Constructing Parsers from Programming Language BNF Grammars

Joshua Van Leeuwen

Backus-Naur Form

All languages follow a set of rules which dictate how the contents of the language can be used and how the language is structured. The set of rules which govern the language is called the grammar. Just like natural language, programming languages also follow a grammar. However, these languages are always formal languages meaning they have a formal grammar consisting of a finite set of symbols and a finite set of rules that dictate it. These grammars are context-free meaning they are represented by a set of production rules which describe all the possible strings of the programming language. An example of a simple context-free grammar is one which describes all of the strings which form well nested parenthesis and square brackets over the alphabet $\Sigma = \{(,), [,], \varepsilon\}$

$S \rightarrow (S)$
$S \rightarrow [S]$
$S \rightarrow SS$
$S \rightarrow \varepsilon$

Non-terminals are used to define place holders that become patterns of terminals that are generated by the non-terminals. Here S is the only non-terminal which is also the start symbol. Non-terminals can be generated by themselves which gives context-free grammars the power of recursion so a finite grammar can define an infinite set of valid strings belonging to the language.

In order to better represent context-free grammars a notation was created called Backus-Naur form (BNF) which was first introduced by John Backus and Peter Naur when they developed the description for the ALGOL 60 programming language in 1960. This notation and its extensions has become the standard way of specifying the grammar of programming languages.

BNF is a formal notation which is used to define grammars in a way which is more readable for humans. The structure of BNF is a set of rules which have the form of a tag which is a non-terminal that is inside angled brackets, followed by '::=' and a rule involving terminals and non-terminals. Just like production rules, BNF can also use recursion using non-terminals giving the notation as much power. A simple BNF to define a digit:
<digit> ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

The symbols and meaning for BNF are as follows:

Symbol	Meaning
<expr>	Things inside '<>' are non-terminals. These refer to the tags mentioned which are labelled. In this case labelled expr.
“1”	Things inside “” quotations are terminals. This means that this explicitly states that the symbol 1 should occur at that point.
::=	This means that the non-terminal on the left hand side is defined by what is on the right hand side of this symbol.
|	The bar, meaning 'or', is used in order to make a choice in the productions. | provides complexity in the grammar to build complex syntax.

With the use of these symbols a grammar tree can be formed of multiple definitions of different non-terminals. By creating this abstraction, a simple and easy to understand representation of part of

some syntax tree can be made of a complex grammar. A simple example of a BNF using recursion would be that of addition of numbers.

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

Here you can see that this language would accept strings like:
“1”, “4+0+5”
but not of:
“+3+4”, “+”, “”, etc.

Extended BNF (EBNF) also exists which is a collection of extensions to BNF but keeps all the same power. EBNF uses less redundant notation and condenses notation to describe the same rules in standard BNF and so greatly increases the readability of a grammar by a human. Some of the extensions include grouping denoted by parenthesis that define precedence of rules, square brackets to denote the contents are optional and also repetition, denoted by curly-braces around an expression which means that expression can be repeated zero or more times. Repetition is very useful in cutting down long rules defined in BNF increasing readability:

BNF

<signedInt> ::= “-” <number> | <number>
<number> ::= <digit> <number> | <digit>

EBNF

<signedInt> ::= [“-”] <digit> {<digit>}

Lexical Analysis

With a BNF of a language defined a parser can be developed which consumes an input string which then this is mapped to data inside a computer to build a data structure that follows the grammar of the language and can check the input string is valid. The data structure which is the target domain of the parser may be of a different form than the input string however the syntax and meaning of the output is the exact same in the scope of the language.

To consume the input stream, lexical analysis is used to transform the input stream into corresponding tokens forming an abstract syntax tree to be used by the parser. This stage ensures that the input stream has correct syntax in terms of spelling and use of symbols so can return an error if needed. This stage will also remove whitespace in the source code or comments if they are supported by the language. The lexical analyser will consume the input stream a character at a time so its efficiency is vital to the efficiency of the parsing process as a whole.

Although lexical analysis is normally fully completed on the input stream before parsing of the tokens is done, it can also be implemented alongside the parser so that both processes happen iteratively together when parsing an input stream.

