MoonBit 月兔编程

现代编程思想

案例：语法解析器

Hongbo Zhang

语法解析器

MoonBit 月兔编程

- 案例目标
  - 解析基于自然数的数学表达式：`"(1+ 5) * 7 / 2"`
  - 转化为单词列表

    ```
    LParen Value(1) Plus Value(5) Multiply Value(7) Divide Value(2)
    ```

  - 转化为抽象语法树

    ```
    Division(Multiply(Add(Value(1), Value(5)), Value(7)), Value(2))
    ```

  - 计算最终结果：21
- 语法分析
  - 对输入文本进行分析并确定其语法结构
  - 通常包含**词法分析**和**语法分析**
  - 本节课均利用**语法解析器组合子**（parser combinator）为例

词法分析

MoonBit 月兔编程

- 将输入分割为单词
  - 输入：字符串/字节块
  - 输出：单词流
  - 例如：`"12 +678"` -> `[ Value(12), Plus, Value(678) ]`
- 通常可以通过有限状态自动机完成
  - 一般用领域特定语言定义后，由软件自动生成程序
- 算术表达式的词法定义

```
1. Number     = %x30 / (%x31-39) *(%x30-39)
2. LParen     = "("
3. RParen     = ")"
4. Plus       = "+"
5. Minus      = "-"
6. Multiply   = "*"
7. Divide     = "/"
8. Whitespace = " "
```

词法分析

- 算术表达式的词法定义

```
1. Number = %x30 / (%x31-39) *(%x30-39)
2. Plus   = "+"
```

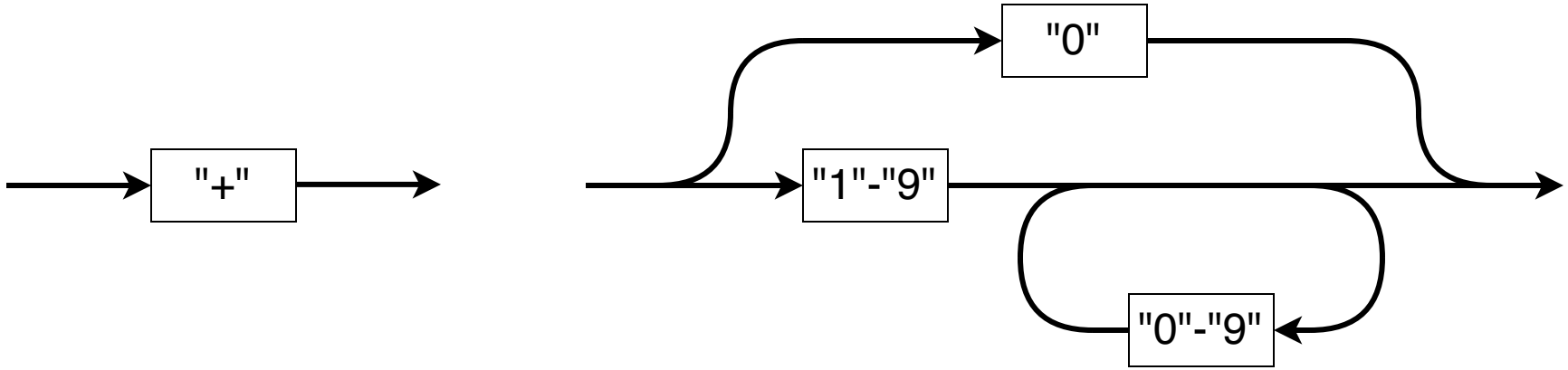
- 每一行对应一个匹配规则
  - `"xxx"`：匹配内容为xxx的字符串
  - `a b`：匹配规则a，成功后匹配规则b
  - `a / b`：匹配规则a，匹配失败则匹配规则b
  - `*a`：重复匹配规则a，可匹配0或多次
  - `%x30`：UTF编码十六进制表示为30的字符（`"0"`）

词法分析

MoonBit 月兔编程

- 算术表达式的词法定义

```
1. Number = %x30 / (%x31-39) *(%x30-39)
2. Plus   = "+"
```



- 单词定义

```
1. enum Token {
2.   Value(Int); LParen; RParen; Plus; Minus; Multiply; Divide
3. } derive(Show)
```

MoonBit 月兔编程

解析器组合子

- 构造可组合的解析器

```
// V 代表解析成功后获得的值
// @string.View 代表 String 的一部份
type Lexer[V] (@string.View) -> (V, @string.View)?

fn Lexer::parse[V](self : Lexer[V], str : @string.View) -> (V, @string.View)? {
  self.inner()(str)
}
```

