


Compilers - Principles, Techniques and Tools (2nd)

Chapter 2: A Simple Syntax-Directed Translator

2023-02-07

本章的主要内容是开发一个 Java 程序, 将下图左侧的展示性编程语言 (representative programming language) 转换为右侧三地址中间指令表示.

<pre>{ int i; int j; float[100] a; float v; float x; while (true) { do i = i+1; while (a[i] < v); do j = j-1; while (a[j] > v); if (i >= j) break; x = a[i]; a[i] = a[j]; a[j] = x; } }</pre>		<pre>1: i = i + 1 2: t1 = a [i] 3: if t1 < v goto 1 4: j = j - 1 5: t2 = a [j] 6: if t2 > v goto 4 7: ifFalse i >= j goto 9 8: goto 14 9: x = a [i] 10: t3 = a [j] 11: a [i] = t3 12: a [j] = x 13: goto 1 14:</pre>
--	--	--

1. 语法定义

文法 (grammar)

编程语言使用语法 (syntax) 来描述语言的标准形式, 而使用语义 (semantic) 来表示语言的含义. 为了能够描述语法, 我们引入一种叫做上下文无关文法 (CFG, context free grammar), 这种文法实际上就是 BNF (Backus-Naur Form). 举个例子, java 的 if-else 语句其标准格式是这样的

```
"if" "(" expression ")" statement "else" statement
```

这种格式可以使用下面的规则 (CFG) 来表示

```
stmt -> "if" "(" <expr> ")" <stmt> "else" <stmt>
```

这样的规则叫做生成式 (production), `if`, `(`, `)` 和 `else` 表示终结符 (terminal), `expr` 和 `stmt` 是非终结符. CFG 一般由如下 4 个部分组成

- 终结符集合

终结符通常也被称作 token, 是程序语言的基本组成元素, 一个 token 通常包含名称和属性 (指向符号表) 两部分.

- 非终结符集合

非终结符也通常被称作为句法变量 (syntactic variable), 每个非终结符都是一个终结符集合.

- 生成式集合

生成式箭头左侧部分叫做 head 或者 left side, 右侧部分叫做 body 或者 right side. head 为终结符, body 为终结符或者非终结符.

- 起始符号

从终结符中选择一个作为起始符号, 程序语言的语法定义由起始符开始, 起始符对应的生成式就是起始生成式.

看下面一个例子

```
list -> list "+" digit
list -> list "-" list
list -> digit
digit -> "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

这个例子表示由数字和加减符号组成的表达式, 第一个生成式总是表示 CFG 的起始符号, 由双引号括起来的表示终结符, 其他表示非终结符, 多个可选项使用竖线分割, 因此上面的 CFG 和下面的写法等价

```
list -> list + digit | list - list | digit
digit -> "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

要解析一个 CFG, 就是从起始生成式开始解析所有的非终结符集直到整个程序解析完毕, 如果无法解析, 则表示程序编写有误.

CFG 解析之后会生成一个解析树 (parse tree), 一般包含如下属性

- 根结点是起始符
- 所有的叶结点都是终结符
- 所有的中间结点都是非终结符

- 如果 A 是一个非终结符, 且 X_1, X_2, \dots, X_n 是 A 的从左到右排列的孩子结点, 那么必有生成式 $A \rightarrow X_1 X_2 \dots X_n$.

这里面涉及到了数据结构 tree 中的一些常用概念, 如下所示

- 中间结点 (interior node)

含有 1 个或者多个孩子的结点

- 叶子结点 (leaf node)

没有孩子的结点

- 后代 (descendant)

结点 N 的后代包含 N , N 的孩子结点, N 的孩子结点的孩子结点, 以此类推.

- 祖先 (ancestor)

如果 M 是 N 的后代, 那么我们说 N 是 M 的祖先.

优先级和关联性

给定一个文法和一个由终结符组成的字符串, 如果该文法可以生成不止一种解析树, 那么这个文法就是有歧义 (ambiguous) 的, 在设计文法时要注意避免这种歧义.

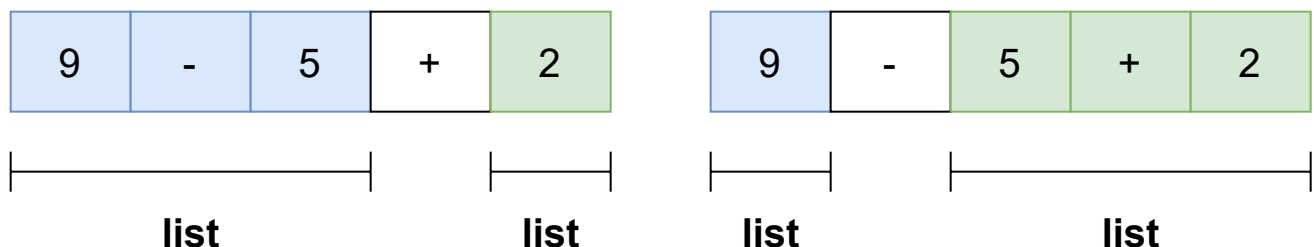
考虑前面的文法例子

```
list -> list + digit | list - list | digit
digit -> "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

可以看到实际上 list 最终也是由 digit 组成的, 似乎可以不考虑 list 和 digit 的区别, 于是上面的文法可以简写为

```
list -> list + list | list - list | "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

看起来好像没问题, 但是这个文法是有歧义的. 比如 $9-5+2$ 可以用下两种不同的方式解释



后者得到的结果是 2 显然是不正确的. 为了解决这种模糊, 像 $9-5+2$ 中的 5 必须解决一个问题, 什么问题呢? 即 5 到底跟两侧操作符中的哪一个更亲近一点, 按照我们的算术规则 5 应该和 - 更亲近一点. 在编程语言中, 这种规则被称之为关联性 (associativity). 一般的, 四则运算中的操作符都是左关联 (left-associative), 也就是说如果一个操作数两侧都有加减乘除运算符, 那么该操作符属于其左侧的操作符.

举个例子, C 中的 = 是右关联操作符号, 所以 $a=b=c$ 等价于 $a=(b=c)$, 这种关联性可以用下面这种文法来表示

```
right -> letter = right | letter
letter -> "a" | "b" | ... | "z"
```

在第一条生成式的 body 中, $\text{letter} = \text{right}$ 就表示 letter 属于右侧的赋值号 = (因为右关联性就是说, 以操作数为中心, 如果其两侧有操作符号, 那么该操作树就属于右侧的操作符).

尽管关联性可以解决 $9-5+2$ 这种问题, 但是对于 $9+5*2$ 则无济于事. 因为按照左关联性, 那么 $9+5*2$ 将被解释为 $(9+5)*2$, 这个问题可以通过优先级来解决. 看下面一个文法定义

```
expr -> expr + term | expr - term | term
term -> term * factor | term / factor | factor
factor -> digit | "(" expr ")"
```

这个文法假定 +, -, *, / 都是左关联性, 并且 +, - (这两者优先级相同) 的优先级低于 *, / (这两者优先级相同). 那么这个文法是怎么得到的呢?

通常情况下, 我们在定义的时候是从基本单元开始定义, 四则运算中的一个基本单元就是一个数值, 或者一个括号括起来的表达式, 即有

```
factor -> digit | "(" expr ")"
```

每个优先级一般由一个非终结符表示, 且从高优先级的开始, 于是我们需要再定义两个终结符, 先来看高优先级的终结符, 即乘除运算, 有

```
term -> term * factor
      | term / factor
      | factor
```

首先 `term -> term * factor` 中写作 `term * factor` 和之前定义 `letter = right` 是异曲同工的, 因为 `*` 是左关联性, 所以操作符 `factor` 属于其左侧的 `*`; 其次, 在当前优先级生成的 `body` 中, 一般会包含本优先级的所有操作符号, 因此我们出列 `term * factor` 之外, 还有一个 `term / factor`; 最后, 当前优先级生成式的 `body` 中, 还会包含上一级的非终结符, 所以我们能看到还有一个 `factor` .