Parsers

When the input stream has been converted into tokens by the lexical analyser, it is then the job of parsers to consume these and map them onto data using rules. Parsers which do this are deeply tied to the language the compiler it is written in. The example parser will be written in Haskell and so parser combinators will be used in this solution. Parser combinators are high order functions which consume some input stream and convert these into a target data type. Parser combinators work by parsers consuming an input stream and then producing new data types which will be then input into another parser.

To capture the notion of a parser which can succeed or fail however still returns the rest of the input stream a new data type `Parser` is made:

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

The parser data type has an input stream of type string that will output a list of tuples consisting of the target of the parser and a string that is the rest of the unconsumed input stream. By returning the unconsumed part of the input stream it is possible to form combinators to build parsers that consume other parser outputs. By returning a list gives the ability to have different options for the result chosen according to the grammar which is defined in the BNF of the language. By using the Maybe type constructor a parser can result in Nothing if it fails.

In this notion, a library of parsers and combinators are written to build the target structure which can be directly mapped to the grammar. One of the fundamental functions needed for the library to represent grammars is one in which always succeeds but does not consume any of the input stream; produce.

```
produce :: a -> Parser a
produce x = Parser (\ts -> Just [(x, ts)])
```

The 'x' refers to the target value we want to produce.
The 'ts' is the input stream which is also the output stream since produce does not consume any input.
The '\' denotes λ-abstraction which is an anonymous function. These are useful when passing them as parameters to higher order functions. This is very useful for parsers created in functional programming languages as an input stream can be used with the λ-abstraction. A trivial example of how an anonymous function can be used:

```
f x = x+x
```

can be written and is equivalent to

```
f = \x -> x+x
```

In order to initiate a parser to operate on an input stream, a function parse is defined:

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

The function parse will lift the given parser from its parser data type and revelling its contents. Using parse with produce will output the target and remaining input stream of the parser:
>parse (produce 0) “hello” = Just [(0, “hello”)]

In order to differentiate code written in the Haskell program and the results of some input into the compiled program, '>' will donate the following text is input into the current program made by the code defined so far, and '=' will denote the end of input and the following text after is the output of the program.

Conversely, another important fundamental function is a parser which always fails its input stream and returns Nothing, this is called failure and defined:

```
failure :: Parser a
failure = Parser (\ts -> Nothing)
```

>parse failure “123” = Nothing

A final fundamental parser needed is one in which consumes and returns the next character in the input stream unconditionally. This parser is needed in order to read each character in the input string return it to a parser to be tested.

```
item :: Parser Char
item = Parser (\ts -> case ts of
      [] -> Nothing
      (x:xs) -> Just [(x,xs)])
```

>parse item “hello” = Just [(“h”, “ello”)]

The item parser is taking in an input string then successfully returns the first character and then the rest of the input stream if some is left. It will output Nothing if an empty string is input.
>parse item “” = Nothing

Parser Combinators

With some fundamental parsers defined, an input stream can now be parsed to a target data structure however only one parser can be performed and so cannot define any non-trivial grammar. In order to achieve the concept of chaining parsers together to sequentially operate on some input stream to match a grammar, parser combinators are needed. Parser combinators are higher-order functions that consume multiple parsers and returns one. First the data type Parser must be an instance of Functor so that `fmap` can be used with functions over parser data types. The functor class is defined by:

```
class  Functor f  where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  (<$) =  fmap . Const
```

In the functor class, `fmap` is defined as a function that behaves the same as map by taking a function parameter however, it is applied to each part of the second parameter that is a functor according to the context of that functor, not just a list structure that `map` takes, then returns this result. '`(<$)`' is a function that takes an input and a functor then applies the functor to the first parameter returning the first parameter that is now of the functor type. The second parameter is discarded which is useful in parsing when used for lexical analysis to create a token and discard the corresponding characters from in input string just consumed.
>10 <$ Just “input” = Just 10

The instance of Functor for Parser data types will be defined:

```
instance Functor Parser where
  fmap f p = Parser (\ts -> case parse p ts of
    Nothing -> Nothing
    Just [(x, ts')] -> Just [((f x), ts')])
```

If the result of the parser `p` on the input stream fails it returns `Nothing` therefore `fmap` also returns `Nothing`. If the parser is successful then the function `f` is applied to the results of the parsed input stream.

