编译原理笔记

作者：kenshinl

时间：Last Update: October 6, 2022

版本：0.1

封面图片作者：Elina Bernpaintner

人生代代无穷已，江月年年望相似。

目录

第 1 章 《编译原理第二版》笔记

类型检查

语意分析的一个重要部分是类型检查 (type checking)。编译器检查每个运算符是否具有匹配的运算分量。比如，很多程序设计语言的定义中要求一个数组的下标必须是整数。如果用一个浮点数作为数组下标，编译器就必须报告错误。

C 语言的静态作用域策略

1. 一个 C 程序由一个顶层的变量和函数声明的序列组成。
2. 函数内部可以声明变量，变量包括局部变量和参数。每个这样的声明的作用域被限制在它们所出现的那个函数内。
3. 名字 x 的一个顶层声明的作用域包括其后的所有程序。但是如果一个函数中也有一个 x 的声明，那么函数中的那些语句就不在这个顶层声明的作用域内。

文法定义

一个上下文无关文法 (context-free grammar) 由四个元素组成：

1. 一个终结符号集合，它们有时也称为“词法单元”。终结符号是该文法所定义的语言的基本符号的集合。
2. 一个非终结符号集合，它们有时也被称为“语法变量”。每个非终结符号表示一个终结符号串的集合。
3. 一个产生式集合，其中每个产生式包括一个称为产生式头或左部的非终结符号、一个箭头，和一个称为产生式体或右部的由终结符号及非终结符号组成的序列。产生式主要用来表示某个构造的某种书写形式。如果产生式头非终结符号代表一个构造，那么该产生式体就代表了该构造的一种书写方式。
4. 指定一个非终结符号为开始符号。(2.2.1)

多优先级运算符表达式文法的设计

我们可以将因子 (factor) 理解成不能被任何运算符分开的表达式。不能分开的意思是说当在任意因子的任意一边放置一个运算符，都不会导致这个因子的任何部分分离出来，成为这个运算符的运算分量。当然，因子本身作为一个整体可以成为该运算符的一个运算分量。如果这个因子是一个由括号括起来的表达式，那么括号将起到保护其不被分开的作用。如果因子就是一个运算分量，那么它当然不能被分开。

一个（不是因子的）项（term）是一个可能被高优先级的运算符分开，但不能被低优先级运算符分开的表达式。一个（不是因子也不是项的）表达式可能被任何一个运算符分开。

我们可以把这种思想推广到具有任意 n 层优先级的情况。我们需要 n + 1 个非终结符号。通常，这个非终结符号的产生式体只能是单个运算分量或括号括起来的表达式。然后，对于每个优先级都有一个非终结符，表示能被该优先级或更高优先级的运算符分开的表达式。通常，这个非终结符的产生式有一些产生式体表示了该优先级的运算符的应用。另有一个产生式体只包含了代表更高一级优先级的非终结符号。（2.3.1）

语法制导翻译

语法制导翻译是通过向一个文法的产生式附加一些规则或程序片段而得到的。(2.3)

语法制导翻译相关的两个概念

属性：属性表示与某个程序构造相关的任意的量。属性可以是多种多样的，比如表达式的数据类型、生成的代码中的指令数量或为某个构造生成的代码中第一条指令的位置等。因为我们用文法符号（终结符号或者非终结符号）来表示程序构造，所以我们将属性的概念从程序构造拓展到表示这些构造的文法符号上。

语法制导的翻译方案：翻译方案是一种将程序片段附加到一个文法的各个产生式上的表示法。当在语法分析过程中使用一个表达式时，相应的程序片段就会执行。这些程序片段的执行效果按照语法分析过程的顺序组合起来，得到的结果就是这次分析/综合过程处理源程序得到的翻译结果。(2.3)

语法制导定义 (syntax-directed definition)

把每个文法符号和一个属性集合相关联，并且把每个产生式和一组语义规则 (semantic rule) 想关联，这些规则用于计算与该产生式中符号相关联的属性值。(2.3.2)