使用这种方法, 我们可以很快得出加减的非终结符, 也就是

```
expr -> expr + term
      | expr - term
      | term
```

将所有构造的过程逆序编写, 就得到了文法定义.

再来看一个例子, 下面的文法定义了语句的标准形式

```
stmt -> id "=" expression ";"
      | "if" "(" expression ")" stmt
      | "if" "(" expression ")" stmt else stmt
      | "while" "(" expression ")" stmt
      | "do" stmt "while" "(" expression ")" ";"
      | "{" stmts "}"
stmts -> stmts stmt
      | ε
```

这里的 `id` 表示一个合法的标志符, `stmts` 表示一个可能为空的语句列表, 而 `stmt` 表示一个语句. 我们注意到 `stmt` 中没有以 `stmt` 结尾的都是以分号结尾的, 这是因为分号用来分隔语句块, 因此没有以分号结尾的项最终都将归结到分号结尾的项.

2. 语法翻译

制导翻译

语法制导翻译是通过在文法生成式中加入规则或者程序片段来完成的, 主要涉及到如下两个概念

- 属性 (attributes): 属性是指与程序结构相关的任何确定值 (quantity)

比如表达式的数据类型, 生成代码的指令数等.

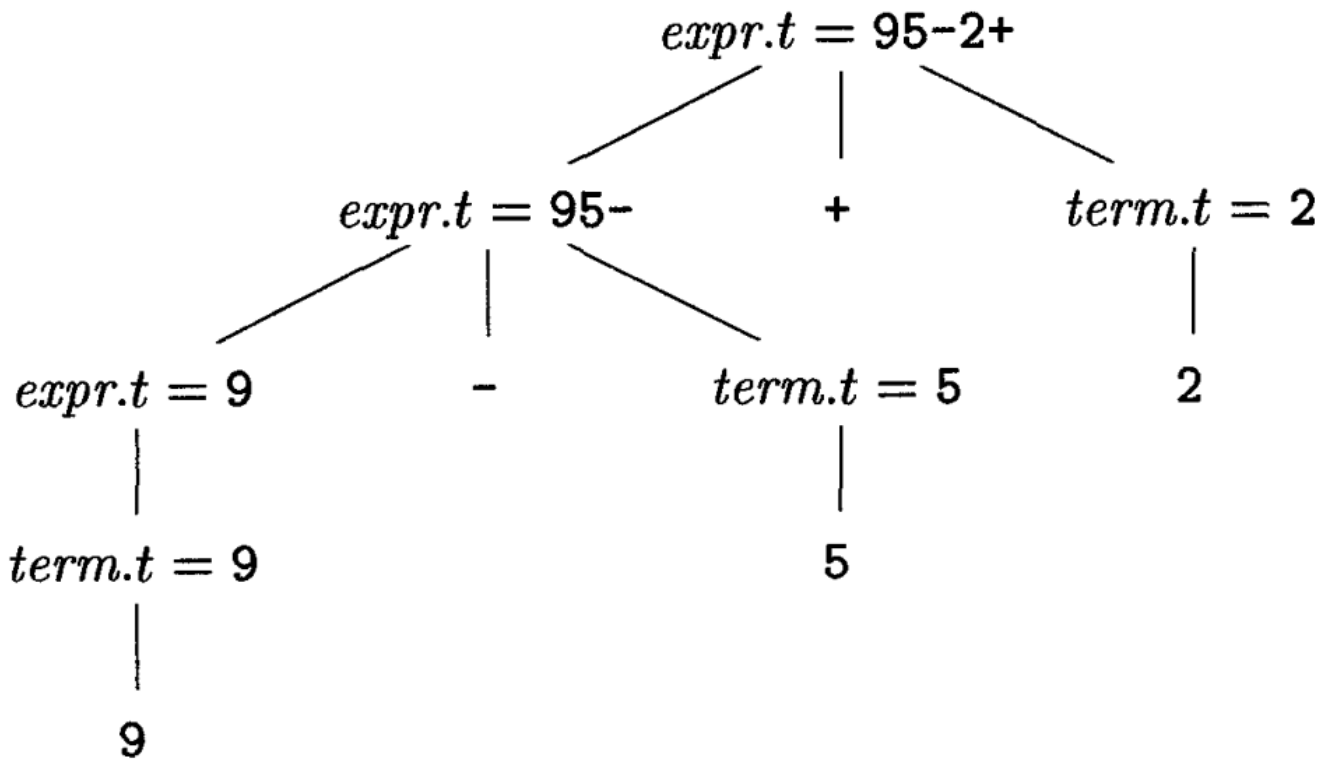
- 翻译方案 (translation scheme)

一个翻译方案是一种标记法, 用来将程序段附加到文法生成式上, 附加的程序段将会在语法分析器分析到该文法时执行.

我们将这两个概念统一称作语法指导定义 (syntax-directed definition). 本节将以中缀表达式到后缀表达式的转换为例, 说明语法制导的相关概念, 我们首先给出后缀表达式的定义

- 如果 E 是一个变量或者常量, 那么 E 的后缀表达式就是 E 自身
- 如果 E 是一个表达式 $E_1 \text{op} E_2$, 其中 op 是任意操作符, 那么 E 的后缀表达式是 $E'_1 E'_2$, 其中 E'_1, E'_2 分别是 E_1, E_2 的后缀表达式
- 如果 E 是一个括号括起来的表达式 (E_1) , 那么 E 的后缀表达式和 E_1 一致

看下面一棵后缀表达式解析树



这里非终结符 expr 和 term 都有一个属性 t , 根节点 $95-2+$ 就是 $9-5+2$ 的后缀表达式. 可以看到树中标注了每个节点的属性值, 我们将这样的树叫做注解解析树 (annotated parse tree).

特别地, 如果解析树中节点 N 的属性值是由 N 以及 N 的孩子节点属性值决定, 那么我们就说该属性是合成属性 (synthesized attributes). 类似地, 继承属性 (inherited attribute) 是指节点的属性值由节点自身, 父节点, 以及兄弟节点属性值决定的属性.