Now the parser instance of Functor is defined, an instance for the Applicative class can be defined for the parser type. An applicative is the next extension to a functor as it gives the ability to operate on a value which is wrapped by a functor however the function applied is also wrapped in a functor. The applicative class is defined:

```
class (Functor f) => Applicative f where
  pure     :: a -> f a
  (<*>)    :: f (a -> b) -> f a -> f b
  (*>)     :: f a -> f b -> f b
  a1 *> a2 = (id <$ a1) <*> a2
  (<*)     :: f a -> f b -> f a
  a1 <* a2 = a1 <*> (id <$ a2)
```

The `pure` function simply transforms the given value into the given functor by applying the appropriate identity constructor of that functor. The `(<*>)` extracts the functions which is wrapped in a functor and is mapped over the second functor's values:
>[(+5), (*10)] <*> [0, 1] = [5,6,0,10]
`(*>)` and `(<*)` will sequentially run the their first and second arguments however will discard the first or second argument respectively when completed.

The instance of Applicative for Parser will be defined:

```
instance Applicative Parser where
  pure a = Parser (\ts -> Just [(a, ts)])
  p <*> q = Parser (\ts -> do
          [(f,ts')] <- parse p ts
          [(x,ts'')] <- parse q ts'
          Just [(f x, ts'')])
```

The `pure` method will simply wrap the given parameter into the Parser data type and is given the input stream. Here the `<*>` combinator will parse the first argument then the second. The first parsers resulting target is then applied to the second parsers result.

With the Parser instance for applicative being defined this means that `<$>` can be used as an infix operator instead of `fmap`:
`f <$> x` is equivalent to `fmap f x`
This is useful when writing parsers as it increases the readability of the code and helps to display the grammar in much the same way the BNF of the language is displayed.

Bind is used to attach parsers together so that a parser's input can be that of the result of another parser. This is an important tool as it achieves the combining of multiple parsers together from the library of parsers to act on an input stream. Bind '`(>>=)`' belongs to the monad class that is defined:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  x >> y = x >>= \_ -> y
  return :: a -> m a
  fail :: String -> m a
```

The `(>>)` function sequentially performs the first parameter and then the second however discards the first result. The `fail` is used to return something if the monad fails. The `return` will take an argument and return it wrapped in a monad. Bind will operate the first parameter and then passes any result made as a parameter to the second parameter and then is also operated.
All monad instances must satisfy the following rules:
Left identity holds : return a >>= k == k a
Right identity holds : m >>= return == m
Associativity : m >>= (\x -> k x >>= h) == (m >>= k) >>= h

To be able to use bind in the parsers a new instance of Monad is defined:

```
instance Monad Parser where
  p >>= f = Parser (\ts -> case parse p ts of
        Nothing -> Nothing
        Just [(x,ts')] -> parse (f x) ts')
```

Bind will attempt to parse the first parameter, if successful then the second argument is applied to the result and this is then parsed with the rest of the input stream. If parsing of the first argument fails then `Nothing` is returned.

In order to implement rules that appear in BNF, parsers must be able to make choices depending on their input which directly corresponds to the '|' symbol in BNF notation. A new class called Choice that is a monoid on applicative functors will be defined to handle choice:

```
class Applicative f => Choice f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

The `empty` denotes the identity of `<|>`, a function that returns one of its two functor parameters depending on its instance definition. A Parser instance of this class is added to handle choice for

parsers:

```
instance Choice Parser where
  empty = failure
  px <|> py = Parser (\ts -> case parse px ts of
          Nothing -> parse py ts
          xs -> xs)
```

The identity for this instance is failure, a parser that always fails defined earlier. The choice symbol has two parameters which are parsers. The first parser is first run on the input stream which if succeeds, its result is retuned however if fails, the second parameter parser is run on the same input stream and returned.
This gives parsers the ability of choice defined in BNF grammars as they can try to parse an input stream with multiple parsers until one succeeds.

Lexer

In order to read in the input string to form tokens that are fed to the parsers, a small group of functions will be made. This process of forming tokens will be done together with the library of parsers when parsing an input string instead of before parsing starts. The item parser which was defined earlier is the parser which will read each character of the input string one by one and return it to be tested on.

