IIMP: Interactive IMPerative Language Parser

Ruggiero Altini

2020
January

1 Introduction

1.1 What is a parser?

A parser is a program that takes a string of characters as input, and produces some form of tree that makes the syntactic structure of the string explicit.

Parsers are an important topic in computing, because most real-life programs use a parser to preprocess their input (just like a calculator program parses numeric expressions prior to evaluating them).

1.2 Parser as function

The idiom in Haskell to define a parser is the following one:

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

where $Parser\, a$ is a type-constructor for a parser of generic type a.

The parser of some a here is a function that maps a string to a list of pairs of a and String. Or, as the rhyme by Dr Seuss says:

"A parser for things
Is a function from strings
To list of pairs
Of things and strings"

1.3 What is an interpreter?

An Interpreter in computing is a computer program that directly executes instructions written in a programming or scripting language, without requiring them to be previously compiled into a machine language program.

Generally an interpreter either parsers the source code and performs its behavior directly, or translates source code into some efficient intermediate representation and immediately executes it, assuming no precompilers involved.

1.4 IMP Language

The language implemented is composed of the classical constructs and control structures at the base of all imperative languages.

- **Assignment:** It assigns a value to a variable identified (univocally) by its name. To mimic classic IMP, variables treat only integer values, but the Haskell code provided in the project allows for a more easily extensibility of this.
- **Skip:** Instruction that does not change the state in any way nor changes the flow of execution; it's usually used when an instruction is expected.
- **if-then-else:** Classic conditional control structure: It checks a boolean expression and the flow of execution goes in the *then* block (the block from the first curly braces) if the condition is true, or in *else* otherwise. (The *else* block is optional)
- **while-loop:** Loop control structure

  In this documentation, the grammar used to describe this language is specified in the next section. An Haskell interpreter has been written to actually implement the grammar.

A (bad) grammar is here first supplied, to clarify how many operations and constructs are provided from IIMP (in inspiration to classic IMP). A better grammar for parsing (unambiguous) is presented in the following section to accommodate precedence of operators problems, and parentheses.

⟨*nat*⟩ ::= 0 | 1 | 2 | 3 | ...

⟨*integer*⟩ ::= [-] ⟨*nat*⟩

⟨*identifier*⟩ ::= ... (rules below)

⟨*aexp*⟩ ::= ⟨*nat*⟩ | ⟨*identifier*⟩ | ⟨*aexp*⟩ + ⟨*aexp*⟩ | ⟨*aexp*⟩ - ⟨*aexp*⟩ | ⟨*aexp*⟩ * ⟨*aexp*⟩

⟨*bexp*⟩ ::= True | False
| ⟨*aexp*⟩ '`=`' ⟨*aexp*⟩ | ⟨*aexp*⟩ '`<=`' ⟨*aexp*⟩
| '`!`'⟨*bexp*⟩
| ⟨*bexp*⟩ '`OR`' ⟨*bexp*⟩ | ⟨*bexp*⟩ '`AND`' ⟨*bexp*⟩

⟨*command*⟩ ::= skip'`;`' | ⟨*identifier*⟩ '`:=`' ⟨*aexp*⟩'`;`'
| ⟨*command*⟩ ⟨*command*⟩
| '`if`' ⟨*bexp*⟩ '`{`'⟨*command*⟩'`}`' ['`else`' '`{`'⟨*command*⟩'`}`']
| while ⟨*bexp*⟩ '`{`'⟨*program*⟩'`}`'

2 IIMP Grammar

BNF grammar used for this programming language interpreter shown below:

⟨*aexp*⟩ ::= ⟨*aterm*⟩ '`+`' ⟨*aexp*⟩
  | ⟨*aterm*⟩ '`-`' ⟨*aexp*⟩
  | ⟨*aterm*⟩

⟨*aterm*⟩ ::= ⟨*afactor*⟩ '`*`' ⟨*aterm*⟩ | ⟨*afactor*⟩

⟨*afactor*⟩ ::= '`(`'⟨*aexp*⟩'`)`' | ⟨*integer*⟩ | ⟨*identifier*⟩

⟨*bexp*⟩ ::= ⟨*bterm*⟩ '`OR`' ⟨*bexp*⟩ | ⟨*bterm*⟩

⟨*bterm*⟩ ::= ⟨*bfactor*⟩ '`AND`' ⟨*bterm*⟩ | ⟨*bfactor*⟩

⟨*bfactor*⟩ ::= '`True`' | '`False`'
  | '`!`'⟨*bfactor*⟩
  | '`(`'⟨*bexp*⟩'`)`'
  | ⟨*bcomparison*⟩