这棵注解解析树的生成方法由下表给出, 这个表实际上就是中缀到后缀的语法制导定义

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

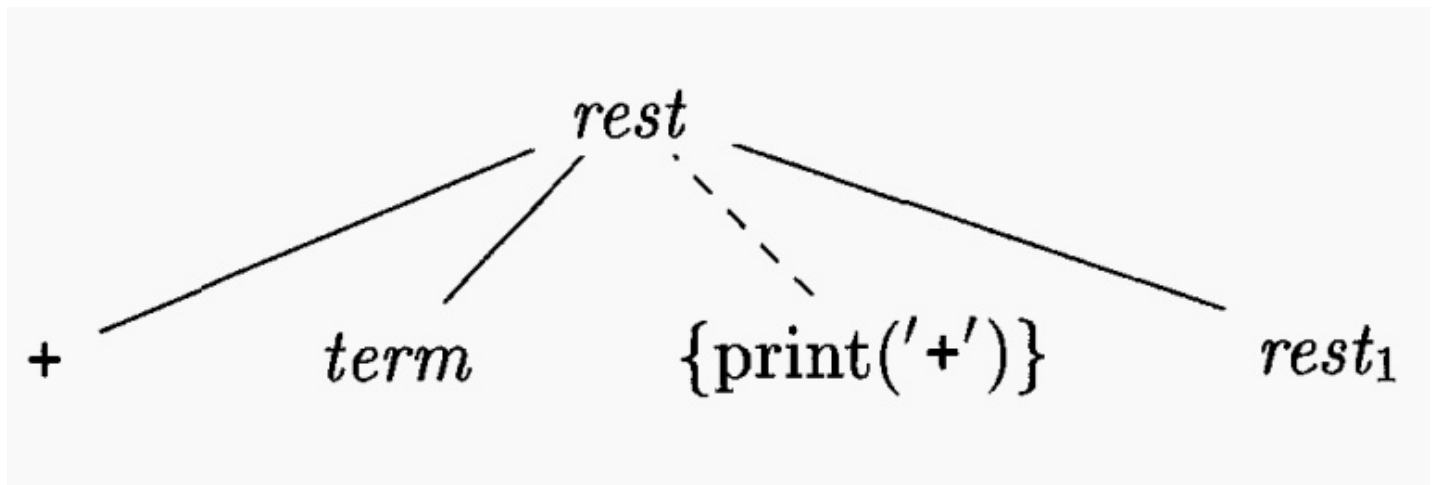
由于数字的后缀表达式就是其自身, 所以 $term \rightarrow 9$ 的翻译规则就是 $term.t = '9'$. 再比如 $expr \rightarrow expr_1 + term$ 根据后缀表达式的定义, 我们的翻译规则就是 $expr.t = expr_1.t \parallel term.t \parallel '+'$, 这里面 \parallel 表示连接符号. 在此图中, 我们能注意到每一个生成式的翻译规则, 其组成中的非终结符顺序和生成式 body 中的非终结符顺序是一致的 (非终结符之间可以插入一些字符串), 我们将具有这种属性的语法制导定义称作为 simple.

自增翻译

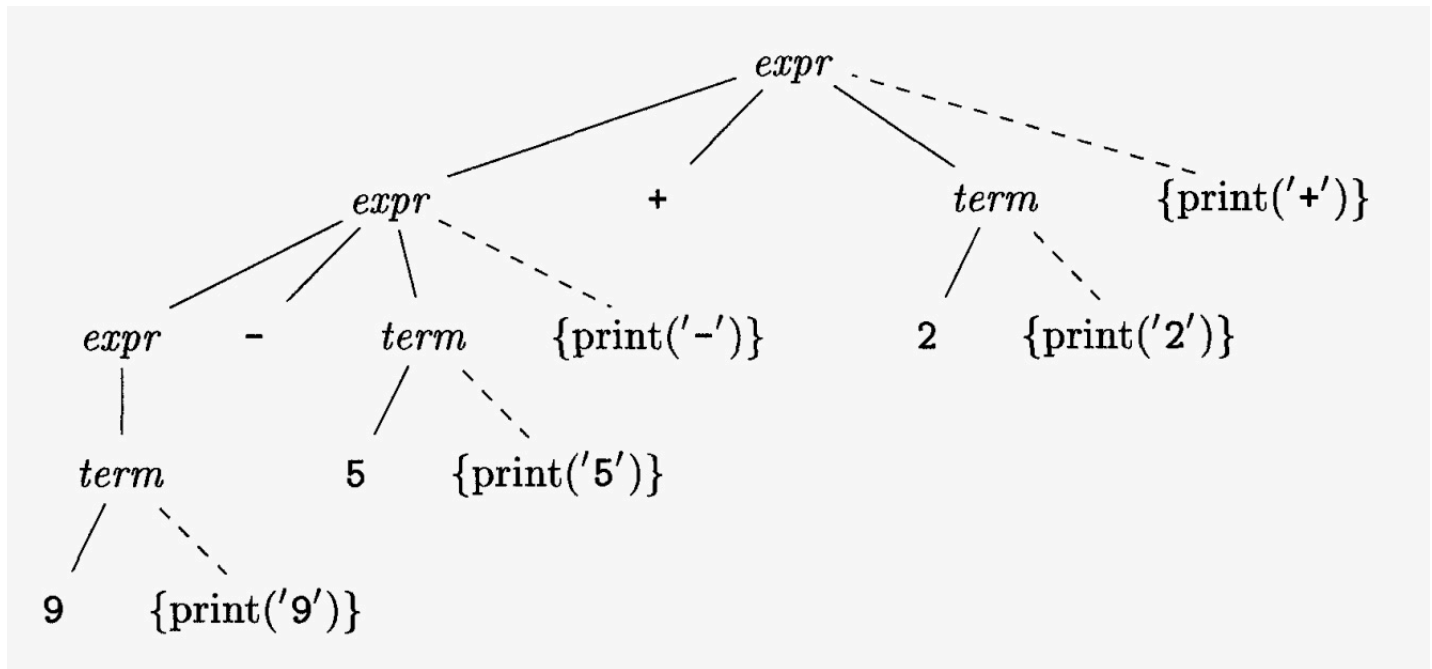
语法制导翻译是通过在生成树节点中加入属性来实现翻译, 还有一种可选方法自增翻译. 自增翻译通过在生成式 body 中加入要执行的程序片段 (我们将其称作 semantic actions), 然后指定遍历解析数的熟悉, 从而实现翻译的目的. 看下面一个例子

```
rest -> + term {print('+')} rest1
```

要执行的程序片段用花括号括起来, 比如这里要执行的代码就是 `print('+')`. 自增翻译的解析树中, 程序片段单独使用一个节点表示, 该节点没有孩子节点, 和其父节点的连线使用虚线表示, 如下图所示



以语法制导翻译中的 9-5+2 为例说明该自增翻译实现中缀到后缀的翻译过程, 该表达式对应的解析树如下所示



自增翻译必须明确指定树的遍历方式, 要实现中缀到后缀翻译, 必须使用深度遍历的后序遍历法, 依次遍历到节点顺序如下

```

{print('9')}
{print('5')}
{print('-')}
{print('2')}
{print('+')}
  
```

于是 9-5+2 的后缀表达式为 95-2+ . 该树对应的文法如下图所示

$expr$	\rightarrow	$expr_1 + term$	$\{\text{print}('+')\}$
$expr$	\rightarrow	$expr_1 - term$	$\{\text{print}('-')\}$
$expr$	\rightarrow	$term$	
$term$	\rightarrow	0	$\{\text{print}('0')\}$
$term$	\rightarrow	1	$\{\text{print}('1')\}$
		...	
$term$	\rightarrow	9	$\{\text{print}('9')\}$

3. 解析 (Parsing)

解析就是指根据给定文法生成终结字符串的方法, 本节将采用递归下降 (recursive descent) 的方法来说明解析过程和语法制导翻译的实现.

对于任意文法, 要解析包含 n 个终结符的字符串, 一定存在解析时间不超过 $O(n^3)$ 的解析器, 虽然从理论上来说 $O(n^3)$ 难以忍受, 但幸运的是, 在实际的编程语言实践中, 解析器一般都是很快的, 所以这个问题不用担心.

解析时一般要么是从根节点到叶子节点进行, 或者从叶子节点到根节点进行, 因此根据方向的不同, 前者被称作自顶向下 (top-down), 后者被称作自低向上 (bottom-up). top-down 的方法比较流行, 因为手写起来比较方便, bottom-up 方法则是在自动生成解析器时用的比较多, 因为自动生成的解析器可以处理规模较大的文法和翻译规则.

