Lec 22: Interpreter

CS220: Programming Principles

Sang Kil Cha

Parser (2)

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.

Recap: Parser Combinator

- A parser combinator is a higher-order function that accepts several parsers as input and returns a new parser as its output.
- We can use computation expressions to build a parser combinator.

Extending Our Parser

We have built a parser for AddSubLang. What if we want to extend the language to include multiplication and division?

1. Extend the AST type.
2. Extend the parsing rules.

Extending the AST

```
type Expr =
   | Number of int
   | Add of Expr * Expr
   | Sub of Expr * Expr
   | Mul of Expr * Expr
   | Div of Expr * Expr
```

Extending the Parsing Rules

```
let arith op constructor =
  parser {
    let! n = number
    let! _ = many (char ' ')
    let! _ = char op
    let! _ = many (char ' ')
    let! e = expr
    return constructor (n, e)
  }

let add = arith '+' Add
let sub = arith '-' Sub
let mul = arith '*' Mul
let div = arith '/' Div

exprRef <- add <|> sub <|> mul <|> div <|> number
```

Check the Parsed AST

Let's try to parse "$1 + 2 \times 3 - 4$" with the extended parser. Then we get:

```
Ok (Add (Number 1, Mul (Number 2, Sub (Number 3, Number 4))), "")
```

The parsed AST is not what we expected.

Precedence and Associativity

- We need to consider the precedence and associativity of operators.
- For example, "$1 + 2 \times 3 - 4$" should be parsed as "$((1 + (2 \times 3)) - 4)$".

Fixing the Parsing Rules (1st Attempt)

```
// give higher precedence to multiplication and
   division.
exprRef <- mul <|> div <|> add <|> sub <|> number

Parser.runOnInput expr "1 + 2 * 3 - 4"
|> printfn "%A"
```

We get:

`Ok (Add (Number 1, Mul (Number 2, Sub (Number 3, Number 4))), "")`

Need Left-Associativity

- The parsing rules above give higher precedence to multiplication and division.
- However, the parsed AST is still not what we expected because our parsing rules follow right-associativity.

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

Fixing the Parsing Rules (2nd Attempt)

```
let arith op lhs rhs constructor =
  parser {
    let! n = lhs
    let! _ = many (char ' ')
    let! _ = char op
    let! _ = many (char ' ')
    let! e = rhs
    return constructor (n, e)
  }

let add = arith '+' expr number Add
let sub = arith '-' expr number Sub
let mul = arith '*' expr number Mul
let div = arith '/' expr number Div
exprRef <- mul <|> div <|> add <|> sub <|> number
```

Left-Recursion Problem

The parsing rules above are left-recursive, causing an infinite recursion when parsing.

Stack overflow. Repeat X times: ☹

Fixing the Left-Recursion Problem

Introduce a new ‘term’ using repeatition to wrap multiplication and division.

```
<term> ::= <number> { ( * | / ) <number> }
<expr> ::= <term> + <expr>
         | <term> - <expr>
         | <term>
```

★ There are many other ways to refactor the parsing rules to avoid left-recursion.

Fixing the Grammar

```
let muldiv = parser {
  // ... omitted
}

let term = parser {
  let! n = number
  let! _ = many (char ' ')
  let! r = many muldiv
  return r
}

let add = arith '+' term expr Add
let sub = arith '-' term expr Sub
exprRef <- add <|> sub <|> term
```

In-Class Activity #22

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

Fix the `muldiv` parser and the `term` parser to finish the implementation of the parser.

Interpreter

Evaluation Step

Interpreter "evaluates" a list of ASTs to derive a value and update the context.

From an AST to a Value

```
Parse:   Grammar -> string[1]-> AST

Evaluate:   AST -> Context -> Context * Value
```

★ Recall our AddSubLang can only return an int as a value.

[1] Grammar was built-in to our parser.

Context in AddSubLang?

Is there a certain context to consider? In other words, can an expression in AddSubLang be interpreted in a different manner depending on a context?

Implementing an Interpreter

F# is perfect for implementing interpreters (and compilers). Why?

1. Representing an AST (a tree) is natural and easy.
2. Pattern matching on an AST can make interpreter concise.
3. Descendant of ML (Meta Language) ☺.

Interpreter for AddSubLang

```
let evaluate = function
  | Number n -> n // This is just a value
  | Add (a, b) -> // ?
  | Sub (a, b) -> // ?
```

How can we handle nested expressions?

Recursion!

```
let rec evaluate = function
  | Number n -> n
  | Add (a, b) ->
    let a' = evaluate a
    let b' = evaluate b
    a' + b'
  | Sub (a, b) ->
    let a' = evaluate a
    let b' = evaluate b
    a' - b'
```

Interpreting a Sequence of Statements

We need to maintain a context to correctly handle multiple statements. For example, the following code has three statements to evaluate.

```
let x = 1
let y = 2
x + y
```

After evaluating the first two statements, we need to remember that the symbol `x` and `y` corresponds to a value 1 and 2, respectively.

Implementation Sketch

```
let rec evalExpr ctxt = function
  | Number n -> n
  | ...
and rec evalStmt ctxt = function // mutually recursive
  | Let (v, e) ->
    let e' = evalExpr ctxt e
    ctxt' // update the ctxt to have a mapping v to e'
  | Expression e ->
    evalExpr ctxt e

let run program =
  let initialContext = // empty map (symbols to values).
  program
  |> List.fold evaluate initialContext
```

In-Class Activity (cont'd)

Implement your own interpreter for our language (with multiplication and division) in addition to the previous parser implementation. The end result should be a calculator that can handle simple arithmetic expressions.

Conclusion

Further Readings

- The Wizard Book Ch. 4.1 (and the rest of Ch. 4).

Question?