
>         result  = valueOf var storage
>         (_,_,storage,_)   = eval (amInit (compile prog))

Example usage:
> amGetVal "z" "x := 5; y := 3; z := x + y"

Output: 8
```

5.1.10 amRun

Executes a given program string and prints the output.

Haskell Code

```
> amRun :: String -> IO ()
> amRun prog = mapM_ (putStrLn . show . snd) (debugList)
>     where
>         (_, _, _, debug) = eval (amInit (compile prog))
>         debugList = reverse debug

Example usage:

> amRun "x := 5; y := 1; while x > 0 do { y := y * x; x := x - 1 };
print(y)"

Output: 120
```

6 Conclusion

In this project, we have developed a comprehensive functional parsing library based on Graham Hutton's "Programming in Haskell." The parser is capable of interpreting arithmetic and boolean expressions and translating them into abstract machine code. By leveraging Haskell's strong type system and monadic parsing capabilities, we ensured that the parsing process is both robust and flexible, allowing for clear and maintainable code.

The abstract machine state and associated operations provide a solid foundation for evaluating parsed expressions and executing commands. The design of the abstract machine and its evaluation function demonstrates the power of functional programming in simulating complex computational processes. Through careful implementation, we have created a system that can parse, compile, and execute a wide range of expressions and commands, making it a versatile tool for interpreting a small language.

The interface functions, amExec, amGetVal, and amRun, offer an accessible way to interact with the parsing and execution engine. These functions allow users to execute

programs, retrieve variable values, and print output as specified by the programs' instructions, thereby making the system user-friendly and practical for various applications.

In summary, this project showcases the effectiveness of Haskell for building a functional parsing library and an abstract machine interpreter. It highlights the importance of clear design and the advantages of using functional programming paradigms for such tasks. The resulting system is not only a demonstration of theoretical concepts but also a practical tool for parsing and executing small programming languages.

7 Appendix

7.1 Haskell Code

```
Ramel Cary B. Jamen
2019-2093

Functional parsing library from chapter 8 of Programming in Haskell,
Graham Hutton, Cambridge University Press, 2007.

> import Data.Char
> import Control.Monad
> import qualified Control.Applicative as CA

> infixr 5 +++

> newtype Parser a              =  P (String -> [(a,String)])

> instance CA.Applicative Parser where
>    pure  = return
>    (<*>) = ap

> instance Functor Parser where
>    fmap  = liftM

> instance CA.Alternative Parser where
>    (<|>) = mplus
>    empty = mzero

> instance Monad Parser where
>    return v                   =  P (\inp -> [(v,inp)])
>    p >>= f                    =  P (\inp -> case parse p inp of
>                                               []        -> []
>                                               [(v,out)] -> parse (f v) out)

> instance MonadPlus Parser where
>    mzero                      =  P (\inp -> [])
>    p `mplus` q                =  P (\inp -> case parse p inp of
>                                               []        -> parse q inp
>                                               [(v,out)] -> [(v,out)])

> failure                       :: Parser a
> failure                       =  mzero
```

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

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

> (+++)                         :: Parser a -> Parser a -> Parser a
> p +++ q                       =  p `mplus` q

> sat                           :: (Char -> Bool) -> Parser Char
> sat p                         =  do x <- item
>                                     if p x then return x else failure

> digit                         :: Parser Char
> digit                         =  sat isDigit

> lower                         :: Parser Char
> lower                         =  sat isLower

> upper                         :: Parser Char
> upper                         =  sat isUpper

> letter                        :: Parser Char
> letter                        =  sat isAlpha

> alphanum                      :: Parser Char
> alphanum                      =  sat isAlphaNum

> char                          :: Char -> Parser Char
> char x                        =  sat (== x)


> string                        :: String -> Parser String
> string []                     =  return []
> string (x:xs)                 =  do char x
>                                     string xs
>                                     return (x:xs)

> many                          :: Parser a -> Parser [a]
> many p                        =  many1 p +++ return []

> many1                         :: Parser a -> Parser [a]
> many1 p                       =  do v  <- p
>                                     vs <- many p
>                                     return (v:vs)

> ident                         :: Parser String
> ident                         =  do x  <- lower
>                                     xs <- many alphanum
>                                     return (x:xs)

> nat                           :: Parser Int
> nat                           =  do xs <- many1 digit
>                                     return (read xs)