预测性解析 (predictive parsing) 是一种简单的递归下降解析方法, 这种方法通过向前查看符号来确定处理生成式 body 的具体过程. 假设给定的文法如下

```

stmt -> expr ;
      | if ( expr ) stmt
      | for ( optexpr; optexpr; optexpr ) stmt
      | other

optexpr -> ε
         | expr

```

要处理的字符串为

```
for(; expr; epxr) other
```

那么预测性解析首先要做的就是判定当前字符串的符号能和生成式中的那一项匹配上. 我们记 α 为某个终结符或者非终结符, 并使用 $FIRST(\alpha)$ 表示第一个 α 所能推导出来的第一个终结符 (对于非终结符, 需要递归的往下推, 直到出现第一个终结符为止).

在上面的文法符号中, 我们通过手动推导, 可以得到如下结果

```
FIRST(stmt) = { epxr, if, for, other }
```

那么解析上面的字符串时, 我们第一个符号遇到的是 `for`, 于是我们能匹配到 `stmt` 的如下 `body`

```
for (optexpr; optexpr; optexpr) stmt
```

于是按照这个 `body` 的组成继续解析即可, 完整的解析如下所示

```
void stmt() {
    switch(lookahead) {
        case expr:
            match(expr); match(';');
            break;
        case if:
            match(if); match('('); match(expr); match(')'); stmt();
            break;
        case for:
            match(for); match('('); optexpr(); match(';') optexpr(); match(';') optexpr(); mat
            break;
        case other:
            match(other);
            break;
        default:
            report("syntax error");
    }
}

void optexpr() {
    if (lookahead == expr) match(expr);
}

void match(terminal t) {
    if (lookahead == t) lookahead = nextTerminal;
```

```

    else report("syntax error");
}

```

注意 `optexpr` 函数, 当匹配完 `for(` 之后, 接着匹配的是 `;`, 在 `optexpr` 函数中仅仅是消耗该符号, 其他什么也不做, 这就对应于 `optexpr` 文法中的 ϵ 生成式 `body`.

一般地, 在 predictive parser 中, 每一个非终结符 A 的解析例程 (procedure) A_p 做了两件事情

- 根据当前的符号判断是否应该使用 A , 只要当前符号位于 $FIRST(\alpha)$ 中 (其中 α 是生成式 A 的 `body`), 那么就说明应该使用该 A_p .
- A_p 会依次解析 α , 如果遇到一个非终结符, 那么就调用该非终结符的 procedure 进行解析, 如果遇到的是终结符, 那么会匹配该终结符之后, 继续读取下一个符号进行后续解析.

但是这种解析方法可能导致解析器陷入死循环, 比如下面这种生成式

```

expr -> expr + term

```

这种文法中 `expr` 的 `body` 中最左侧的项也是 `expr`, 我们将这种文法称之为左递归 (left recursive) 文法. 对于这种表达式, 相应的 procedure 可能如下所示

```

expr() {
    expr();
    if (token == '+') {
        getNextToken();
    }
    term();
}

```

可以看到这种方式会导致 `expr` 递归的调用自己, 但是当前 `token` 并没有向前有任何移动, 因此陷入死循环. 可以通过重写生成式消除左递归, 考虑下面这个生成式

$$A \rightarrow A\alpha \mid \beta$$

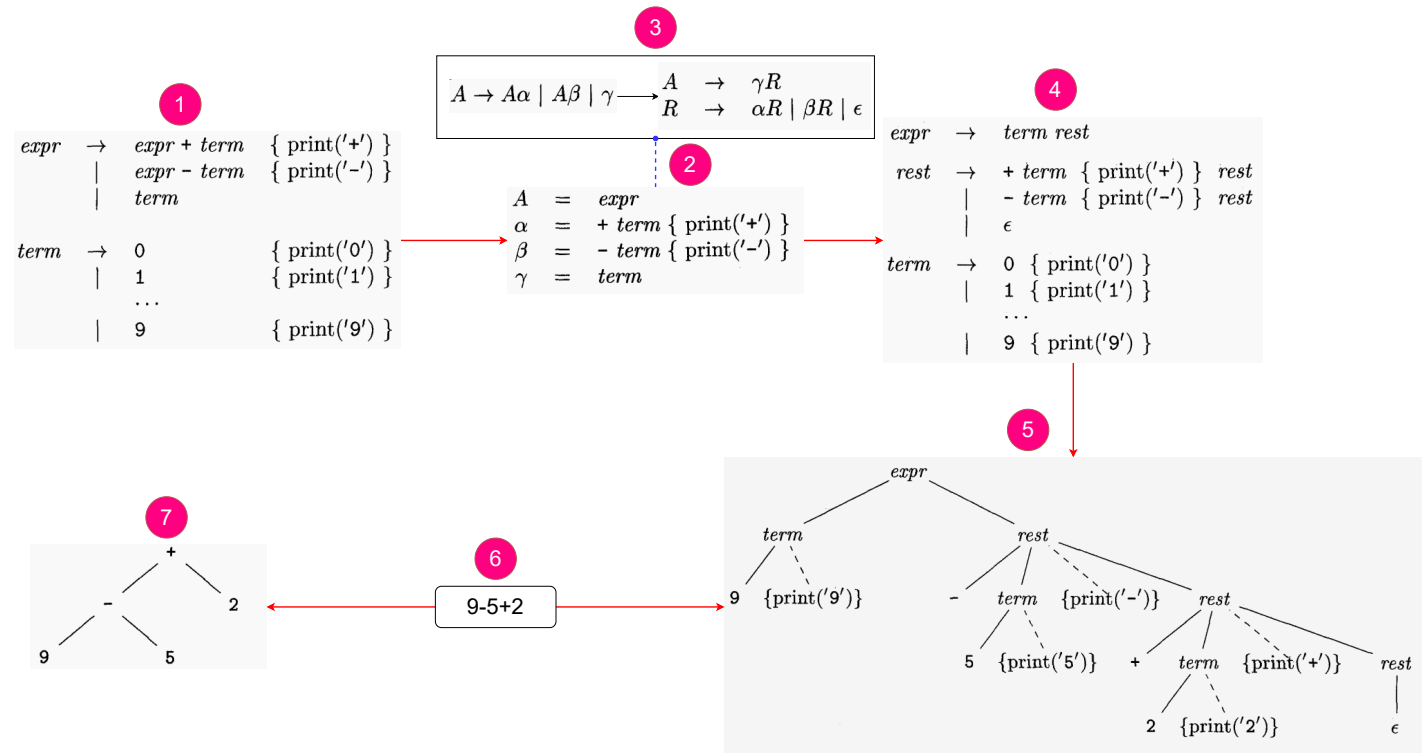
其中 α 和 β 都是由终结符序和非终结符组成的序列且不以 A 开头, 这个左递归可以通过引入一个非终结符 R 进行消除

$$\begin{aligned}
 A &\rightarrow \beta R \\
 R &\rightarrow \alpha R \mid \epsilon
 \end{aligned}$$

这个表达式的一般生成方式将会在后面讲解.

简单表达式的翻译器

本节将会利用前面几节讲到的知识来实现一个可用的表达式翻译器, 涉及的内容如下图所示



1 表示翻译器的自增翻译文法定义, 但是该文法定义包含左递归的情况, 所以需要消除左递归, 消除方法在 2 和 3 处给出, 消除左递归后得到的文法位于 4 中. 通过 4 中的文法, 我们可以得到 5 处的自增翻译解析树, 该解析树也包含了要翻译的表达式, 即 6 中的 `9-5+2`. 可以看到图中的 5 和 7 其实都是一棵树, 通常我们将 5 称作解析树 (parse tree), 将 7 称作语法树 (syntax tree), 这两者的区别也是很明显的, `parse tree` 的内部结点是非终结符, 而语法树中的内部结点为操作符 (operator).