如果某个属性在语法分析树节点 N 上的值是由 N 的子节点以及 N 本身的属性值确定的，那么这个属性就称为综合属性 (synthesized attribute)。综合属性具有一个很好的性质：只需要对语法分析树进行一次自底向上的遍历，就可以计算出属性的值。(2.3.2)

后缀表示的归纳定义

1. 如果 E 是一个常量或者变量，则 E 的后缀表示是 E 本身。
2. 如果 E 是一个形如 $E_1\ op\ E_2$ 的表达式，其中 op 是一个二目运算符，那么 E 的后缀表达式是 $E_1^{'}\ E_2^{'}\ op$，这里 $E_1^{'}$ 和 $E_2^{'}$ 分别是 E_1 和 E_2 的后缀表示。
3. 如果 E 是一个形如 (E_1) 的被括号括起来的表达式，则 E 的后缀表示就是 E_1 的后置表示。(2.3.1)

自顶向下分析方法

Listing 1.1: 示例文法

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

optexpr ->
         | expr
```

自顶向下方法构造语法分析树

在自顶向下地构造一棵语法分析树时，从标号为开始非终结符 stmt 的根节点开始，反复执行下面两个步骤：

1. 在标号为非终结符号 A 的节点 N 上，选择 A 的一个产生式，并为该产生式体中的各个符号构造出 N 的子节点。
2. 寻找下一个结点来构造子树，通常选择的是语法分析树最左边的尚未拓展的非终结符。

一般来说，为一个非终结符号选择产生式是一个“尝试并犯错”的过程。也就是说，我们首先选择一个产生式，并在这个产生式不合适时进行回溯，再尝试另一个产生式。一个产生式“不合适”是指使用了该产生式之后，我们无法构造得到一棵与当前输入串相匹配的语法分析树。(2.4.1)

预测分析法

递归下降分析方法 (recursive-descent parsing) 是一种自顶向下的语法分析方法，它使用一组递归过程来处理输入。文法的每个非终结符都有一个相关联的过程。预测分析法 (predictive parsing) 是递归下降分析法的一种简单形式，在预测分析法中，各个非终结符号对应的过程中的控制流可以由向前看符号 (lookahead) 无二义地确定。在分析输入串时出现的过程调用序列隐式地定义了该输入串的一棵语法分析树。如果需要，还可以通过这些过程调用来构建一个显式的语法分析树。(2.4.2)

左递归消除算法

考虑有两个产生式 $A \to A\alpha|\beta$ 的非终结符号 A，其中 α 和 β 是不以 A 开头的终结符号/非终结符符号的序列。因为产生式 $A \to A\alpha$ 的右部的最左符号是 A 自身，非终结符号 A 和它的产生式就称为左递归的 (left recursive)。不断应用这个产生式将在 A 的右边生成一个 α 的序列，当 A 最终被替换为 β 时，就得到一个在 β 后跟有 0 个或多个 α 的序列。预测分析器不能处理左递归的文法。

使用一个新的非终结符号 R，并按照如下方式改写 A 的产生式可以达到相同的效果。

$$A \to \beta R$$
$$R \to \alpha R \mid \epsilon$$

非终结符号 R 和它的产生式 $R \to \alpha R$ 是右递归的 (right recursive), 因为这个产生式的右部的最后一个符号就是 R 本身。(2.4.5)

