

现代编程思想

案例：语法解析器

Hongbo Zhang

语法解析器

- 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

- 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   = "+"
```

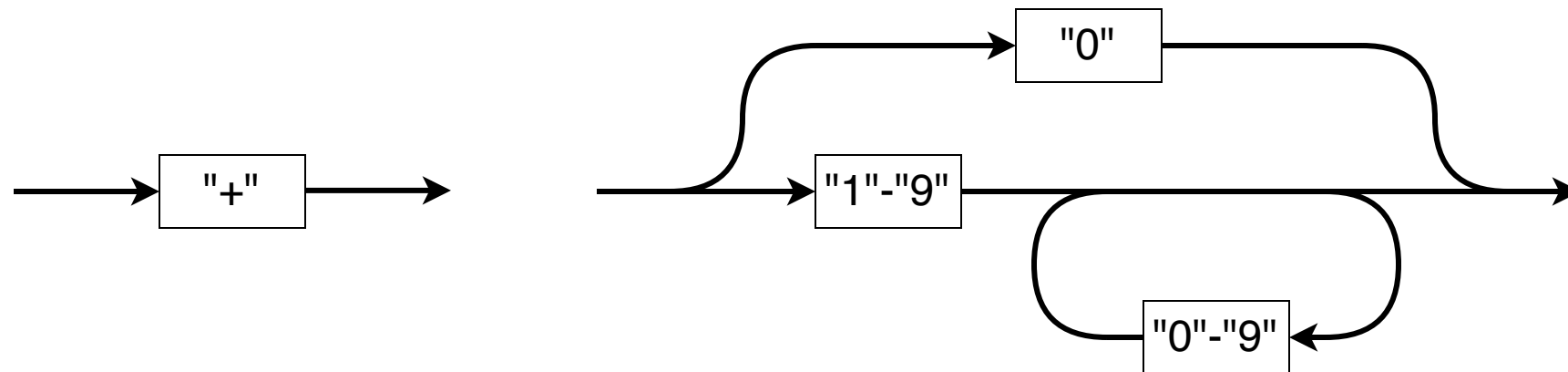
- 每一行对应一个匹配规则

- "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

- 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(Debug)
```

解析器组合子

- 构造可组合的解析器

```
1. // V 代表解析成功后获得的值
2. // Lexer[V] == (String) -> Option[(V, String)]
3. type Lexer[V] (String) -> Option[(V, String)]
4.
5. fn parse[V](self : Lexer[V], str : String) -> Option[(V, String)] {
6.   (self.0)(str)
7. }
```

- 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

- 最简单的解析器
 - 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

```
1. fn pchar(predicate : (Char) -> Bool) -> Lexer[Char] {
2.   Lexer(fn(input) {
3.     if input.length() > 0 && predicate(input[0]) {
4.       Some(input[0], input.to_bytes().sub_string(1, input.length() - 1))
5.     } else {
6.       None
7.     } }) }
```

- 例如

```
1. fn init {
2.   debug(pchar(fn{ ch => ch == 'a' }).parse("asdf")) // Some(('a', "sdf"))
3.   debug(pchar(fn{
4.     'a' => true
5.     _   => false
6.   }).parse("sdf")) // None
7. }
```

词法分析

- 单词定义：数字或左右括号或加减乘除

```

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

```

- 分析运算符、括号、空白字符等

```

1. let symbol: Lexer[Char] = pchar(fn{
2.     '+' | '-' | '*' | '/' | '(' | ')' => true
3.     _ => false
4. })
5. let whitespace : Lexer[Char] = pchar(fn{ ch => ch == ' ' })

```


- 如果解析成功，对解析结果进行转化

```
1. fn map[I, O](self : Lexer[I], f : (I) -> O) -> Lexer[O] {
2.   Lexer(fn(input) {
3.     // 非空为Some(v)中的v, 空值直接返回
4.     let (value, rest) = self.parse(input)?
5.     Some(f(value), rest)
6.   }) }
```

- 分析运算符、括号并映射为对应的枚举值

```
1. let symbol: Lexer[Token] = pchar(fn{
2.   '+' | '-' | '*' | '/' | '(' | ')' => true
3.   _ => false
4. }).map(fn{
5.   '+' => Plus;    '-' => Minus
6.   '*' => Multiply; '/' => Divide
7.   '(' => LParen;  ')' => RParen
8. })
```

解析器组合子

- 解析 `a`，如果成功再解析 `b`，并返回 `(a, b)`

```

1. fn and[V1, V2](self : Lexer[V1], parser2 : Lexer[V2]) -> Lexer[(V1, V2)] {
2.   Lexer(fn(input) {
3.     let (value, rest) = self.parse(input)?
4.     let (value2, rest2) = parser2.parse(rest)?
5.     Some((value, value2), rest2)
6.   }) }