纸上得来终觉浅, 绝知此事要躬行, 所以让我们根据上图实现一个 translator 吧, 首先我们给出程序的伪代码, 做一些简单的讲解, 最后给出实际能运行的代码 (本文我将使用 rust 编写, 原书使用 java).

伪代码如下所示

```
// expr -> term rest
void expr() {
    term(); rest();
}

// rest -> + term { print('+' ) } rest
//         | - term { print('-' ) } rest
//         | ε
void rest() {
    if (lookahead == '+') {
        match('+'); term(); print('+'); rest();
    }
}
```

```

    } else if(lookahead == '-') {
        match('-'); term(); print('-'); rest();
    } else {
        // ε
    }
}
void term() {
    if (lookahead is a digit) {
        t = lookahead;
        match(lookahead);
        print(t);
    } else {
        report("syntax error");
    }
}
}

```

可以看到一旦定义了文法, 那么写程序其实是顺其自然的事情. 上面的代码中 `expr()` 和 `rest()` 是比较好理解的, `term()` 中稍微有点难理解的就是为什么要首先保留一下 `lookahead` 之后再打印, 这是因为 `match` 会修改 `lookahead` 的值.

这个程序我们可以对其做一点优化, 可以看到 `rest()` 函数内部不管走到哪一条分支, 最终执行的都是 `rest()` 函数. 如果函数的最后一条语句仍然是调用自身, 那么我们就说这种调用是尾递归调用 (tail recursive). 如果尾递归调用所在的函数没有参数, 那么该递归可以简单的被替换一个跳转语句, 直接跳转到函数头即可. 因此 `rest()` 函数可以被重写为如下

```

void rest() {
    while(true) {
        if (lookahead == '+') {
            match('+'); term(); print('+'); continue;
        } else if(lookahead == '-') {
            match('-'); term(); print('-'); continue;
        }
        break;
    }
}

```

当 `rest()` 函数被重写之后, `rest()` 函数就没必要存在了, 因为该函数只在 `expr()` 中进行了调用, 因此直接把 `rest()` 替换到 `expr()` 中即可.

我们用 rust 实现的代码如下所示

```

pub struct Translator {
    pub stream: Vec<char>,
    pub index: usize,
}

```

```

impl Translator {
  pub fn new<S: AsRef<str>>(data: S) -> Self {
    Self {
      stream: data.as_ref().chars().collect(),
      index: 0,
    }
  }

  pub fn translate(&mut self) {
    self.expr();
  }

  pub fn next(&mut self) -> Option<char> {
    self.stream.get(self.index).and_then(|c| { self.index += 1; Some(*c) })
  }

  fn term(&mut self) {
    let ch = self.next().expect("unexpected EOF");
    if !ch.is_ascii_digit() {
      println!("digit expected");
      std::process::exit(1);
    }
    print!("{ch}");
  }

  fn expr(&mut self) {
    self.term();
    while let Some(ch) = self.next() {
      match ch {
        '+' | '-' => {
          self.term();
          print!("{ch}");
        }
        _ => break
      }
    }
  }
}

fn main() {
  let mut translator = Translator::new("9-5+2");
  translator.translate();
}

```

词法分析

词法分析的作用是将输入的文本转换为一个个 token 对象, 本节提出的词法分析器将支持数字, 标识符, 以及空白字符, 其文法可以认为是对前一节文法的扩展, 如下所示

```
expr -> expr + term { print('+') }
      | expr - term { print('-') }
      | term

term -> term * factor { print('*') }
      | term / factor { print('/') }
      | factor

factor -> (expr)
        | num { print(num.value) }
        | id { print(id.lexeme) }
```

这个文法定义中, factor 可以是一个数字 num 或者一个标识符 id, num 和 id 都是终结符, 这些终结符在词法分析器分析之后会得到一个对应的 token, token 会带有一些属性, 比如这里 num 有一个 value 属性, 就表示该数字的具体值, 而 id 或有一个 lexeme 属性, lexeme 就是指该 token 对应的原始字符串, 比如 `int count = 10;` 这里面 count 就属于 id, 其 lexeme 属性值就是 `count`, 而 `10` 对应的则是 num, 其 value 属性值就是 `10`.

词法分析器在确定 token 时需要预读一些字符串, 通常的做法是使用一个专用的缓存区, 每次读入一个缓存区, 然后根据该缓存区确定要生成的是什么 token.

词法分析器所处理的 token 类型一般有常量, 以及标识符等, 在处理的过程中会忽略注释和空白字符, 一些特殊的标识符由编译器使用, 这些标识符称之为关键字, 区分关键字和标识符需要借助一个哈希表来实现, 这些基本理论可以参考下图加深理解

1

```

for ( ; ; peek = next input character ) {
    if ( peek is a blank or a tab ) do nothing;
    else if ( peek is a newline ) line = line+1;
    else break;
}

```

2

31 + 28 + 59

```

if ( peek holds a digit ) {
    v = 0;
    do {
        v = v * 10 + integer value of digit peek;
        peek = next input character;
    } while ( peek holds a digit );
    return token (num, v);
}

```

<num, 31> <+> <num, 28> <+> <num, 59>

3

count = count + increment;

```

if ( peek holds a letter ) {
    collect letters or digits into a buffer b;
    s = string formed from the characters in b;
    w = token returned by words.get(s);
    if ( w is not null ) return w;
    else {
        Enter the key-value pair (s, <id, s>) into words
        return token (id, s);
    }
}

```

<id, "count"> <=> <id, "count"> <+> <id, "increment"> <;>

1 处是忽略空格的伪代码, 2 处是解析常量 token, 3 处是解析标识符以及区分关键字和普通标识符的方法 (words 是一个哈希表), 在此基础上我们将实现一个词法分析器, 代码如下所示

```

use std::collections::HashMap;

#[derive(Debug, Clone)]
pub enum Token {
    Num(Num),
    Word(Word),
    Char(char),
}

#[derive(Debug, Clone, Copy)]
pub struct Num {
    tag: i32,
    value: i32,
}

impl Num {
    pub const TAG: i32 = 256;
    pub fn new(value: i32) -> Self {
        Self {
            tag: Num::TAG,
            value,
        }
    }
}

```



```

#[derive(Debug, Clone)]
pub struct Word {
    tag: i32,
    lexeme: String,
}

impl Word {
    pub const ID: i32 = 257;
    pub const TRUE: i32 = 258;
    pub const FALSE: i32 = 259;

    pub fn new<S: AsRef<str>>(tag: i32, value: S) -> Self {
        Self {
            tag,
            lexeme: value.as_ref().to_owned(),
        }
    }
}

pub struct Lexer {
    line: i32,
    words: HashMap<String, Token>,
    buffer: Vec<char>,
    cursor: usize,
}

impl Lexer {
    pub fn new<S: AsRef<str>>(buffer: S) -> Self {
        Self {
            line: 1,
            words: {
                let mut mp = HashMap::new();
                let word = Word::new(Word::TRUE, "true");
                mp.insert(word.lexeme.clone(), Token::Word(word));
                let word = Word::new(Word::FALSE, "false");
                mp.insert(word.lexeme.clone(), Token::Word(word));
                mp
            },
            buffer: buffer.as_ref().chars().collect(),
            cursor: 0,
        }
    }

    pub fn scan(&mut self) -> Option<Token> {
        if self.cursor >= self.buffer.len() {
            return None;
        }

        let blank = &self.buffer[self.cursor..].iter().take_while(|&&x| {