标志符

字符串可以作为标识符，来为变量、数组、函数等命名。为了简化语法分析器，语言的文法通常把标志符当作终结符号进行处理。当某个标志符出现在输入中时，语法分析器都会得到相同的终结符号。(2.6.4)

符号表

符号表 (symbol table) 是一种供编译器用于保存有关源程序构造的各种信息的数据结构。这些信息在编译器的分析阶段被逐步收集并放入符号表，它们在综合阶段用于生成咪表代码。符号表的每个条目中包含与一个标志符相关的信息，比如它的字符串（或者词素）、它的类型、它的存储位置和其他相关信息。符号表通常需要支持同一标志符在一个程序中的多重声明。(2.7)

语句块的最近嵌套规则

语句块的最近嵌套规则 (most-closely) 规则是说，一个标志符 x 在最近的 x 声明的作用域中。也就是说，从 x 出现的块开始，从内向外检查各个块时找到的第一个对 x 的声明。

实现语句块的最近嵌套规则时，我们可以将符号表链接起来，也就是使得内嵌语句块的符号表指向外围语句块的符号表。(2.7.1)

静态检查

静态检查是指在编译过程中完成的各种一致性检查。包括：

1. 语法检查。语法要求比文法中的要求要更多。例如任何作用域内同一个标识符最多只能被声明一次。
2. 类型检查。一种语言的类型规则确保一个运算符或函数被应用到类型和数量都正确的运算分量上。(2.8.3)

词法单元、模式和词素

1. 词法单元。由一个词法单元名和一个可选的属性值组成。词法单元名是一个表示某种词法单元的抽象符号。词法单元名字是由语法分析器处理的输入符号。
2. 模式。描述了一个词法单元的词素可能具有的形式。
3. 词素。是源程序中的以恶搞字符序列，它和某个词法单元的模式匹配，并被词法分析器识别为该词法单元的一个实例。(3.1.2)

输入缓冲与哨兵标记

由于在编译一个大型源程序时需要处理大量的字符，处理这些字符需要很多的时间，因此开发了一些特殊的缓冲技术来减少用于处理单个输入字符的时间开销。一中重要的机制就是利用两个教徒读入的缓冲区（双缓冲技术）。

每个缓冲区的容量都是 N 个字符，通常 N 是一个磁盘块的大小，可以使用系统读取命令一次将 N 个字符读入到缓冲区中。如果输入文件中的剩余字符不足 N 个，那么就会有一个特殊字符（用 eof 表示）来标记源文件的结束。

程序为输入委维护了两个指针：

1. lexmeBegin 指针，该指针指向当前词素的开始处。
2. forward 指针，它一直向前扫描，直到发现某个模式被匹配为止。一旦确定了下一个词素，forward 指针将指向该词素结尾的字符。词法分析器将这个词素作为某个返回给语法分析器的词法单元的属性值记录下来。然后使 lexmeBegin 指针指向刚刚找到的词素之后的第一个字符。(3.2.1)

如果采用双缓冲方案，那么在每次向前移动 forward 指针时，都必须检查是否到达了缓冲区的末尾。如果是，那么我们必须加载另一个缓冲区。因此每读入一个字符，都需要做两次测试：一次是检查是否到达缓冲区的末尾，另一次是确定读入的字符是什么。如果我们拓展每个缓冲区，在它们末尾包含一个哨兵字符，就可以把对缓冲区末端的测试和对当前字符的测试合二为一。这个哨兵字符必须是一个不会在源程序中出现的特殊字符，一个自然的选择就是字符 eof。(3.2.2)

Listing 1.2: 带有哨兵标记的 forward 指针移动算法