```

- 解析 `a`，如果失败则解析 `b`

```

1. fn or[Value](self : Lexer[Value], parser2 : Lexer[Value]) -> Lexer[Value] {
2.   Lexer(fn (input) {
3.     match self.parse(input) {
4.       None => parser2.parse(input)
5.       Some(_) as result => result
6.     } }) }

```

解析器组合子

- 重复解析 `a`，零或多次，直到失败为止

```
1. fn many[Value](self: Lexer[Value]) -> Lexer[List[Value]] {
2.   Lexer(fn(input) {
3.     let mut rest = input
4.     let mut cumul = List::Nil
5.     while true {
6.       match self.parse(rest) {
7.         None => break
8.         Some(value, new_rest) => {
9.           rest = new_rest
10.          cumul = Cons(value, cumul) // 解析成功添加内容
11.        } } }
12.     Some(reverse_list(cumul), rest) // ⚠List是栈，需要翻转以获得正确顺序
13.   }) }
```

词法分析

- 整数分析

```
1. // 通过字符编码将字符转化为数字
2. let zero: Lexer[Int] =
3.   pchar(fn{ ch => ch == '0' }).map(fn{ _ => 0 })
4. let one_to_nine: Lexer[Int] =
5.   pchar(fn{ ch => ch.to_int() >= 0x31 && ch.to_int() <= 0x39 }).map(fn { ch => ch.to_int() - 0x30 })
6. let zero_to_nine: Lexer[Int] =
7.   pchar(fn{ ch => ch.to_int() >= 0x30 && ch.to_int() <= 0x39 }).map(fn { ch => ch.to_int() - 0x30 })
8.
9. // number = %x30 / (%x31-39) * (%x30-39)
10. let value: Lexer[Token] =
11.   zero.or(
12.     one_to_nine.and(zero_to_nine.many()) // (Int, List[Int])
13.     .map(fn{ // 1 2 3 -> 1 * 100 + 2 * 10 + 3
14.       (i, ls) => fold_left_list(ls, fn {i, j => i * 10 + j }, i)
15.     })
16.   ).map(Token::Value)
```

- 对输入流进行分析
 - 在单词之间可能存在空格

```
1. let tokens: Lexer[List[Token]] =  
2.   number.or(symbol).and(whitespace.many())  
3.   .map(fn { (symbols, _) => symbols }) // 忽略空格  
4.   .many()  
5.  
6. fn init {  
7.   debug(tokens.parse("-10123-+-523 103    ( 5) ) "))  
8. }
```

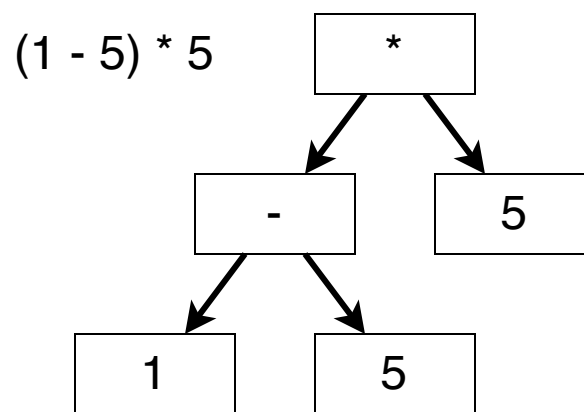
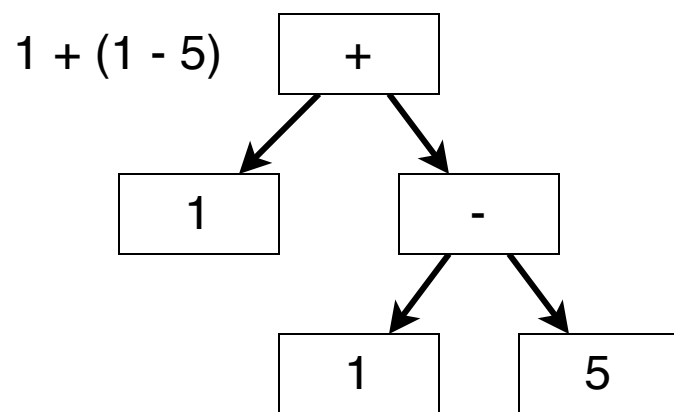
- 我们成功地分割了字符串
 - - 10123 - + - 523 103 (5))
 - 但这不符合数学表达式的语法

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： $a + b \times c \rightarrow a + (b \times c)$
- 结合性： $a + b + c \rightarrow (a + b) + c$
- 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