```

```

        match x {
            ' ' | '\t' | '\n' => {
                if x == '\n' {
                    self.line += 1;
                }
                true
            },
            _ => false,
        }
    }).collect::<Vec<&char>>();
    self.cursor += blank.len();

    if let Some(peek) = self.buffer.get(self.cursor) {
        if peek.is_digit(10) {
            let mut value = 0;
            self.buffer[self.cursor..].iter()
                .take_while(|&c| c.is_digit(10))
                .for_each(|x| {
                    value = value * 10 + x.to_digit(10).unwrap() as i32;
                    self.cursor += 1;
                });
            return Some(Token::Num(Num::new(value)));
        }

        if peek.is_ascii_alphabetic() {
            let mut identifier = String::new();
            self.buffer[self.cursor..].iter()
                .take_while(|&c| {
                    c.is_ascii_alphanumeric() || c == '_'
                })
                .for_each(|&c| {
                    identifier.push(c);
                    self.cursor += 1;
                });
            if let Some(v) = self.words.get(identifier.as_str()) {
                return Some(v.clone());
            } else {
                let word = Token::Word(Word::new(Word::ID, identifier.clone()));
                self.words.insert(identifier.clone(), word.clone());
                return Some(word);
            }
        }

        self.cursor += 1;
        return Some(Token::Char(*peek));
    }

    None
}

```

```

}

fn main() {
    let mut lexer = Lexer::new("int a = 10 + 3;\nbool is_digit = false;");
    while let Some(tk) = lexer.scan() {
        println!("{tk:?}");
    }
}

```

输出如下所示

```

Word(Word { tag: 257, lexeme: "int" })
Word(Word { tag: 257, lexeme: "a" })
Char('=')
Num(Num { tag: 256, value: 10 })
Char('+')
Num(Num { tag: 256, value: 3 })
Char(';')
Word(Word { tag: 257, lexeme: "bool" })
Word(Word { tag: 257, lexeme: "is_digit" })
Char('=')
Word(Word { tag: 259, lexeme: "false" })
Char(';')

```

符号表

符号表 (symbol table) 是一个数据结构, 用于存储源程序的结构信息, 在编译器的各个阶段中, 会自增的将收集到的信息填充到符号表中. 本节将以下面的代码为例

```

{ int x; char y; { bool y; x; y; } x; y; }

```

通过符号表分析, 移除所有的声明, 然后分析各个变量的类型, 也就是说对于上面的代码我们将得到以下结果

```

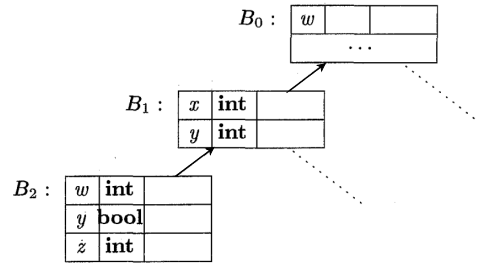
{ { x:int; y:boo; } x:int; y:char; }

```

每一个符号声明都对应一个作用域 (scope), 作用域本质上是程序的一个范围, 更准确来说是某个块 (block) 的全部或者一部分. 因为同一个符号可能在多个块中声明, 这就导致同一个符号可能被多次使用, 但是却表示完全不同的符号, 因此每个块都独占一个符号表, 对于嵌套块, 内部符号表通过指针指向外部符号表, 对于所有嵌套块, 会形成一个符号表链表, 如下图所示

2

1) { int x_1 ; int y_1 ;
 2) { int w_2 ; bool y_2 ; int z_2 ;
 3) ... w_2 ...; ... x_1 ...; ... y_2 ...; ... z_2 ...;
 4) }
 5) ... w_0 ...; ... x_1 ...; ... y_1 ...;
 6) }

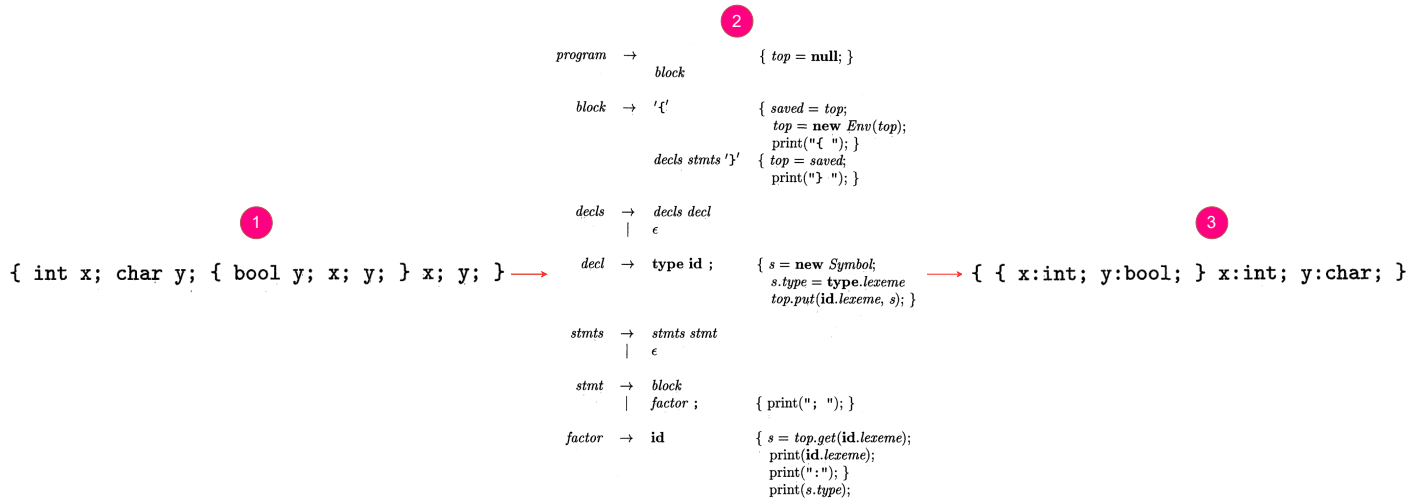


1 表示代码的嵌套块结构, 其下标仅仅表示变量声明所在的行号 (因此 y_1 和 y_2 表示名称都是 y 的变量). B_1 表示 1 中从行 1 开始的块 (范围为 $[1, 6]$), 而 B_2 表示从行 2 开始的块 (范围为 $[2, 4]$), B_0 表示代码的顶层作用域 (顶层块).

当对 1 中的代码分析后, 得到的词法表将由子块指向父块, 最终形成一个符号表链表, 2 中的箭头就表明了这一结果.

每进入一个块, 当遇到一个变量 x 时, 我们会从当前的符号表开始往上寻找, 一旦找到 x 的声明或定义, 那么我们就获取 x 的相关信息, 这种方法叫做最近嵌套原则 (most-closely nested rule), 从这个角度来看, 我们也可以认为符号表承载了信使的角色, 即将声明处的信息传递给使用者.

根据这些基本原理, 本节开始处的问题可以通过如下自增翻译文法解决



这个文法看上去有点复杂, 让我们一步步来解析它.

1. 进入最外层的块 B_1 , 所以我们打印一个 {

2. 接着是 int x ; 这是一个 decl, 所以我们解析 type id; 将 int x ; 放到 B_1 的符号表 S_1 中, 对应的是 char y ; 也是同样的操作, 现在 S_1 中的符号表大概是这样的

```
// S1
{
  "x": {
    "type": "int",
    "lexme": "x",
  }
  "y": {
    "type": "int",
    "lexme": "x",
  }
  "parent": null,
}
```

3. 现在进入内层块 B_2 , 打印 { 并新建一个符号表 S_2 , 遇到 `bool y;` 将其加入 S_2 , 于是 S_2 变成

```
// S2
{
  "y": {
    "type": "bool",
    "lexme": "y",
  }
  "parent": "S1",
}
```

4. 接着遇到一个变量 `x`, 对应于 `factor`, 于是从当前符号表 S_2 开始向上找, 在 S_2 中没有 `x`, 于是去其父块也就是 S_1 中找, 找到了 `x`, 于是打印 `x.lexme : x.type`, `factor` 解析完之后回到了 `stmt` 并打印 `;`, 即有 `x: int;`.