```

```
> int                              :: Parser Int
> int                              =  do char '-'
>                                        n <- nat
>                                        return (-n)
>                                     +++ nat

> space                            :: Parser ()
> space                            =  do many (sat isSpace)
>                                        return ()

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

> identifier                       :: Parser String
> identifier                       =  token ident

> natural                          :: Parser Int
> natural                          =  token nat

> integer                          :: Parser Int
> integer                          =  token int

> symbol                           :: String -> Parser String
> symbol xs                        =  token (string xs)

> p                                :: Parser (Char, Char)
> p                                = do
>                                      x <- item
>                                      item
>                                      y <- item
>                                      return (x,y)


Arithmetic Expression Structure

> data AOp
>      = Add | Mul | Sub
>      deriving Show

> data AExp
>      = Num Int
>      | Var String
>      | A AOp AExp AExp
>      deriving Show


Boolean Expression Structure

> data LOp
>      = Or | And
>      deriving Show
```

```
> data ROp
>     = Equ | Lt | Gt | Lte | Gte | Ne
>     deriving Show

> data BExp
>     = BVal Bool
>     | Not BExp
>     | R ROp AExp AExp
>     | L LOp BExp BExp
>     deriving Show


Commands Structure

> data Com
>     = Skip
>     | Print String
>     | Assign String AExp
>     | Seq Com Com
>     | Coms [Com]
>     | If BExp Com Com
>     | WhileDo BExp Com
>     | DoWhile Com BExp
>     | RepUntil Com BExp
>     | ForLoop (Com, BExp, Com) Com
>     deriving Show


Translation and Evaluation Structure

> type Loc = String

> data Ins = PUSH Int | ADD | MUL | SUB
>          | TRUE | FALSE | EQU | GRT | LST | GTE | LTE | AND | OR | NOT
>          | PRINT Loc
>          | LOAD Loc | STORE Loc | NOOP | BRANCH (Code, Code) | LOOP (
    Code, Code)
>          deriving Show

> type Code = [Ins]


Abstract Machine State Structure

> data ValZB      = Z Int | B Bool deriving Show
> type Stack      = [ValZB]
> type Binding    = (Loc, Int)
> type Storage    = [Binding]
> type Debug      = [Binding]
> type MState     = (Code, Stack, Storage, Debug)
```

```
Parsing Arithmitic Expression

> expr :: Parser AExp
> expr = do t <- term
>           do symbol "-"; e <- expr; return (A Sub t e)
>             +++ do symbol "+"; e <- expr; return (A Add t e)
>              +++ return t

> term :: Parser AExp
> term = do f <- factor
>           do symbol "*"; t <- term; return (A Mul f t)
>              +++ return f

> factor :: Parser AExp
> factor = do symbol "("
>             e <- expr; symbol ")"; return e
>             +++ number
>             +++ var

> number :: Parser AExp
> number = do x <- natural; return (Num x)

> var :: Parser AExp
> var = do v <- identifier; return (Var v)


Parsing Boolean Expression

> bexp :: Parser BExp
> bexp = (do t <- bterm; symbol "|"; e <- bexp;  return (L Or t e)) +++
    bterm

> bterm :: Parser BExp
> bterm = (do f <- bfactor; symbol "&"; t <- bterm; return (L And f t))
    +++ bfactor

> bfactor :: Parser BExp
> bfactor = nbval +++ bval

> bval :: Parser BExp
> bval = bpar +++ tval +++ rexp

> nbval :: Parser BExp
> nbval = do symbol "!"; f <- bfactor; return (Not f)

> tval :: Parser BExp
> tval = (do symbol "F"; return (BVal False)) +++ (do symbol "T"; return
     (BVal True))

> bpar :: Parser BExp
> bpar = do symbol "("; e <- bexp; symbol ")"; return e