  - 我们简化处理报错信息以及错误位置（可以使用 `Result[A, B]` ）

解析器组合子

MoonBit 月兔编程

- 最简单的解析子：判断下一个待读取的字符是否符合条件，符合则读取并前进

```
fn pchar(predicate : (Char) -> Bool) -> Lexer[Char] {
  Lexer(input => match input {
    [ch, .. rest] if predicate(ch) => Some((ch, rest))
    _ => None
  })
}
```

- 例如

```
test "pchar" {
  inspect(pchar(ch => ch is 'a').parse("asdf"),
    content=(
      #|Some(('a', "sdf"))
  ))
  inspect(pchar(ch => ch is 'a').parse("sdf"), content="None")
}
```

词法分析

- 单词定义：数字或左右括号或加减乘除

```
enum Token {
  Value(Int)
  LParen; RParen; Plus; Minus; Multiply; Divide
} derive(Show)
```

- 分析运算符、括号、空白字符等

```
let symbol: Lexer[Char] =  pchar(ch => ch
  is ('+' | '-' | '*' | '/' | '(' | ')'))
let whitespace : Lexer[Char] = pchar(ch => ch is ' ')
```

解析器组合子

- 如果解析成功，对解析结果进行转化

```
fn[I, O] Lexer::map(self : Lexer[I], f : (I) -> O) -> Lexer[O] {
  Lexer(fn(input) { self.parse(input).map(pair => (f(pair.0), pair.1)) })
}
```

- 分析运算符、括号并映射为对应的枚举值

```
let symbol : Lexer[Token] = pchar(ch => ch
  is ('+' | '-' | '*' | '/' | '(' | ')')).map(token => match token {
  '+' => Token::Plus; '-' => Token::Minus
  '*' => Token::Multiply; '/' => Token::Divide
  '(' => Token::LParen; ')' => Token::RParen
  _ => panic()
})
```

解析器组合子

MoonBit 月兔编程

- 解析 `a`，如果成功再解析 `b`，并返回 `(a, b)`

```
fn[V1, V2] Lexer::then(self : Lexer[V1], parser2 : Lexer[V2]) -> Lexer[(V1, V2)] {
  Lexer(fn(input) {
    guard self.parse(input) is Some((value, rest)) else { return None }
    guard parser2.parse(rest) is Some((value2, rest2)) else { return None }
    Some(((value, value2), rest2))
}) }
```

- 解析 `a`，如果失败则解析 `b`

```
fn[Value] Lexer::or(self : Lexer[Value], parser2 : Lexer[Value]) -> Lexer[Value] {
  Lexer(fn(input) {
    match self.parse(input) {
      None => parser2.parse(input)
      Some(_) as result => result
    }
}) }
```

MoonBit 月兔编程

解析器组合子

- 重复解析 `a`，零或多次，直到失败为止

```
fn[Value] Lexer::many(self : Lexer[Value]) -> Lexer[@list.T[Value]] {
  Lexer(fn(input) {
    loop (input, @list.empty()) {
      (rest, cumul) =>
        match self.parse(rest) {
          None => Some((cumul.rev(), rest)) // List 是栈，需要反转
          Some((value, rest)) => continue (rest, @list.construct(value, cumul))
        }
    }
  })
}
```

词法分析

- 整数分析

```
// 通过字符编码将字符转化为数字
let zero : Lexer[Int] = pchar(ch => ch is '0').map(_ => 0)

let one_to_nine : Lexer[Int] = pchar(ch => ch is ('1'..='9')).map(ch => ch.to_int() - '0'.to_int())

let zero_to_nine : Lexer[Int] = pchar(ch => ch is ('0'..='9')).map(ch => ch.to_int() - '0'.to_int())

// number = %x30 / (%x31-39) *(%x30-39)
let value : Lexer[Token] = zero
  .or(
    one_to_nine
    .then(zero_to_nine.many())
    .map(pair => {
      let (i, ls) = pair
      ls.fold((i, j) => i * 10 + j, init=i)
    }),
  )
  .map(Token::Value(_))
```