5. 类似地重复以上步骤得到最终结果.

4. 中间代码生成

有两种中间代码表示形式, 一种是树 (包括 `parse tree` 和 `abstract syntax tree`), 另一种是线性表示, 比如 `三地址代码 (three-address code)`.

- 语法树的构建

语法树可以表示任何结构, 每个结构使用一个节点表示, 其孩子节点表示该结构语义上的任意子结构. 比如说 C 中的 `while` 语句, 其组成如下所示

```
while (expr) stmt
```

该语句对应的语法树节点就是一个 while 节点, 且 while 节点包含两个孩子节点, 即 expr 和 stmt. 在代码层次上, 这样的节点可以用如下代码表示

```
new While(x, y)
```

其中 While 对应操作符 while, x 和 y 分别对应 while 的两个孩子节点.

一般地, 对于任意一条语句, 我们都需要提取出对应的操作符 (operator) 和操作数 (operand). 如果一条语句以关键字开始, 那么该关键字将被用作操作符, 比如 while 语句对应 While 操作符, do 语句对应 Do 操作符. 条件语句则使用 Ifelse 和 If 操作符来表示 (我们没有使用 Else 操作符, 因为 Else 操作符会引起一些解析问题, 在后面的章节中我们会对其介绍). 如果一条语句不是以关键字开头, 那么我们使用 Eval 操作符来表示这样的语句.

此外, 我们定义 Seq 操作符, 用来表示一系列的语句, 由于块语句本质上就是一些列语句的组合, 所以我们也使用 Seq 操作符来表示块语句.

对于表达式, 其语法树基于如下映射表进行构造

CONCRETE SYNTAX	ABSTRACT SYNTAX
=	assign
	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	op
* / %	op
!	not
⁻ <i>unary</i>	minus
[]	access

对该表有如下说明

- 左侧一列是具体的表达式操作符, 右侧是映射到到抽象语法树中的节点名.

- 表达式操作符中除了 `=` 是右结合之外, 其余均是左结合.
- 从上到下操作符的优先级是自增的, 比如 `==` 的优先级比 `&&` 和 `=` 要高.

在构建语法树的时候, 会对代码进行检查, 这种检查叫做静态检查 (static checking), 静态检查一般包括两方面: 语义检查 (syntactic checking) 和类型检查 (type checking).

语义检查就是检查代码有没有错误, 比如同一个作用域内, 标志符只能被声明一次, 声明多次就是非法的. 再比如 `break` 语句必须包含在循环或者 `switch` 语句中. 另外一个常见的例子是左值 (L-values) 和右值 (R-values) 的概念, 一般来讲, 左值是指 "位置", 而右值是指 "值", 左右分别可以看作是赋值号 (`=`) 两侧的对象. 语义检查的时候必须确保左值是一个位置, 比如说 `i = 5` 中的 `i` 就是一个左值, 而 `2 = 5` 中的 `2` 就不是一个左值.

类型检查是检查操作符的操作数类型是否正确, 比如执行一个整数和浮点数的加法操作, 如果 `+` 操作符要求两侧的操作符类型必须一致, 那么这样的代码就是错误的. 但是通常我们在写 C 代码时, 这样的代码并没有报错, 为啥呢? 编译器会自动将两边的操作数转换为一致的类型后, 再执行加法操作, 这种自动将操作数转换为某一类型的操作就叫做协变 (coercion).

- 三地址代码 (Three-Address Code)

语法树构建成功后, 我们就可以通过语法树中每个节点的属性和代码段来分析并生成三地址代码. 三地址代码的操作数少于或等于 3 个, 主要有如下几种形式

```
// 标准形式
x = y OP z
```

```
// 数组
x[y] = z
x = y[z]
```

```
// 普通赋值
x = y
```

- 语句翻译

语句翻译主要涉及到控制流的问题, 假设要翻译的语句为 `if expr then stmt`, 那么对应的伪代码如下所示

```

class If extends Stmt {
    Expr E; Stmt S;
    public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }
    public void gen() {
        Expr n = E.rvalue();
        emit( "ifFalse" + n.toString() + " goto" + after);
        S.gen();
        emit(after + ":");
    }
}

```

这里的 If 类就是 if 语句对应的节点, 它的构造函数有 2 个参数, 分别表示 If 的两个孩子节点.

每一个节点都有一个 gen 方法, 用来生成三地址指令序列, 在 If 的 gen 方法中, 我们首先获取 If 表达式节点的右值, 生成一条跳转三地址指令, 如下所示

```
ifFalse x goto after
```

这条三地址指令表示, 如果 x 为假, 则跳转到标签为 after 处, 在生成该指令之后, 我们调用语句 S 的 gen 方法, 生成它的三地址指令.

最后我们生成 after 标签.

○ 表达式翻译

在语句翻译中, 我们看到对表达式 E 的翻译是通过调用其 rvalue 方法完成的, 为了减少复杂度, 我们只介绍包含二元操作符, 数组访问, 赋值, 常量以及标识符的表达式翻译方法, 特别地, 在数组访问中, 比如 $y[z]$ 这种形式, 我们只考虑 y 是一个标识符的情形, 更复杂的情形我们将在后面的章节中介绍.

标识符和常量表达式是不需要处理得, 因为它们可以直接出现在三地址表示中. 现在考虑节点 x 包含二元操作符, 那么在在翻译过程中需要借助一个中间变量, 比如 $i - j + k$ 将会被翻译为如下三地址序列

```

t1 = i - j
t2 = t1 + k

```

而 $2 * a[i]$ 将会被翻译为如下

```

t1 = a[i]
t2 = 2 * t1

```


但是如果我们要给 $a[i]$ 赋值, 那么我们不能简单的用临时变量来替代 $a[i]$, 因此我们需要区分左值和右值, 由此我们引出两个方法 $lvalue()$ 和 $rvalue()$.

$lvalue$ 的实现如下所示

```
Expr lvalue(x : Expr) {  
    if (  $x$  is an Id node ) return  $x$ ;  
    else if (  $x$  is an Access( $y, z$ ) node and  $y$  is an Id node ) {  
        return new Access( $y, rvalue(z)$ );  
    }  
    else error;  
}
```

如果结点 x 是一个标志符, 那么 x 的左值就是 x 自身, 因此直接返回即可. 如果结点 x 表示的是数组访问, 比如 $a[i]$, 那么 x 的抽象语法可表示为 $Access(y, z)$, 其中 y 表示数组名, z 表示下标. 这种情况下, 我们只需要获取 z 的右值即可, 然后返回元素的地址 $y[rvalue(z)]$ 即可.

比如说 x 表示 $a[2*k]$, 那么调用 $lvalue(x)$ 后, 开始执行 $new Access(a, rvalue(2 * k))$, 这条语句会先执行 $rvalue(2 * k)$, 而 $rvalue(2 * k)$ 实际上会生成如下表达式

```
t = 2 * k
```

其中 t 是一个临时变量, 于是上面的表达式返回的结点就是

```
new Access(a, t)
```

因此新返回的节点表示一个地址 $a[t]$, 即我们预期的左值.