In order to perform tests on the input stream that is being consumed, a function is needed to evaluate these tests and return the result. This is important for making sure the input stream's symbols are matching according to the BNF grammar.

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = item >>= \x ->
        if p x then produce x else failure
```

This function will receive a boolean expression which is to be matched of the form of a character and `'=='`. It will then call `item` to receive the next character in the input stream, this is then tested with the received expression with the use of bind. If the match was successful the parser will return the produce of the character else will return a failure with no match.

With `satisfy` defined, another combinator is one which is used to match a character.

```
char :: Char -> Parser Char
char x = satisfy (x ==)
```

This parser simply receives a character and calls `satisfy` to match it.
>parse (char 'h') “hello” = Just [('h',”ello”)]
>parse (char 'x') “hello” = Nothing

Now another combinator can be used to test for entire strings using the `char` function.

```
string :: String -> Parser String
string "" = produce ""
string(x:xs) = char x >>= \x' ->
               string xs >>= \xs' ->
               produce (x:xs)
```

This parser takes a string parameter denoted by a header character and a tail of characters. The function `char` is then called with the header and if successful, using bind, `string` is called by itself on the rest of the string input. Once the input string has been fully checked successfully, `string` will be called with an empty string ending the recursion and the target will be produced along with the unconsumed string. If the input string fails at any point during matching then it returns failure.
>parse (string “hello”) “hello world” = Just [(“hello”,” world”)]

In most programming languages, apart from an initial space character to explicitly separate tokens,

whitespace is ignored when the code is compiled and so must be ignored during parsing. A parser can be made to deal with this whitespace:

```
whitespace :: Parser ()
whitespace = many (oneOf " \t") *> produce ()
```

`whitespace` will parse zero or more space characters from the input stream and once `many` fails, will return nothing, discarding the spaces from the input stream.

With this set of functions a token combinator can be made which uses them together to gather tokens from the input string.

```
tok :: String -> Parser String
tok t = string t <* whitespace
```

This function will take in an input string and test it against the input stream which if successful, will also consume whitespace and return the parsed result.

Tool Parsers

As well as a set of lexer functions, a set of combinators that can define various features of a BNF are needed. Grammars represented in BNF use recursion when a non-terminal has a rule which is self calling. In EBNF, repetition of some expression is expressed by containing it in curly-braces which denotes that expression can be repeated zero or more times. In order to capture this inside a parser library two functions are created; 'many' denoting zero or more repetitions and 'some' denoting one or more repetitions.

```
many :: Parser a -> Parser [a]
many x = some x <|> produce []

some :: Parser a -> Parser [a]
some x = (:) <$> x <*> many x
```

The many function will take in a parser which attempts to pass to the some function. If some fails then the produce parser is called because of the choice operator. The some function also receives a parser which is executed and concatenated with the rest of the results it has gathered so far and then many is called with that parser. This loop of many and some will keep repeating until some fails and many will call produce.

In BNF grammars, non-terminals are often defined by a long rule of many terminals separated by the '|' symbol such as a non-terminal for all valid characters. When creating a parser for one of these types of non-terminals it is useful to have a tool which helps to find and match an input character with an element of a list of characters.

```
oneOf :: [Char] -> Parser Char
oneOf cs = satisfy (\c -> elem c cs)
```

This parser oneOf will take a list of characters and uses the parser satisfy with the function elem to match the test the next character in the input string with one from the list.

```
noneOf :: [Char] -> Parser Char
noneOf cs = satisfy (\c -> not(elem c cs))
```

This is simply the opposite of the oneOf parser to succeed when the character is not present in the list.

Creating the parser

With the previous tools defined a grammar represented by an extended BNF can now be converted into a Haskell program which parses a valid input string. With the use of data constructors and the parser combinators defined, the code of the parser will be very readable and will closely resemble the BNF rules. This is a simple grammar of a subset of Tiny BASIC programming language which will be the target grammar for the parser.