> rexp :: Parser BExp
> rexp = eq +++ lt +++ gt +++ lteq +++ gteq +++ noteq
```

```
Parsing Relational Expression

> eq :: Parser BExp
> eq = do l1 <- expr; symbol "=="; l2 <- expr; return (R Equ l1 l2)

> lt :: Parser BExp
> lt = do l1 <- expr; symbol "<"; l2 <- expr; return (R Lt l1 l2)

> gt :: Parser BExp
> gt = do l1 <- expr; symbol ">"; l2 <- expr; return (R Gt l1 l2)

> lteq :: Parser BExp
> lteq = do l1 <- expr; symbol "<="; l2 <- expr; return (R Lte l1 l2)

> gteq :: Parser BExp
> gteq = do l1 <- expr; symbol ">="; l2 <- expr; return (R Gte l1 l2)

> noteq :: Parser BExp
> noteq = do l1 <- expr; symbol "!="; l2 <- expr; return (R Ne l1 l2)


Parsing Commands and Statements

> coms = do lst <- (sep0 comparse (symbol ";")); return (if length lst
    == 1 then head lst else Coms lst)

> seqcom :: Parser Com
> seqcom = do t <- comparse; symbol ";"; e <- seqcom; return (Seq t e)
>             +++ comparse

> comparse :: Parser Com
> comparse = skip +++ block +++ cblock +++ assign +++ printcom +++
    ifchoice +++ wdloop +++ dwloop +++ repuntil +++ forloop

> skip :: Parser Com
> skip = do symbol "skip"; return Skip

> assign :: Parser Com
> assign = do id <- identifier; symbol ":="; e <- expr; return (Assign
    id e)

> printcom :: Parser Com
> printcom = do symbol "print"; symbol "("; v <- identifier; symbol ")";
     return (Print v)

> block :: Parser Com
> block = do symbol "begin"; t <- coms; symbol "end"; return (t) +++
    comparse

> cblock :: Parser Com
> cblock = do symbol "{"; t <- coms; symbol "}"; return (t) +++ comparse

> ifchoice :: Parser Com
> ifchoice = do symbol "if"; b <- bexp; symbol "then"; ct <- comparse;
    symbol "else"; cf <- comparse; return (If b ct cf)
```

```
> wdloop :: Parser Com
> wdloop = do symbol "while"; b <- bexp; symbol "do"; ct <- comparse;
    return (WhileDo b ct)

> dwloop :: Parser Com
> dwloop = do symbol "do"; b <- comparse; symbol "while"; ct <- bexp;
    return (DoWhile b ct)

> repuntil :: Parser Com
> repuntil = do symbol "repeat"; b <- comparse; symbol "until"; ct <-
    bexp; return (RepUntil b ct)

> forloop :: Parser Com
> forloop = do symbol "for"; symbol "("; b <- assign; symbol ";"; ct <-
    bexp; symbol ";"; cf <- assign; symbol ")"; symbol "do"; c <-
    comparse; return (ForLoop (b, ct, cf) c)


Translate an Arithmetic Expression to Abstract Machine Code

> tr_a :: AExp -> Code
> tr_a (Num n)         = [PUSH n]
> tr_a (Var x)         = [LOAD x]
> tr_a (A Add a1 a2)   = tr_a a2 ++ tr_a a1 ++ [ADD]
> tr_a (A Sub a1 a2)   = tr_a a2 ++ tr_a a1 ++ [SUB]
> tr_a (A Mul a1 a2)   = tr_a a2 ++ tr_a a1 ++ [MUL]

Translate a Boolean Expression to Abstract Machine Code

> tr_b :: BExp -> Code
> tr_b (BVal True)     = [TRUE]
> tr_b (BVal False)    = [FALSE]
> tr_b (Not l1)        = tr_b l1 ++ [NOT]
> tr_b (R Equ l1 l2)   = tr_a l2 ++ tr_a l1 ++ [EQU]
> tr_b (R Gt l1 l2)    = tr_a l2 ++ tr_a l1 ++ [GRT]
> tr_b (R Lt l1 l2)    = tr_a l2 ++ tr_a l1 ++ [LST]
> tr_b (R Gte l1 l2)   = tr_a l2 ++ tr_a l1 ++ [GTE]
> tr_b (R Lte l1 l2)   = tr_a l2 ++ tr_a l1 ++ [LTE]
> tr_b (L And l1 l2)   = tr_b l1 ++ tr_b l2 ++ [AND]
> tr_b (L Or l1 l2)    = tr_b l1 ++ tr_b l2 ++ [OR]

Translate commands to abstract machine code

> tr_c :: Com -> Code
> tr_c (Coms (c:cs))  =  tr_c c ++ if length cs == 1 then tr_c (head cs)
     else tr_c (Coms cs)
> tr_c (Skip)          = [NOOP]
> tr_c (Print v)       = [PRINT v]
> tr_c (Assign m v)    = tr_a v ++ [STORE m]
> tr_c (If b c1 c2)    = tr_b b ++ [BRANCH(tr_c c1, tr_c c2)]
> tr_c (WhileDo b c)   = [LOOP (tr_b b, tr_c c)]
> tr_c (DoWhile c b)   = tr_c (Coms (c : [WhileDo b c]))
> tr_c (RepUntil c b) = tr_c (Coms (c : [WhileDo (Not b) c]))
> tr_c (ForLoop (c1, b, c2) c3) = tr_c (Coms (c1 : [WhileDo b (Coms (c3