$rvalue$ 的实现如下所示

```

Expr rvalue(x : Expr) {
    if ( x is an Id or a Constant node ) return x;
    else if ( x is an Op(op, y, z) or a Rel(op, y, z) node ) {
        t = new temporary;
        emit string for t = rvalue(y) op rvalue(z);
        return a new node for t;
    }
    else if ( x is an Access(y, z) node ) {
        t = new temporary;
        call lvalue(x), which returns Access(y, z');
        emit string for t = Access(y, z');
        return a new node for t;
    }
    else if ( x is an Assign(y, z) node ) {
        z' = rvalue(z);
        emit string for lvalue(y) = z';
        return z';
    }
}

```

如果 x 是一个常量或者是一个标志符, 那么直接返回 x 即可.

如果 x 表示 $y \text{ op } z$, 那么将其转换为

$$t = \text{rvalue}(y) \text{ op } \text{rvalue}(z)$$

如果 x 表示 $y[z]$, 那么将其转换为

$$\begin{aligned}
 z' &= \text{lvalue}(x) \\
 t &= y[z']
 \end{aligned}$$

比如 $a[j - k]$, 那么 $\text{lvalue}(a[j - k])$ 将会返回 $a[\text{rvalue}(j - k)]$, 而 $\text{rvalue}(j - k)$ 会打印 $t = j - k$, 并返回 t . 所以 $a[\text{rvalue}(j - k)]$ 的最终效果就是打印 $t = j - k$ 并返回 $a[t]$.

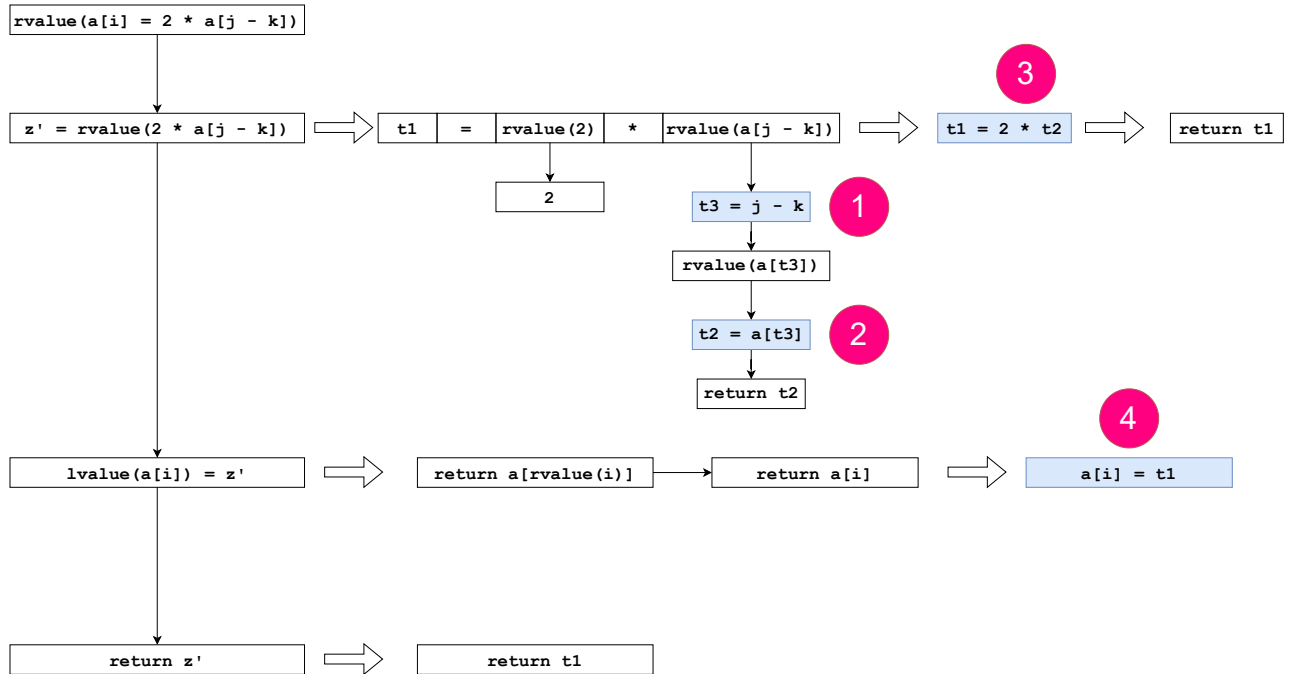
如果 x 表示 $y = z$, 那么将其转换为

```

z' = rvalue(z)
lvalue(y) = z'

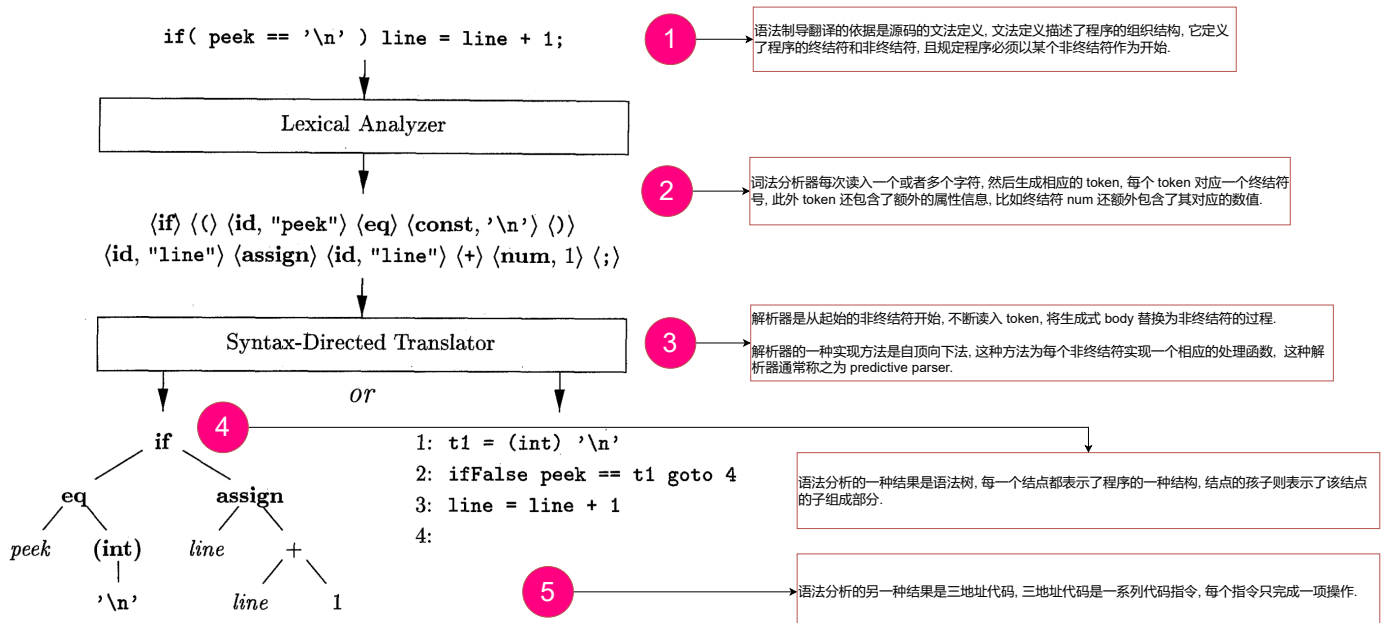
```

现在假设我们要处理 $a[i] = 2 * a[j - k]$ ，这是一个赋值表达式，该表达式的 $rvalue$ 求解过程如下



最终生成的表达式就是图中 1 - 4 所标记的。

5. 总结



POWERED BY [YEW](#) AND [RUST](#)

ICP 备案号: 粤ICP备2021148577号-1  公安备案号: 44030502008292号