```
Prog = {line}
```

```
line    = cmnd cr | stmt cr | number stmt cr
cmnd = CLEAR | LIST | RUN
stmt = PRINT args
     | IF expr rel expr THEN stmt
     | GOTO expr
     | INPUT vars
     | LET var '=' expr
     | GOSUB expr
     | RETURN
     | END
args = ( string | expr ) {',' ( string | expr )}
vars = var {',' var}
expr = ['+'|'-'] term {('+'|'-') term}
term = fact {('*'|'/') fact}
fact = var | number | '(' expr ')'
var = 'A' | 'B'  | ... | 'Z'
number = digit {digit}
digit = '0' | '1' | ... | '9'
rel = '<' [ '>' | '=' ] | '>' [ '<' | '=' ] | '='
string = '"' {achar} '"'
achar = any character except '"' or '\n' or '\r'
```

When developing a parser it is beneficial to begin to build the library with the terminals of the grammar as this makes testing easier as the program is being written. Firstly a character of the grammar needs a parser.

```
achar :: Parser Char
achar = noneOf("\"\n\r");
```

And a parser for string which is defined as zero or more characters inside quotations.

```
str :: Parser String
str = tok "\"" *> many achar <* tok "\""
```

Numbers are defined as a `digit` followed by zero or more digits.

```
digit :: Parser Char
digit = oneOf ['0' .. '9'] <* whitespace

number :: Parser Int
number = ((some digit) >>= return . read) <* whitespace
```

The parser for number uses `return . read` which converts the string representing the number to an integer type. Whitesapce is also removed from the right of the input stream.

The `rel` non-terminal can be simply expressed by a new data type which has a set of symbols for possible values and a parser to return the correct symbol according to the input stream.

```
data Rel     =  (:<:) | (:<=:) | (:<>:) | (:=:) | (:>:) | (:>=:)
rel :: Parser Rel
rel = ((:<>:)  <$ tok "<>")
  <|> ((:<>:)  <$ tok "><")
  <|> ((:=:)   <$ tok "=")
  <|> ((:<=:)  <$ tok "<=")
  <|> ((:<:)   <$ tok "<")
  <|> ((:>=:)  <$ tok ">=")
  <|> ((:>:)   <$ tok ">")
```

This method of implementing each parser for each grammar rule and creating a new data type for non-terminals can be continued throughout the entire grammar. This is code for the grammar and parsers of the this language.

```
data Prog  =  Prog [Line]                    data Cmnd  =  CLEAR | LIST | RUN
data Line  =  Cmnd Cmnd | Stmt Stmt | Line   data Stmt  =  PRINT Args
Int Stmt                                               |  IF Expr Rel Expr Stmt
```

```
          |  GOTO Expr
          |  INPUT [Ident]
          |  LET Ident Expr
          |  GOSUB Expr
          |  RETURN
          |  END
type Args    =  [Either String Expr]
data Parity  =  POS | NEG
data Expr    =  Expr Parity Term [Exprs]
data Exprs   =  (:+:) Term | (:-:) Term
data Term    =  Term Fact [Terms]
data Terms   =  (:*:) Fact | (:/:) Fact
data Fact    =  Var Ident | Number Int |
Parens Expr
data Rel     =  (:<:) | (:<=:) | (:<>:) |
(:=:) | (:>:) | (:>=:)
type Ident   =  Char

achar :: Parser Char
achar = noneOf("\"\n\r.")

str :: Parser String
str = tok "\"" *> many achar <* tok "\""


digit :: Parser Char
digit = oneOf ['0' .. '9'] <* whitespace

number :: Parser Int
number = ((some digit) >>= return . read) <*
whitespace

rel :: Parser Rel
rel = ((:<>:)  <$ tok "<>")
  <|> ((:<>:)  <$ tok "><")
  <|> ((:=:)   <$ tok "=")
  <|> ((:<=:)  <$ tok "<=")
  <|> ((:<:)   <$ tok "<")
  <|> ((:>=:)  <$ tok ">=")
  <|> ((:>:)   <$ tok ">")

var :: Parser Ident
var = oneOf ['A' .. 'Z'] <* whitespace

fact :: Parser Fact
fact = (Var <$> var)
  <|> (Number <$> number)
  <|> (Parens <$ tok "(" <*> expr <* tok
")")