⟨*bcomparison*⟩ ::= ⟨*aexp*⟩ '`=`' ⟨*aexp*⟩ | ⟨*aexp*⟩ '`<=`' ⟨*aexp*⟩

⟨*program*⟩ ::= ⟨*command*⟩
  | ⟨*command*⟩ ⟨*program*⟩

⟨*command*⟩ ::= ⟨*assignment*⟩
  | ⟨*ifThenElse*⟩
  | ⟨*while*⟩
  | skip'`;`'

⟨*assignment*⟩ ::= ⟨*identifier*⟩ '`:=`' ⟨*aexp*⟩'`;`'

⟨*ifThenElse*⟩ ::= '`if`' ⟨*bexp*⟩ '`{`' ⟨*program*⟩ '`}`'
  | '`if`' ⟨*bexp*⟩ '`{`'⟨*program*⟩'`}`' '`else`' '`{`'⟨*program*⟩'`}`'

⟨*while*⟩ ::= while ⟨*bexp*⟩ ⟨*program*⟩

⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*nat*⟩ ::= ⟨*digit*⟩ ⟨*nat*⟩ | ⟨*digit*⟩

⟨*integer*⟩ ::= [-] ⟨*nat*⟩

⟨*identifier*⟩ ::= ⟨*lower*⟩ | ⟨*lower*⟩ ⟨*alphanum*⟩

⟨*alphanum*⟩ ::= ⟨*upper*⟩ ⟨*alphanum*⟩
| ⟨*lower*⟩ ⟨*alphanum*⟩
| ⟨*nat*⟩ ⟨*alphanum*⟩
| ⟨*upper*⟩ | ⟨*lower*⟩ | ⟨*nat*⟩

⟨*lower*⟩ ::= a-z

⟨*upper*⟩ ::= A-Z

In the most basic version of IIMP, the division operator (/), and $<$, $>$, $\geq$ comparison operators are not supported to keep the grammar much similar to IMP inspired from the book "The Formal Semantics of Programming Languages–An Introduction" by Glynn Winskel.

3 Environment

IIMP interpreter keeps things as simple as IMP can be; even though, it passively supports the adding of types in the program in a following extension.

*Variable*s are abstracted in a record of this form

```
data Variable = Variable {  name :: String,
                            vtype :: String,
                            value :: Int }
                              deriving Show
```

to structure the program in a nice and less error-prone form.

They feature a *name* (from which they are identified), a *type* (unused), and an integer *value*.

An Environment is thus with this abstraction, just a list of variables:

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

The Environment can be seen as a memory that can get updated (and read onto) along the program execution.

3.1 Environment management

The following code is employed to update–or read from–the environment.

updateEnv and *readVariable* make use of Parser monadic-style to be easily used inside the monadic parsing logic. Instead $modifyEnv$ and $searchVariable$ (used by the before discussed functions, respectively) act onto more basic collections/types.

```
-- Update the environment with a variable
-- If the variable is new (not declared before), it will be
   added to the environment
-- If the variable is existing, its value will be overwritten
    in.
updateEnv :: Variable -> Parser String
updateEnv var = P (\env input -> case input of
                     xs -> [((modifyEnv env var),"",xs)])

modifyEnv :: Env -> Variable -> Env
modifyEnv [] var = [var]
modifyEnv (x:xs) newVar = if (name x) == (name newVar) then [
   newVar] ++ xs
                          else [x] ++ modifyEnv xs newVar
```

```
-- Return the value of a variable given the name
readVariable :: String -> Parser Int
readVariable name = P (\env input -> case searchVariable env
   name of
    [] -> []
    [value] -> [(env, value, input)])

-- Search the value of a variable stored in the Env. given
   the name
searchVariable :: Env -> String -> [Int]
searchVariable []     queryname = []
searchVariable (x:xs) queryname = if (name x) == queryname
   then [(value x)]
                                  else searchVariable xs
                                     queryname
```

4 Parser: An inside look

As told in the introduction, a parser can be seen as a function that takes a string as input and produces a result of a generic type in the output.

The parser strategy employed is inspired to the one used in the book "Programming in Haskell– Second Edition" by Graham Hutton, which is based on a set of Parser-type monads.

The design progression of a parser can be summed up in this way.