词法分析

MoonBit 月兔编程

- 对输入流进行分析
  - 在单词之间可能存在空格

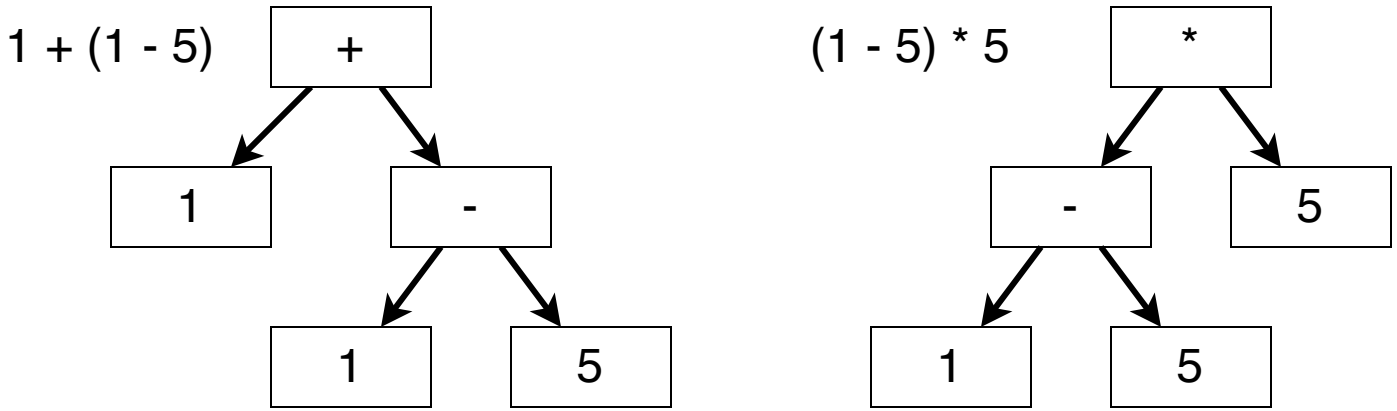
```
let tokens : Lexer[@list.T[Token]] = value
  .or(symbol)
  .then(whitespace.many())
  .map(p => p.0)
  .many()
test {
  inspect(
  tokens.parse("-10123+-+523 103    ( 5) )  ").unwrap(),
  content=(
    #|(@list.of([Minus, Value(10123), Plus, Minus, Plus, Value(523), Value(103), LParen, Value(5), RParen, RParen]), "")
  ),
)
}
```

- 我们成功地分割了字符串
  - `-` `10123` `-` `+` `-` `523` `103` `(` `5` `)` `)`
  - 但这不符合数学表达式的语法

语法分析

- 对单词流进行分析，判断是否符合语法
  - 输入：单词流
  - 输出：抽象语法树

```
1. expression = Value / "(" expression ")"
2. expression =/ expression "+" expression / expression "-" expression
3. expression =/ expression "*" expression / expression "/" expression
```



语法分析

- 语法定义

```
1. expression = Value / "(" expression ")"
2. expression =/ expression "+" expression / expression "-" expression
3. expression =/ expression "*" expression / expression "/" expression
```

- 问题：运算符的优先级、结合性
  - 优先级：$\mathtt{a} + \mathtt{b} \times \mathtt{c} \rightarrow \mathtt{a} + (\mathtt{b} \times \mathtt{c})$
  - 结合性：$\mathtt{a} + \mathtt{b} + \mathtt{c} \rightarrow (\mathtt{a} + \mathtt{b}) + \mathtt{c}$
  - 当前语法具有二义性

语法分析

- 修改后的语法定义

```
1. atomic     = Value / "(" expression ")"
2. combine    = atomic  /    combine "*" atomic  /    combine "/" atomic
3. expression = combine / expression "+" combine / expression "-" combine
```

- 注意到除了简单的组合以外，出现了左递归
  - 左递归会导致我们的解析器进入循环
  - 解析器将尝试匹配运算符左侧的规则而不前进
  - 拓展：自底向上解析器可以处理左递归

语法分析

MoonBit 月兔编程

- 修改后的语法定义

```
atomic     = Value / "(" expression ")"
combine    = atomic  *( ("*" / "/") atomic)
expression = combine *( ("+" / "-") combine)
```

- 数据结构

```
enum Expression {
  Number(Int)
  Plus(Expression, Expression)
  Minus(Expression, Expression)
  Multiply(Expression, Expression)
  Divide(Expression, Expression)
}
```

MoonBit 月兔编程

语法解析

- 定义语法解析组合子

```
type Parser[V] (@list.T[Token]) -> (V, @list.T[Token])?

fn[V] Parser::parse(self : Parser[V], tokens : @list.T[Token]) -> (V, @list.T[Token])? { self.inner()(tokens) }
```

- 大部分组合子与 `Lexer[V]` 类似
- 递归组合：`atomic = Value / "(" expression ")"`
  - 延迟定义
  - 递归函数

MoonBit 月兔编程

递归定义

- 延迟定义
  - 利用引用定义 `Ref[Parser[V]]`： `struct Ref[V] { mut val : V }`
  - 在定义其他解析器后更新引用中内容

```
1. fn[Value] Parser::from_ref(ref_ : Ref[Parser[Value]]) -> Parser[Value] {
2.   Parser(fn(input) { ref_.val.parse(input) })
3. }
```