expr :: Parser Expr
expr = Expr <$> ((POS <$ tok "+") <|> (NEG
<$ tok "-") <|> pure POS)
  <*> term <*> many exprs

exprs :: Parser Exprs
exprs = ((:+:) <$ tok "+" <*> term)
    <|> ((:-:) <$ tok "-" <*> term)

term :: Parser Term
term = Term <$> fact <*> many terms

terms :: Parser Terms
terms = ((:*:) <$ tok "*" <*> fact)
  <|> ((:/:) <$ tok "/" <*> fact)


vars :: Parser [Ident]
vars = sepBy1 var (tok ",")

args :: Parser Args
args = sepBy1 ((Left <$> str) <|> (Right <$>
expr)) (tok ",")

stmt :: Parser Stmt
stmt = (PRINT   <$ tok "PRINT" <*> args)
   <|> (IF      <$ tok "IF" <*> expr <*> rel
<*> expr <* tok "THEN" <*> stmt)
   <|> (GOTO    <$ tok "GOTO" <*> expr)
   <|> (INPUT   <$ tok "INPUT" <*> vars)
   <|> (LET     <$ tok "LET" <*> var <* tok
"=" <*> expr)
   <|> (GOSUB   <$ tok "GOSUB" <*> expr)
   <|> (RETURN  <$ tok "RETURN")
   <|> (END     <$ tok "END")

cr :: Parser [Char]
cr = many (oneOf "\r\n")


cmnd :: Parser Cmnd
cmnd = (CLEAR  <$ tok "CLEAR") <|> (LIST <$
tok "LIST") <|> (RUN <$ tok "RUN")

line :: Parser Line
line = (Cmnd  <$> cmnd <* cr) <|> (Stmt  <$>
stmt <* cr) <|> (Line  <$> number <*> stmt
<* cr)

prog :: Parser Prog
prog = Prog <$> many line
```

To fully test the parser on some input stream the console cannot be used as input as the grammar of the language involves carriage returns and so will read from a file instead. First a new function is defined which wraps over the current top parsing function so that it can return the result or an error if parsing of the input stream fails completely.

```
runParser :: Parser a -> String -> a
     runParser m s =
      case parse m s of
           Just [(res, [])] -> res
           Just [(_, rs)]   -> error rs
           _      -> error "Parser error."
```

Now this function can be used in a parse file function that reads an input stream from file.

```
parseFile :: FilePath -> IO ()
parseFile filePath = do
  file <- readFile filePath
```

```
        putStrLn (show(runParser prog file))
```

Calling parseFile with the following text;

```
40 LET N = (100)
50 PRINT "GUESS A NUMBER?"
60 INPUT G
70 LET C = C+1
80 IF G=N THEN GOTO 110
90 IF G>N THEN PRINT "LOWER"
100 IF G<N THEN PRINT "HIGHER"
110 GOTO 50
120 PRINT "YOU GUESSED IT IN", C, " TRIES!"
END
```

Outputs;

> parsefile “test.tiny”

Prog [Line 40 (LET 'N' (Expr POS (Term (Parens (Expr POS (Term (Number 100) []) [])) []) [])),Line 50 (PRINT [Left "GUESS A NUMBER?"]),Line 60 (INPUT "G"),Line 70 (LET 'C' (Expr POS (Term (Var 'C') []) [(:+:) (Term (Number 1) [])])),Line 80 (IF (Expr POS (Term (Var 'G') []) []) (:=:) (Expr POS (Term (Var 'N') []) []) (GOTO (Expr POS (Term (Number 110) []) []))),Line 90 (IF (Expr POS (Term (Var 'G') []) []) (:>:) (Expr POS (Term (Var 'N') []) []) (PRINT [Left "LOWER"])),Line 100 (IF (Expr POS (Term (Var 'G') []) []) (:<:) (Expr POS (Term (Var 'N') []) []) (PRINT [Left "HIGHER"])),Line 110 (GOTO (Expr POS (Term (Number 50) []) [])),Line 120 (PRINT [Left "YOU GUESSED IT IN",Right (Expr POS (Term (Var 'C') []) []),Left "TRIES!"]),Stmt END]

This input stream is properly parsed to the grammar defined in the BNF.

Bibliography

About BNF notation Available at: http://cuiwww.unige.ch/dbresearch/Enseignement/analyseinfo/AboutBNF.htm

Adit.io (2013) 17 April. Available at: http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

ALGOL 60 (2016) in *Wikipedia*. Available at: https://en.wikipedia.org/wiki/ALGOL_60

BNF notation for syntax (no date) Available at: https://www.w3.org/Notation.html

Compiler design - lexical analysis (2016) Available at: https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm

Context-free grammar (2016) in *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Context-free_grammar

Context-free Grammars (no date) Available at: https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html

Diehl, S. (2015) *Write you a Haskell building a modern functional compiler from first principles*. Available at: http://dev.stephendiehl.com/fun/WYAH.pdf

Formal language (2016) in *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Formal_language

Functors, Applicative Functors and Monoids - learn you a Haskell for great good! Available at: http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Haskell/Applicative functors - Wikibooks, open books for an open world Available at: https://en.wikibooks.org/wiki/Haskell/Applicative_functors

Hutton, G. and Meijer, E. *Under consideration for publication in J. Functional programming Monadic parsing in Haskell*. Available at: http://www.cs.nott.ac.uk/~pszgmh/pearl.pdf

Monad - HaskellWiki Available at: https://wiki.haskell.org/Monad

Pythux (2016) *What's in a parser combinator?* Available at: http://remusao.github.io/whats-in-a-parser-combinator.htm

Swierstra, D.S. (2009) 'Combinator parsing: A short Tutorial', in *Language Engineering and Rigorous Software Development*. Springer Nature, pp. 252–300.

Available at: https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html

Apendix

Source code of Haskell program.