We start with the basic parser that maps a string to a tree:

```
type Parser = String -> Tree
```

Since, in general, a parser might not always consume its entire argument string, we generalize our type for parsers to also return any unconsumed part of the argument string:

```
type Parser = String -> (Tree, String)
```

A parser may also not always suceed. In that case, we generalize our parser to return a list of results, with the convention that empty list denotes failure, and singleton list denotes success:

```
type Parser = String -> [(Tree, String)]
```

Since there can be different kinds of parser that return different kinds of trees, or more generally, any kind of value, we further generalize our parser to return generic types (as for example, a parser for arithmetic expressions may be interested in returning an Int at the end):

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

The last adding done to this design, is the one of adding the environment. As such, the environment as a memory allows us to do parsing with some initial state, and ending (hoopefully) with some final state:

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

We also switch from *type* to *newtype* (using P as dummy constructor), because otherwise we could not make the type-constructor an instance of Functor, Applicative, and Monad classes in Haskell language.

4.1 The Functor, the Applicative, and the Monad

To get advantage from the `do` notation and combine parsers in sequence, we need to make the parser type an instance of functor, applicative and monad.

Functor Below there is the definition to make the Parser a functor.

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

A functor is a mapping between categories, that preserves structure (function composition) and identity morphisms. Here we use `fmap` to apply a function to the result value fo a parser if the parser succeeds, and propagate the failure otherwise.

```
> parse (fmap toUpper item) [] "abc"
[([],'A',"bc")]

> parse (fmap toUpper item) [] ""
[]
```

Applicative The Parser type can then be made into an applicative functor as follows

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

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

In this case, `pure` transforms a value into a parser that always succeeds with this value as result, without consuming any of the input string:

```
> parse (pure 1) [] "abc"
[([],1,"abc")]
```

In turn, `<*>` applies a parser that returns a function to a parser that returns an argument to give a parser that returns the result of applying the function to the argument, and only succeeds if all the components succeed.

Monad Below there goes the definition to make the Parser a Monad.

```
instance Monad Parser where
    -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    p >>= f = P (\env input -> case parse p env input of
                                []         -> []
                                [(env, v,out)] -> parse (f v)
                                    env out)
```

That is, the parser `p >>= f` fails if the application of the parser `p` to the input string `input` fails, and otherwise applies the function `f` to the result value `v` to give another parser `f v`, which is then applied to the original output string `out` that was produced by the first parser to give the final result.

Now we can use the `do` notation to sequence parsers and process their result values.

Recall that `return` is just a function alias to the applicative function `pure` that returns a parser that always succeeds.

4.2 ... and the Alternative

Besides combining parsers in sequence with `do`, we also got a "trick" to combine parsers simulating choices.

That is, we apply one parser to the input string, and if this fails to then we apply another to the same input instead.

There are principally two primitives (functions) to consider, and those are `empty` and `<|>`. The intuition is that `empty` represents an alternative that has failed, and `<|>` an appropriate choice operator for the type.

They are required to satisfy the following identity and associativity laws:

```
empty <|> x     = x
x <|> empty     = x
x <|> (y <|> z) = (x <|> y) <|> z
```

In our parser type, `empty` is the parser that always fails regardless of the input string, and `<|>` is a choice operator that returns the result of the first parser if it succeeds on the input, and applies the second parser to the same input otherwise:

```
instance Alternative Parser where
    -- empty :: Parser a
    empty = P (\env input -> [])

    -- (<|>) :: Parser a -> Parser a -> Parser a
    p <|> q = P (\env input -> case parse p env input of
                                [] -> parse q env
                                   input
                                [(env,v,out)] -> [(env, v,out
                                   )]
```

4.3 Arithmetic Parsing

Arithmetic Parsing is divided into 3 sub-parsers. `aexp` is the main one, and manages additions, subtractions, or the expansions of `aterm`. `aterm` manages the multiplication between an `afactor` and another `aterm`.

`afactor` manages the recursion on `aexp` surrounded by parentheses, or either the reading of a variable (from its identifier), or parsing just an integer.

```
aexp :: Parser Int
aexp = (do t <- aterm
           symbol "+"
           a <- aexp
           return (t+a))
        <|>
       (do t <- aterm
           symbol "-"
           a <- aexp
           return (t-a))
        <|>
       aterm

aterm :: Parser Int
aterm = do { f <- afactor
           ; symbol "*"
           ; t <- aterm
           ; return (t * f)
            }
            <|>
            afactor

afactor :: Parser Int
```

```
afactor = (do symbol "("
              a <- aexp
              symbol ")"
              return a)
            <|>
          (do i <- identifier
              readVariable i)
            <|>
          integer
```

4.4 Boolean Parsing

Boolean Parsing is addressed in a similar fashion. The parsing phase is divided this time into four different sub-parsers: `bexp`, `bterm`, `bfactor` and `bcomparison`.