  - `ref.val` 将在使用时获取，此时已更新完毕

递归定义

MoonBit 月兔编程

- 延迟定义

```
fn parser() -> Parser[Expression] {
  // 首先定义空引用
  let expression_ref : Ref[Parser[Expression]] = { val : Parser(_ => None) }

  // atomic = Value / "(" expression ")"
  let atom =  // 利用引用定义
    (lparen.then(Parser::from_ref(expression_ref)).then(rparen).map(expr => expr.0.1).or(number)

  // combine = atomic *( ("*" / "/") atomic)
  let combine = atom.then(multiply.or(divide).then(atom).many()).map(pair => {
    guard pair is (expr, list)
    list.fold(init=expr, (expr, op_expr) => match op_expr { ... })
  })

  // expression = combine *( ("+" / "-") combine)
  expression_ref.val = combine.then(plus.or(minus).then(combine).many()).map(pair => {
    guard pair is (expr, list)
    list.fold(init=expr, (expr, op_expr) => match op_expr { ... })
  })

  expression_ref.val
}
```

递归定义

- 递归函数
  - 解析器本质上是一个函数
  - 定义互递归函数后，将函数装进结构体

```
fn recursive_parser() -> Parser[Expression] {
  // 定义互递归函数
  // atomic = Value / "(" expression ")"
  letrec atom = fn(tokens: @list.T[Token]) -> (Expression, @list.T[Token])? {
    lparen.then( Parser(expression) ).and(rparen).map(expr => expr.0.1)
      .or(number).parse(tokens)
  }
  and combine = fn(tokens: @list.T[Token]) -> (Expression, @list.T[Token])? { ... }
  and expression = fn (tokens: @list.T[Token]) -> (Expression, @list.T[Token])? { ... }

  // 返回函数代表的解析器
  Parser(expression)
}
```

语法树之外：Tagless Final

- 计算表达式，除了生成为抽象语法树再解析，我们还可以有其他的选择
- 我们通过“行为”来进行抽象

```
1. trait Expr : Add + Sub + Mul + Div {
2.   number(Int) -> Self
3. }
```

  - 接口的不同实现即是对行为的不同语义

语法树之外：Tagless Final

MoonBit 月兔编程

- 我们利用行为的抽象定义解析器

```
fn[E : Expr] recursive_parser() -> Parser[E] {
  let number : Parser[E] = ptoken(token => token is Value(_)).map(ptoken => {
    guard ptoken is Value(i)
    E::number(i)  // 利用抽象的行为
  })

  letrec atomic = fn(tokens: @list.T[Token]) -> (E, @list.T[Token])? { ... }
  // 转化为 a * b * c * ... 和 a / b / c / ...
  and combine = fn(tokens: @list.T[Token]) -> (E, @list.T[Token])? { ... }
  // 转化为 a + b + c + ... 和 a - b - c - ...
  and expression=fn(tokens: @list.T[Token]) -> (E, @list.T[Token])? { ... }

  Parser(expression)
}

// 结合在一起
fn[E : Expr] parse_string(
str : @string.View,
) -> (E, @string.View, @list.T[Token])? {
  guard tokens.parse(str) is Some((token_list, rest_string)) else { return None }
  guard recursive_parser().parse(token_list) is Some((expr, rest_token)) else { return None }
  Some((expr, rest_string, rest_token))
}
```

MoonBit 月兔编程

语法树之外：Tagless Final

- 我们可以提供不同的实现，获得不同的诠释

```
enum Expression { ... } derive(Show) // 语法树实现
type BoxedInt Int derive(Show) // 整数实现
// 实现接口（此处省略其他方法）
fn BoxedInt::number(i: Int) -> BoxedInt { BoxedInt(i) }
fn Expression::number(i: Int) -> Expression { Number(i) }
test {
  // 获得语法树
  inspect(
    (
      parse_string("1 + 1 * (307 + 7) + 5 - 3 - 2") :
      (Expression, @string.View, @list.T[Token])?).unwrap(),
    content=(
      #|(Minus(Minus(Plus(Plus(Number(1), Multiply(Number(1), Plus(Number(307), Number(7)))), Number(5)), Number(3)), Number(2)), "", @list.of([]))
    ),
  )
  // 获得计算结果
  inspect(
    (
      parse_string("1 + 1 * (307 + 7) + 5 - 3 - 2") :
      (BoxedInt, @string.View, @list.T[Token])?).unwrap(),
    content=(
      #|(BoxedInt(315), "", @list.of([]))
    ),
  )
}
```

总结

MoonBit 月兔编程

- 本节课展示了一个语法解析器
  - 介绍了词法解析的概念
  - 介绍了语法解析的概念
  - 展示了语法解析组合子的定义与实现
  - Tagless Final的概念与实现
- 拓展阅读
  - 调度场算法
  - 斯坦福CS143 第1-8课 或
  - 《编译原理》前五章 或
  - 《现代编译原理》前三章
- 拓展练习
  - 实现兼容各类“流”的语法解析组合子