```
{-# LANGUAGE StandaloneDeriving #-}

parseFile :: FilePath -> IO ()

parseFile filePath = do

  file <- readFile filePath

  putStrLn (show(runParser prog file))

runParser :: Parser a -> String -> a

runParser m s =

  case parse m s of

    Just [(res, [])] -> res

    Just [(_, rs)]   -> error rs

    _            -> error "Parser error."

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

produce :: a -> Parser a

produce x = Parser (\ts -> Just [(x, ts)])

parse :: Parser a -> (String -> Maybe [(a, String)])

parse (Parser p) = p

failure :: Parser a

failure = Parser (\ts -> Nothing)

item :: Parser Char

item = Parser (\ts -> case ts of

  [] -> Nothing

  (x:xs) -> Just [(x,xs)])

instance Functor Parser where

  fmap f p = Parser (\ts -> case parse p ts
of

    Nothing -> Nothing

    Just [(x, ts')] -> Just [((f x), ts')])

instance Applicative Parser where

  pure a = Parser (\ts -> Just [(a, ts)])

  p <*> q = Parser (\ts -> do

          [(f,ts')] <- parse p ts

          [(x,ts'')] <- parse q ts'

          Just [(f x, ts'')])

instance Monad Parser where

  p >>= f = Parser (\ts -> case parse p ts
of

        Nothing -> Nothing

        Just [(x,ts')] -> parse (f x) ts')

class Applicative f => Choice f where

  empty :: f a

  (<|>) :: f a -> f a -> f a

instance Choice Parser where

  empty = failure

  px <|> py = Parser (\ts -> case parse px
ts of

          Nothing -> parse py ts

          xs -> xs)

many :: Parser a -> Parser [a]

many x = some x <|> produce []

some :: Parser a -> Parser [a]

some x = (:) <$> x <*> many x

satisfy :: (Char -> Bool) -> Parser Char

satisfy p = item >>= \x ->

        if p x then produce x else failure

oneOf :: [Char] -> Parser Char

oneOf cs = satisfy (\c -> elem c cs)

noneOf :: [Char] -> Parser Char

noneOf cs = satisfy (\c -> not(elem c cs))

sepBy1 :: Parser a -> Parser s -> Parser [a]

sepBy1 p x = (:) <$> p <*> many (x *> p)

char :: Char -> Parser Char

char x = satisfy (x ==)

string :: String -> Parser String

string "" = produce ""

string(x:xs) = char x >>= \x' ->

               string xs >>= \xs' ->

               produce (x:xs)

whitespace :: Parser ()