To remove ambiguity, AND operator gets precedence on OR operator. In `bcomparison`, `aexp` parser gets also used to evaluate arithmetic expressions if nested in the boolean exp.

```
bexp :: Parser Bool
bexp =  (do b0 <- bterm
            symbol "OR"
            b1 <- bexp
            return (b0 || b1))
        <|>
        bterm

bterm :: Parser Bool
bterm = (do f0 <- bfactor
            symbol "AND"
            f1 <- bterm
            return (f0 && f1))
        <|>
        bfactor

bfactor :: Parser Bool
bfactor = (do symbol "True"
              return True)
          <|>
          (do symbol "False"
              return False)
          <|>
          (do symbol "!"
```

```
                b <- bfactor
                return (not b))
            <|>
            (do symbol "("
                b <- bexp
                symbol ")"
                return b)
            <|>
            bcomparison

bcomparison :: Parser Bool
bcomparison = (do a0 <- aexp
                  symbol "="
                  a1 <- aexp
                  return (a0 == a1))
                <|>
              (do a0 <- aexp
                  symbol "<="
                  a1 <- aexp
                  return (a0 <= a1))
```

4.5 Command Parsing

4.5.1 Commands and sequences of commands

Commands (or sequences of) are intuitively parsed in this way:

```
program :: Parser String
program = (do command
              program)
          <|>
          command

command :: Parser String
command = assignment
           <|>
          ifThenElse
           <|>
          while
           <|>
          (do symbol "skip"
              symbol ";")
```

where `assignment`, `ifThenElse`, `while` are sub-parsers that will be presented below.

Skip command is implemented right here, as an instruction that gets just parsed and does not do anything. It can be useful as a placeholder for an empty program, even just as a sub-program that should be used for an empty block.

For example, here it is a never-ending program in IIMP:

```
IIMP> while True { skip; }
```

4.5.2 Assignments

Assignments in IIMP "only" involve identifiers and arithmetic expressions. The syntax is pretty simple, with := as notation of *assigning* the value, and a semicolon to end the statement.

Variables are not declared but just assigned. Trying to read a never assigned variable will throw a memory error.

```
assignment :: Parser String
assignment = do x <- identifier
                symbol ":="
                v <- aexp
                symbol ";"
                updateEnv Variable{name=x, vtype="", value=
                    v}
```

4.5.3 Conditional Statements

Conditional statements (`if-then-else`) in IIMP do not need parentheses around the boolean condition, and else's are not mandatory. Once a block is written, at least an instruction has to be present in order to have correct semantic. If block is expected to be empty (for some reason), `skip;` command will do just fine.

If the condition is not met, the block inside the then condition (first curly braces) will be parsed without being executed, while the else block (if present) will be parsed and executed; if the condition is met, the block inside the then condition will be parsed and executed, and the block in the else condition will be parsed but not evaluated.

```
ifThenElse :: Parser String
ifThenElse = (do symbol "if"
                 b <- bexp
                 symbol "{"
                 if (b) then
                     (do program
                         symbol "}"
                         (do symbol "else"
                             symbol "{"
                             parseProgram;
                             symbol "}"
                             return "")
                            <|>
                            (return ""))
                 else
                     (do parseProgram
                         symbol "}"
                         (do symbol "else"
                             symbol "{"
                             program
                             symbol "}"
                             return "")
                          <|>
                          return "")
                    )
```

4.5.4 Loop Conditional Statements

The only conditional statement that loops in IIMP is the while-statement. It does not need parentheses around the boolean condition. If the boolean condition is met (True), it parses and evaluates the block, adds the block back to unparsed string, and it calls another while parser to evaluate the expression; when the boolean condition is not met anymore, it parses without evaluating the block, and it returns a successful parser.

```
while :: Parser String
while = do w <- consumeWhile
           repeatWhile w
           symbol "while"
           --symbol "("
           b <- bexp
           --symbol ")"
```

```
            symbol "{"
            if (b) then
                (do program
                    symbol "}"
                    repeatWhile w
                    while)
            else
                (do parseProgram
                    symbol "}"
                    return "")

repeatWhile :: String -> Parser String
repeatWhile c = P(\env input -> [(env, "", c ++ input)])
```

4.5.5 Example of input consumption

Since the assignment is the biggest (if only) side-effect action in an imperative language, the code `consumeAssignment` is shown. It is one of the parsers that just consume the input without doing evaluation (and as such, it does not alter the working memory):

```
consumeAssignment :: Parser String
consumeAssignment = do x <- identifier
                       symbol ":="
                       a <- consumeAexp
                       symbol ";"
                       return (x ++ ":=" ++ a ++ ";")
```

5 Running IIMP

IIMP Interpreter can be run either from GHCi (in context of IIMP.hs file), calling the `main` function, or it can be run from a compiled executable. To compile the source code into an executable, you can do in the console:

```
$ ghc --make IIMP.hs
```

and run it from the OS.

