语法解析器

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 Number(1) Plus Number(5) RParen Multiply Number(7) Divide Number(2)`
  - 转化为抽象语法树

    `Division(Multiply(Add(Number(1), Number(5)), Number(7)), Number(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**

词法分析

idea
基础软件中心

- 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"` -> `[ Number(12), Plus, Number(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 : [0-9]+;
2. LPAREN : '(';
3. RPAREN : ')';
4. ADD    : '+';
5. SUB    : '-';
6. MUL    : '*';
7. DIV    : '/';
8. WS     : [ \t\r\n]+ -> skip
```

词法分析

- 算术表达式的词法定义

```
1. NUMBER : [0-9]+;
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+`：重复匹配规则a，匹配1或多次
  - `[0-9]`：0到9中的字符
- 单词定义

```
1. enum Token {
2.   Number(Int); LParen; RParen; Add; Sub; Mul; Div
3. } derive(Show)
```

词法分析

简单状态机

```
struct Context { str : String, mut offset : Int, array : Array[Token] }

fn lex(context : Context) -> Unit {
  let { offset, str, array} = context
  if offset >= str.length() {
    return
  }
  match str[offset] {
    '+' => { array.push(Add); context.offset += 1; lex(context) }
    '-' => { array.push(Sub); context.offset += 1; lex(context) }
    // * / ( )
    ' ' | '\n' | '\t' | '\r' => { context.offset += 1; lex(context) }
    ch => if ch >= '0' && ch <= '9' { lex_number(context) } else { panic() }
  }
}
```

词法分析

idea
基础软件中心

简单状态机

```
fn lex_number(context : Context) -> Unit {
  let { offset, str, array } = context
  let number = "\{str[offset]}"
  context.offset += 1
  lex_number_rest(context, number)
}

fn lex_number_rest(context : Context, number : String) -> Unit {
  let { offset, str, array } = context
  if offset >= str.length() {
    array.push(Number(@strconv.parse_int?(number).unwrap()))
    return
  }
  let ch = str[offset]
  if ch >= '0' && ch <= '9' {
    context.offset += 1
    lex_number_rest(context, number + ch.to_string())
  } else {
    context.array.push(Number(@strconv.parse_int?(number).unwrap()))
    lex(context)
  }
}
```

词法分析

我们成功地分割了字符串

```
1. test {
2.   let array = []
3.   lex({ str: "-10123 -+ - 523 103 (5))", offset: 0, array })
4.   inspect!(array, content="[Sub, Number(10123), Sub, Add, Sub, Number(523), Number(103), LParen, Number(5), RParen, RParen]")
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. expr = NUMBER          | '(' expr ')'
2.      | expr '+' expr | expr '-' expr
3.      | expr '*' expr | expr '/' expr
4.      ;
```

- 上述规则
  - `NUMBER` 表示匹配单词 `NUMBER`
  - `'(' expr ')'` 表示顺序匹配左括号、表达式、右括号
  - `... | ...` 表示匹配多种规则中的任意一个
- 通常分为自顶向下或自底向上

语法分析

- 语法定义

```
1. expr = NUMBER          | '(' expr ')'
2.      | expr '+' expr | expr '-' expr
3.      | expr '*' expr | expr '/' expr
4.      ;
```

- 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：$\mathtt{a}+\mathtt{b}+\mathtt{c}\rightarrow(\mathtt{a}+\mathtt{b})+\mathtt{c}$

语法分析

- 修改后的语法定义

```
1. atomic     : NUMBER  | '(' expr ')'
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（对于自顶向下解析器）

语法分析

- 修改后的语法定义

```
atomic     : NUMBER  | '(' 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)
} derive(Show)
```

语法解析

定义语法解析组合子

```
1. // V 代表解析成功后获得的值
2. type Parser[V] (ArrayView[Token]) -> (V, ArrayView[Token])?
3.
4. fn parse[V](
5.   self : Parser[V],
6.   tokens : ArrayView[Token]
7. ) -> (V, ArrayView[Token])? {
8.   (self._)(tokens)
9. }
```

我们忽略处理报错信息以及错误位置

最简单的解析器

判断下一个元素是否符合条件，符合则读取并前进