    : [c2]))]))
```

```
Evaluate Abstract Machine Code (Single Step)

> amEval :: MState -> MState
> amEval (PUSH n:cs, e, s, d)            = (cs, Z n:e, s, d)
> amEval (ADD:cs, Z n1: Z n2:e, s, d)    = (cs, Z (n1 + n2):e, s, d)
> amEval (SUB:cs, Z n1: Z n2:e, s, d)    = (cs, Z (n1 - n2):e, s, d)
> amEval (MUL:cs, Z n1: Z n2:e, s, d)    = (cs, Z (n1 * n2):e, s, d)
> amEval (TRUE:cs, e, s, d)              = (cs, B True:e, s, d)
> amEval (FALSE:cs, e, s, d)             = (cs, B False:e, s, d)
> amEval (EQU:cs, Z n1: Z n2:e, s, d)    = (cs, B (n1 == n2):e, s, d)
> amEval (GRT:cs, Z n1: Z n2:e, s, d)    = (cs, B (n1 > n2):e, s, d)
> amEval (LST:cs, Z n1: Z n2:e, s, d)    = (cs, B (n1 < n2):e, s, d)
> amEval (LTE:cs, Z n1: Z n2:e, s, d)    = (cs, B (n1 <= n2):e, s, d)
> amEval (GTE:cs, Z n1: Z n2:e, s, d)    = (cs, B (n1 >= n2):e, s, d)
> amEval (AND:cs, B b1: B b2:e, s, d)    = (cs, B (b1 && b2):e, s, d)
> amEval (OR:cs, B b1: B b2:e, s, d)     = (cs, B (b1 || b2):e, s, d)
> amEval (NOT:cs, B b:e, s, d)           = (cs, B (not b):e, s, d)
> amEval (LOAD x:cs, e, s, d)            = (cs, Z (valueOf x s):e, s, d)
> amEval (STORE x:cs, Z z:e, s, d)       = (cs, e, update s x z, d)
> amEval (NOOP:cs, e, s, d)              = (cs, e, s, d)
> amEval (BRANCH (c1, c2):cs, B b:e, s, d)  = if b then (c1 ++ cs, e, s,
    d) else (c2 ++ cs, e, s, d)
> amEval (LOOP(c1, c2):cs, b, s, d)      = (c1 ++ BRANCH(c2 ++ [LOOP(c1,
    c2)],[NOOP]):cs, b, s, d)
> amEval (PRINT x:cs, e, s, d)           = (cs, e, s, (x,valueOf x s):d)
> amEval (c:cs, e, s, d)                 = (cs, e, s, d)
> amEval ([], e, s, d)                   = ([], e, s, d)


Utility Functions for Commands Separator

> sep1 :: Parser Com -> Parser a -> Parser [Com]
> sep1 p sep = do x <- p; xs <- many (sep >> p); return (x : xs)

> sep0 :: Parser Com -> Parser sep -> Parser [Com]
> sep0 p sep = sep1 p sep +++ return []


Utility Functions for Storage and Debug

> valueOf :: Loc -> Storage -> Int
> valueOf loc [] = error ("Unused input " ++ loc)
> valueOf loc ((key, val):rest)
>     | loc == key = val
>     | otherwise = valueOf loc rest

> update :: Storage -> Loc -> Int -> Storage
> update [] loc newVal = [(loc, newVal)]
> update ((key, val) : rest) loc newVal
>     | loc == key = (loc, newVal) : rest
>     | otherwise = (key, val) : update rest loc newVal
```

```
Interface Functions

> compile :: String -> Code
> compile = tr_c . parse_coms

> amInit :: Code -> MState
> amInit xs = (xs, [], [], [])

> eval :: MState -> MState
> eval ([], e, s, d) = ([], e, s, d)
> eval state = eval (amEval state)

> parseAExpr :: String -> AExp
> parseAExpr xs = case (parse expr xs) of
>     [(n,[])] -> n
>     [(_,out)] -> error ("Unused input " ++ out)
>     [] -> error "Invalid input"

> parseBExpr :: String -> BExp
> parseBExpr rstr = case parse bexp rstr of
>     [(n,"")] -> n
>     [(_, out)] -> error ("Unused input " ++ out)
>     [] -> error "Invalid input"

> parse_com :: String -> Com
> parse_com rstr = case parse seqcom rstr of
>     [(n,"")] -> n
>     [(_, out)] -> error ("Unused input " ++ out)
>     [] -> error "Invalid input"

> parse_coms :: String -> Com
> parse_coms rstr = case parse coms rstr of
>     [(n,"")] -> n
>     [(_, out)] -> error ("Unused input " ++ out)
>     [] -> error "Invalid input"

> amExec :: String -> MState
> amExec prog = eval (amInit (compile prog))

> amGetVal :: String -> String -> Int
> amGetVal var prog = result where
>         result  = valueOf var storage
>         (_,_,storage,_)   = eval (amInit (compile prog))

> amRun :: String -> IO ()
> amRun prog = mapM_ (putStrLn . show . snd) (debugList)
>     where
>         (_, _, _, debug) = eval (amInit (compile prog))
>         debugList = reverse debug
```