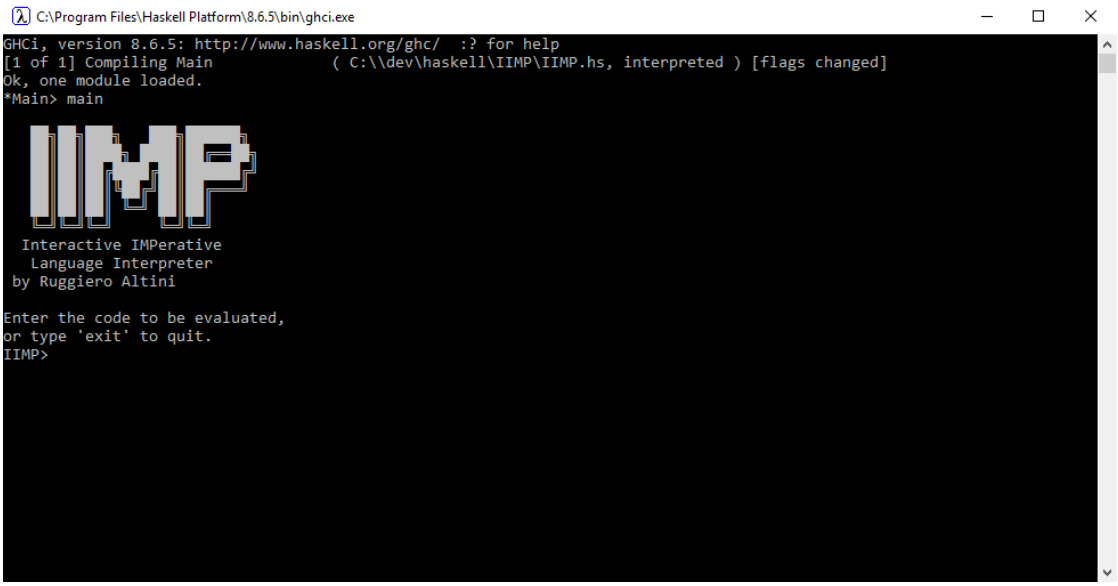

Figure 1: IIMP just started

5.1 The conservation of variables

Variables are kept between program runs in the same instance of IIMP, as the `env` (Environment) is not discarded, but passed over as the initial state of further parsings.

You can though use `clear` to push an empty environment (no variables) for the next execution.

5.1.1 Examples of programs

Iterate over 10 Program Input:

```
x := 1; while x <= 10 { x := x + 1; }
```

Output:

```
Parsed code:

 x:=1;while x<=10 {x:=x+1;}

Memory:
```

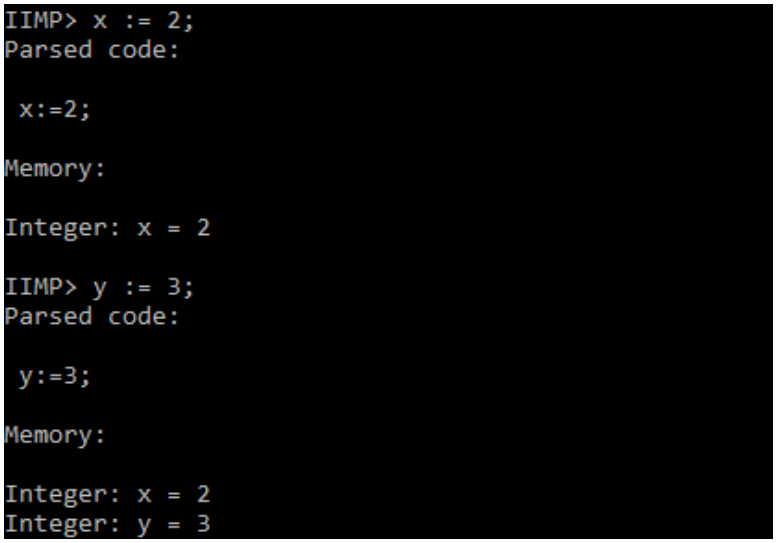

Figure 2: Variables are preserved across runs

```
Integer: x = 11

IIMP>
```

Factorial Program Input:

```
n := 7; fact := 1; while !(n <= 0) { fact := fact * n; n := n-1; }
```

Output:

```
Parsed code:

 n:=7;fact:=1;while !(n<=0) {fact:=fact*n;n:=n-1;}

Memory:

Integer: n = 0
Integer: fact = 5040

IIMP>
```