- 修改后的语法定义

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

- 数据结构

```
1. enum Expression {
2.   Number(Int)
3.   Plus(Expression, Expression)
4.   Minus(Expression, Expression)
5.   Multiply(Expression, Expression)
6.   Divide(Expression, Expression)
7. }
```

语法解析

- 定义语法解析组合子

```
1. type Parser[V] (List[Token]) -> Option[(V, List[Token])]
2.
3. fn parse[V](self : Parser[V], tokens : List[Token]) -> Option[(V, List[Token])] {
4.   (self.0)(tokens)
5. }
```

- 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

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 Parser::ref[Value](ref: Ref[Parser[Value]]) -> Parser[Value] {  
2.   Parser(fn(input) {  
3.     ref.val.parse(input)  
4.   })  
5. }
```

- `ref.val` 将在使用时获取，此时已更新完毕

- 延迟定义

```
1. fn parser() -> Parser[Expression] {
2.   // 首先定义空引用
3.   let expression_ref : Ref[Parser[Expression]] = { val : Parser(fn{ _ => None }) }
4.
5.   // atomic = Value / "(" expression ")"
6.   let atomic = // 利用引用定义
7.     (lparen.and(ref(expression_ref)).and(rparen).map(fn { ((_ , expr), _) => expr}))
8.     .or(number)
9.
10.  // combine = atomic *( "*" / "/" ) atomic)
11.  let combine = atomic.and(multiply.or(divide).and(atomic).many()).map(fn {
12.    ...
13.  })
14.
15.  // expression = combine *( "+" / "-" ) combine)
16.  expression_ref.val = combine.and(plus.or(minus).and(combine).many()).map(fn {
17.    ...
18.  })
19.
20.  expression_ref.val
21. }
```

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

```
1. fn recursive_parser() -> Parser[Expression] {
2.   // 定义互递归函数
3.   // atomic = Value / "(" expression ")"
4.   fn atomic(tokens: List[Token]) -> Option[(Expression, List[Token])] {
5.     lparen.and(
6.       Parser(expression) // 引用函数
7.     ).and(rparen).map(fn { ((_, expr), _) => expr})
8.     .or(number).parse(tokens)
9.   }
10.  fn combine(tokens: List[Token]) -> Option[(Expression, List[Token])] { ... }
11.  fn expression(tokens: List[Token]) -> Option[(Expression, List[Token])] { ... }
12.
13.  // 返回函数代表的解析器
14.  Parser(expression)
15. }
```

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 {  
2.     number(Int) -> Self  
3.     op_add(Self, Self) -> Self  
4.     op_sub(Self, Self) -> Self  
5.     op_mul(Self, Self) -> Self  
6.     op_div(Self, Self) -> Self  
7. }
```

- 接口的不同实现即是对行为的不同语义

语法树之外：Tagless Final

- 我们利用行为的抽象定义解析器

```

1. fn recursive_parser[E : Expr]() -> Parser[E] {
2.   let number : Parser[E] = ptoken(fn { Value(_) => true; _ => false})
3.     .map(fn { Value(i) => E::number(i) }) // 利用抽象的行为
4.
5.   fn atomic(tokens: List[Token]) -> Option[(E, List[Token])] { ... }
6.   // 转化为 a * b * c * ... 和 a / b / c / ...
7.   fn combine(tokens: List[Token]) -> Option[(E, List[Token])] { ... }
8.   // 转化为 a + b + c + ... 和 a - b - c - ...
9.   fn expression(tokens: List[Token]) -> Option[(E, List[Token])] { ... }
10.
11.   Parser(expression)
12. }
13.
14. // 结合在一起
15. fn parse_string[E : Expr](str: String) -> Option[(E, String, List[Token])] {
16.   let (token_list, rest_string) = tokens.parse(str)?
17.   let (expr, rest_token) : (E, List[Token]) = recursive_parser().parse(token_list)?
18.   Some(expr, rest_string, rest_token)
19. }

```

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

```
1. enum Expression { ... } derive(Debug) // 语法树实现
2. type BoxedInt Int derive(Debug) // 整数实现
3. // 实现接口 (此处省略其他方法)
4. fn BoxedInt::number(i: Int) -> BoxedInt { BoxedInt(i) }
5. fn Expression::number(i: Int) -> Expression { Number(i) }
6. // 解析
7. debug((parse_string_tagless_final("1 + 1 * (307 + 7) + 5 - (3 - 2)"))
8.      : Option[(Expression, String, List[Token]])) // 获得语法树
9. debug((parse_string_tagless_final("1 + 1 * (307 + 7) + 5 - (3 - 2)"))
10.     : Option[(BoxedInt, String, List[Token]])) // 获得计算结果
```


总结

- 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