Lec 21: Monadic Parser

CS220: Programming Principles

Sang Kil Cha

Interpreter

Metalinguistic Abstraction

A form of language for describing another language. This allows us to better understand a computation problem by using a new language.

> Can we leverage F# (or any other language) to describe and evaluate another language?

Writing an Interpreter

An evaluator (or interpreter) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression[1].

An interpreter is just another program.

---

[1] Excerpt from the Wizard book Ch. 4.

Why?

We have been simply ***using*** programming languages to develop a program. However, most powerful programmers should be able to design their own languages for their needs: we should become a ***designer*** of programming languages, but not merely a user.

Interpretation

Interpreter.

```
module Interpreter =
  let grammar = // embedded grammar
  let context = // initial context
  let interpret (source: string) =
    parse grammar source // Grammar -> string -> AST
    |> evaluate context // AST -> Context -> Context
```

Interpretation runs in two major steps: `parse` and `evaluate`. We will discuss the details one by one.

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.

Monadic Parser

Parsing

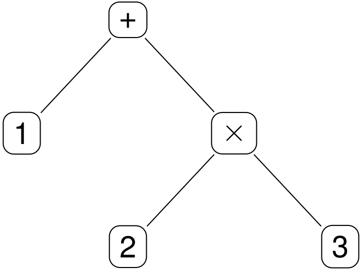
Parsing is the process of transforming a string into an Abstract Syntax Tree (AST) based on a grammar.

```
val parse:  Grammar -> string -> AST
```

AST

AST is a tree representing the syntactic structure of a program (or written words in a language).

An AST for $1 + 2 \times 3$:



AST is Not Ambiguous

A language usually has some ambiguity, but an AST does not. For example, $1 + 2 \times 3$ may mean either $(1 + 2) \times 3$ or $1 + (2 \times 3)$ depending on the precedence of the $+$ and $\times$ operator. However, the AST in the previous page is not ambiguous.

AST is Abstract

Both “1+2” and “1   + 2” correspond to the same AST, although the two strings are different if we look at their ***concrete*** syntax. Therefore, they are ***abstract***.

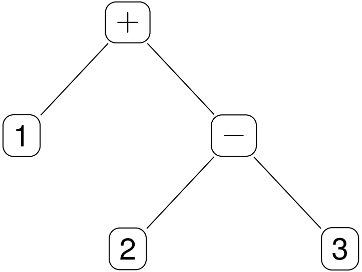
AST is a Tree

AST for a “language of integer addition and subtraction” (***AddSubLang***).

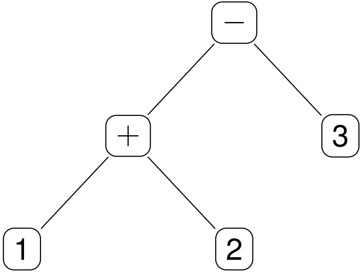
```
type Expr =
  | Number of int
  | Add of Expr * Expr // This line effectively creates a tree
  | Sub of Expr * Expr // This line effectively creates a tree
```

AST Example (1 + 2 - 3)

`Add (Number 1, Sub (Number 2, Number 3))`



`Sub (Add (Number 1, Number 2), Number 3)`



How Do We Describe a Grammar?

Backus-Naur Form (BNF) ***recursively*** describes a grammar[2] of a language.

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

1. `A ::= B` means `A` is defined as `B`.
2. Symbols without angle brackets mean a ***terminal***, which is an elementary symbol that cannot be replaced with another.
3. Symbols with angle brackets mean a ***nonterminal***, which can be replaced by terminals.
4. `A | B` means “A or B”.

---

[2] Context-Free Grammar

Our AddSubLang Grammar

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> ::= <digit> | <number> <digit>
<expr> ::= <number> + <expr>
         | <number> - <expr>
         | <number>
```

Our goal: writing a parser that "looks like" this grammar!

Parser Computation Expression

1. Generic type constructor: `Parser<'a>`.
2. Bind operator: given a `Parser<'a>` and a function (`'a -> Parser<'b>`), return a new parser `Parser<'b>`.
3. Return operator: given a value ('a), return a new parser (`Parser<'a>`).
4. And many more operations possible!

Example Scenarios

- Consider a parser P that can parse a character ‘`1`’. We can “bind” the parsing result of P to a parameter of a function (i.e., continuation) in order to create a new parser that makes use of the result.
- Imagine two parsers P_1 and P_2 that can parse a string `"abc"` and `"def"`, respectively. We can “combine” the two parsers to create a new parser that can parse `"abcdef"`.

> We call this type of computation expression as a “monadic parser”, or a “parser combinator”.

Parser Type

```
type Parser<'a> = {
  Parse: string -> Result<'a * string, string>
}
```

Given a string, the `Parse` function returns a `Result` type. The success case of the `Result` returns a parsed value of type `'a` and the next string to parse, and the failure case returns an error message of type `string`.

Simple Character Parser

```
/// Helper function to run Parser computation expression
let runOnInput parser str =
  parser.Parse str

module Parser =
  let char =
    { Parse = fun s ->
        if System.String.IsNullOrEmpty (s) then
          Error "No more input."
        else
          Ok (s[0], s[1..]) }
```

Make It a Monad

```
type ParserBuilder () =
  member __.Bind (p, f) =
    { Parse = (fun s ->
        match runOnInput p s with
        | Ok (v, rest) -> runOnInput (f v) rest
        | Error e -> Error e) }

  member __.Return (v) =
    { Parse = (fun s -> Ok (v, s)) }

let parser = ParserBuilder ()
```

Combining Parsers

Parser for two consecutive (any) characters.