whitespace = many (oneOf " \t") *> produce
()

tok :: String -> Parser String

tok t = string t <* whitespace
```

```
data Prog  =  Prog [Line]

data Line  =  Cmnd Cmnd | Stmt Stmt | Line Int Stmt

data Cmnd  =  CLEAR | LIST | RUN

data Stmt  =  PRINT Args
           |  IF Expr Rel Expr Stmt
           |  GOTO Expr
           |  INPUT [Ident]
           |  LET Ident Expr
           |  GOSUB Expr
           |  RETURN
           |  END

type Args    =  [Either String Expr]

data Parity  =  POS | NEG

data Expr    =  Expr Parity Term [Exprs]

data Exprs   =  (:+:) Term | (:-:) Term

data Term    =  Term Fact [Terms]

data Terms   =  (:*:) Fact | (:/:) Fact

data Fact    =  Var Ident | Number Int | Parens Expr

data Rel     =  (:<:) | (:<=:) | (:<>:) | (:=:) | (:>:) | (:>=:)

type Ident   =  Char

achar :: Parser Char

achar = noneOf("\"\n\r.")

str :: Parser String

str = tok "\"" *> many achar <* tok "\""

digit :: Parser Char

digit = oneOf ['0' .. '9'] <* whitespace

number :: Parser Int

number = ((some digit) >>= return . read) <* whitespace

rel :: Parser Rel

rel = ((:<>:)  <$ tok "<>")
  <|> ((:<>:)  <$ tok "><")
  <|> ((:=:)   <$ tok "=")
  <|> ((:<=:)  <$ tok "<=")
  <|> ((:<:)   <$ tok "<")
  <|> ((:>=:)  <$ tok ">=")
  <|> ((:>:)   <$ tok ">")

var :: Parser Ident

var = oneOf ['A' .. 'Z'] <* whitespace

fact :: Parser Fact

fact = (Var <$> var)
  <|> (Number <$> number)
  <|> (Parens <$ tok "(" <*> expr <* tok ")")

expr :: Parser Expr

expr = Expr <$> ((POS <$ tok "+") <|> (NEG <$ tok "-") <|> pure POS)
  <*> term <*> many exprs

exprs :: Parser Exprs

exprs = ((:+:) <$ tok "+" <*> term)
    <|> ((:-:) <$ tok "-" <*> term)

term :: Parser Term

term = Term <$> fact <*> many terms

terms :: Parser Terms

terms = ((:*:) <$ tok "*" <*> fact)
  <|> ((:/:) <$ tok "/" <*> fact)

vars :: Parser [Ident]

vars = sepBy1 var (tok ",")

args :: Parser Args

args = sepBy1 ((Left <$> str) <|> (Right <$> expr)) (tok ",")

stmt :: Parser Stmt

stmt = (PRINT   <$ tok "PRINT" <*> args)
   <|> (IF      <$ tok "IF" <*> expr <*> rel <*> expr <* tok "THEN" <*> stmt)
   <|> (GOTO    <$ tok "GOTO" <*> expr)
   <|> (INPUT   <$ tok "INPUT" <*> vars)
   <|> (LET     <$ tok "LET" <*> var <* tok "=" <*> expr)
   <|> (GOSUB   <$ tok "GOSUB" <*> expr)
   <|> (RETURN  <$ tok "RETURN")
   <|> (END     <$ tok "END")

cr :: Parser [Char]

cr = many (oneOf "\r\n")

cmnd :: Parser Cmnd
```

```
cmnd = (CLEAR  <$ tok "CLEAR") <|> (LIST <$
tok "LIST") <|> (RUN <$ tok "RUN")

line :: Parser Line

line = (Cmnd  <$> cmnd <* cr) <|> (Stmt  <$>
stmt <* cr) <|> (Line  <$> number <*> stmt
<* cr)

prog :: Parser Prog

prog = Prog <$> many line

deriving instance Show Prog

deriving instance Show Line
```

```
deriving instance Show Cmnd

deriving instance Show Stmt

deriving instance Show Expr

deriving instance Show Term

deriving instance Show Terms

deriving instance Show Fact

deriving instance Show Parity

deriving instance Show Exprs

deriving instance Show Rel
```