```
fn ptoken(predicate : (Token) -> Bool) -> Parser[Token] {
  fn {
    [hd, .. as tl] =>
      if predicate(hd) {
        Some((hd, tl))
      } else {
        None
      }
    [] => None
  }
}
```

解析结果转化

如果解析成功，对解析结果进行转化

```
fn map[I, O](self : Parser[I], f : (I) -> O) -> Parser[O] {
  fn {
    input =>
      match self.parse(input) {
        Some((token, rest)) => Some((f(token), rest))
        None => None
      }
  }
}
```

顺序解析

解析 `a`，如果成功再解析 `b`，并返回 `(a, b)`

```
fn and[V1, V2](
  self : Parser[V1],
  parser2 : Parser[V2]
) -> Parser[(V1, V2)] {
  fn {
    input =>
      self
      .parse(input)
      .bind(
        fn {
          (value, rest) =>
            parser2
            .parse(rest)
            .map(fn { (value2, rest2) => ((value, value2), rest2) })
        },
      )
  }
}
```

尝试解析

解析 `a`，如果失败则解析 `b`

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

解析器组合子

- 重复解析 `a`，零或多次，直到失败为止

```
fn many[Value](
  self : Parser[Value]
) -> Parser[Array[Value]] {
  fn(input) {
    let cumul = []
    let mut rest = input
    loop self.parse(input) {
      None => Some((cumul, rest))
      Some((v, rest_)) => {
        cumul.push(v)
        rest = rest_
        continue self.parse(rest_)
      }
    }
  }
}
```

递归定义

- 递归组合：`atomic = Number | '(' expr ')';`
  - 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

```
fn Parser::ref[Value](
  ref : Ref[Parser[Value]]
) -> Parser[Value] {
  fn { input => ref.val.parse(input) }
}
```

  - `ref.val` 将在使用时获取，此时已更新完毕

递归定义

idea
基础软件中心

延迟定义

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

  // atomic : Number | LParen expr RParen;
  let atomic =  // 利用引用定义
    (lparen.and(ref(expression_ref)).and(rparen).map(fn { ((_, expr), _) => expr}))
      .or(number)

  // combine : atomic ( (Mul | Div) atomic)*;
  let combine = atomic.and(multiply.or(divide).and(atomic).many()).map(fn {
    ...
  })

  // expression : combine ( (Add | Sub) combine)*;
  expression_ref.val = combine.and(plus.or(minus).and(combine).many()).map(fn {
    ...
  })

  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，将入口函数装进结构体

```
1. // 定义互递归函数
2. fn atomic(tokens: ArrayView[Token]) -> (Expression, ArrayView[Token])? {
3.   lparen.and(
4.     expression // 引用函数，newtype 自动类型转换
5.   ).and(rparen).map(fn { ((_, expr), _) => expr})
6.     .or(number).parse(tokens)
7. }
8. fn combine(tokens: ArrayView[Token]) -> (Expression, ArrayView[Token])? { ... }
9. fn expression(tokens: ArrayView[Token]) -> (Expression, ArrayView[Token])? { ... }
10.
11. // 返回函数代表的解析器
12. let parser : Parser[Token] = Parser(expression)
```

总结


```
test {
  // 字符串输入
  let input = "1 + 2 - 3"
  // 词法解析
  let tokens = []
  lex({ str: input, offset: 0, array: tokens })
  inspect!(
    tokens,
    content="[Number(1), Add, Number(2), Sub, Number(3)]",
  )
  // 语法解析（自顶向下的解析器组合子）
  let (expr, _) = parser.parse(tokens[:]).unwrap()
  inspect!(
    expr,
    content="Minus(Plus(Number(1), Number(2)), Number(3))",
  )
}
```