```
let twoChars =
  parser {
    let! a = char
    let! b = char
    return (a, b)
  } // what is the type of twoChars?
```

This way of combining two parsers is so common, so we even create a function (and an operator) that does this: `andThen` function.

`andThen` Function

`andThen` takes in two parsers and return a new parser.

```
let andThen p1 p2 =
  parser {
    let! a = p1
    let! b = p2
    return (a, b)
  }

let (.>>.) = andThen // infix operator for andThen
let twoChars = char .>>. char // much concise now!
```

Parsing a Specific Character

```
let char ch =
  { Parse = fun s ->
      if System.String.IsNullOrEmpty (s) then
        Error "No more input."
      else
        if s[0] = ch then Ok (s[0], s[1..])
        else Error "Invalid character." }
```

Parsing a Specific String

```
let strABC1 = char 'A' .>>. char 'B' .>>. char 'C'
let rec sequence parsers =
  match parsers with
  | [] -> parser { return [] }
  | hd :: tl ->
    parser {
      let! h = hd
      let! t = sequence tl
      return (h :: t)
    }
let strABC2 =
  [ char 'A'; char 'B'; char 'C' ] |> sequence
```

Map from a Parser to Another

```
let map f parser =
  { Parse = fun s ->
      match runOnInput parser s with
      | Ok (v, rest) -> Ok (f v, rest)
      | Error e -> Error e }

let (|>>) p f = map f p

let strABC =
  sequence [ char 'A'; char 'B'; char 'C' ]
  |>> (List.toArray >> System.String)
```

Parse A or B

Given two parsers P_A and P_B, create a parser that runs either one of them.

```
let orElse p1 p2 =
  { Parse = fun s ->
      match runOnInput p1 s with
      | Ok (v, rest) -> Ok (v, rest)
      | Error _ -> runOnInput p2 s }

let (<|>) = orElse
```

Parser for Numbers

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

```
let digit = char '0' <|> char '1' <|> ... <|> char '9'
```

<number> ::= <digit> | <number> <digit>

```
let rec number =
  (number .>>. digit) <|> digit // type mismatch
```

Recursive Definition of a Number

```
let number =
  let rec num () =
    parser {
      let! d = digit
      let! n = num ()
      return d :: n
    } <|> parser { return [] }
  num ()
  |>> (List.toArray >> System.String >> int >> Number)
```

But this is dirty!

Extended BNF (Use Repetition)

We can use curly braces to represent zero or more occurrences of a term.

```
<number> ::= digit { <digit> }   // repeatition
```

```
let rec zeroOrMore p s =
  match runOnInput p s with
  | Error _ -> ([], s)
  | Ok (v, s) ->
    let v', s' = zeroOrMore p s
    v :: v', s'

let many p = { Parse = fun s -> Ok (zeroOrMore p s) }
```

Redefining Number with `many`

```
<number> ::= digit { <digit> }
```

```
let number =
  parser {
    let! d = digit
    let! ds = many digit // zero or more
    return List.toArray (d :: ds) |> System.String |> int |>
        Number
  }
```

Parsing Expressions

```
<expr> ::= <number> + <expr>
         | <number> - <expr>
         | <number>
```

```
let rec expr =
  parser {
    let! n = number
    let! _ = char '+'
    let! e = expr
    return Add (n, e)
  } <|> number // Subtraction case omitted
```

Warning

```
warning FS0040:  This and other recursive references to the object(s)
being defined will be checked for initialization-soundness at runtime
through the use of a delayed reference.  This is because you are
defining one or more recursive objects, rather than recursive
functions.
```

Can we avoid this warning?

Tying the Knot

Resolve circular dependencies by making the expression parser (`expr`) reference a mutable parser internally.

```
let mutable exprRef = { Parse = fun _ -> failwith "XXX" }
let expr = { Parse = fun s -> runOnInput exprRef s }

exprRef <-
  parser {
    let! n = number
    let! _ = char '+'
    let! e = expr
    return Add (n, e)
  } <|> number // Subtraction case omitted
```

In-Class Activity #21

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.

```
- git clone https://github.com/KAIST-CS220/CS220-Main.git
```

2. Move in to the directory `CS220-Main/Activities`

```
- cd CS220-Main
- cd Activities
```

Problem: Finish the Parser Implementation

Finalize the parser rules so that we can parse AddSubLang expressions with space chars. For example, the parser should be able to parse the following string: `1 + 2 - 3`.

Conclusion

- Monads are a powerful abstraction mechanism.
- We have observed its power by building our own parser combinator.
- Writing and evaluating your own language is essential to be competent in programming.

Further Readings

- `https://fsharpforfunandprofit.com/posts/understanding-parser-combinators/`.

Question?