```
switch(*forward++) {
  case eof:
    if (forward 在第一个缓冲区末尾) {
    装载第二个缓冲区;
    forward = 第二个缓冲区的开头;
  } else if (forward 在第二个缓冲区末尾) {
    装载第一个缓冲区;
    forward = 第一个缓冲区的开头;
  } else { // 缓冲区内部的 eof 标记输入结束
    终止词法分析
  }
  break;
  // 其他字符的情况 ...
}
```

语言上的运算的定义

运算	定义和表示
L 和 M 的并	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
L 和 M 的连接	$LM = \{st \mid s \in L \text{ and } t \in M\}$
L 的 Kleene 闭包	$L^* = \cup_{i=0}^{\infty} L^i$
L 的正闭包	$L^* = \cup_{i=1}^{\infty} L^i$

(3.3.2)

Kleene 闭包和正闭包之间的关系

两种闭包见的转换公式

$$r^* = r^+ \mid \epsilon$$
$$r^+ = r^* r$$
$$r^+ = r r^*$$

(3.3.5)

有穷自动机

有穷自动机分为两类：

1. 一个不确定的有穷自动机 (Nondeterministic Finite Automata, NFA) 对其边上的标号没有任何限制。一个符号标记离开同一状态的多条边，并且空串 ϵ 也可以作为标号。
2. 对于每个状态及自动机输入字母表中的每个符号，去定的有穷自动机 (Deterministic Finite Automata, DFA) 有且只有一条离开该状态、以该符号为标号的边。

确定的和不确定的有穷自动机嫩识别的语言的集合是相同的。事实上，这些语言的集合正好是能够用正则表达式描述的语言的集合。这个集合中的语言称为正则语言 (regular language).(3.6)

不确定的有穷自动机

一个不确定的有穷自动机 (NFA) 由以下几个部分组成:

1. 一个有穷的状态集合 S。
2. 一个输入符号集合 Σ。
3. 一个转换函数 (transition function)，它为每个状态和 $\Sigma \cup \epsilon$ 中的每个符号都给出了相应的后继状态 (next state) 的集合。
4. S 中的一个状态 d_0 被指定为开始状态，或者说初始状态。
5. S 的一个子集 F 被指定为接受状态 (或者说终止状态) 集合。

不管是 NFA 还是 DFA，我们都可以将它表示为一张转换图 (transition graph)。图中的结点是状态，带有标号的边表示自动机的转换函数。从状态 s 到状态 t 存在一条标号为 a 的边当且仅当状态 t 是状态 s 在输入 s 上的后继状态之一。这个图与状态转换图十分相似，但是：

1. 同一个符号可以标记从同一状态出发到达多个目标状态的多条边。
2. 一条边的标号不仅可以是输入字母表中的符号，也可以是空符号串 ϵ。(3.6.1)

转换表

我们也可以将一个 NFA 表示为一张转换表 (transition table), 表的各行对应于状态，割裂对应于输入符号和 ϵ。对应于一个给定状态和给定输入的条目是将 NFA 的转换函数应用于这些参数后得到的值。如果转换函数没有给出对应于” 某个状态- 给定输入” 对的信息，我们就把 $\emptyset$ 放入相应的表项中。

转换表的优点是我们可以很容易地确定和一个给定状态和一个输入符号相对应的转换。它的缺点是如果输入字母表很大，且大多数状态在大多数输入字符上没有转换的时候，转换表需要占用大量空间。(3.6.2)

确定的有穷自动机

确定的有穷自动机是不确定有穷自动机的一个特例。其中：

1. 没有输入 ϵ 之上的转换动作。
2. 对每个状态 s 和每个输入符号 a，有且只有一条标号为 a 的边离开 s。

在构造词法分析器的时候，我们真正实现或者模拟的是 DFA。(3.6.4)

NFA 构造 DFA 的子集构造 (subset construction) 算法

输入: 一个 NFA N
输出: 一个接受同样语言的 DFA D
方法: 为 D 构造一个转换表 $Dtran$。D 的每个状态是一个 NFA 状态集合，我们将构造 $Dtran$，使得 D 并行地

模拟 N 在遇到一个给定输入串时可能执行的所有动作。我们面对的第一个问题是正确处理 N 的 ϵ 转换。在下图中我们可以看到一些函数的定义。这些函数描述了一些需要在这个算法中执行的 N 的状态集上的基本操作。请注意，s 表示 N 的单个状态，而 T 代表 N 的一个状态集。

操作	描述
ϵ-$closure(s)$	能够从 NFA 的状态 s 开始只通过 ϵ 转换到达的 NFA 状态集合
ϵ-$closure(T)$	能够从 T 中某个 NFA 状态 s 开始只通过 ϵ 转换到达的 NFA 状态集合，即 $\cup_{s\epsilon T}\epsilon$-$closure(s)$
$move(T,a)$	能够从 T 中某个状态 s 出发通过标号为 a 的转换到达的 NFA 状态的集合

我们必须找出当 N 读入了某个输入串之后可能位于的所有状态集合。首先，在读入第一个输入符号之前，N 可以位于集合 ϵ-$closure(s_0)$ 中的任何状态上，其中 s_0 是 N 的开始状态。下面进行归纳。假定 N 在读入输入串 x 之后可以位于集合 T 中的状态上。如果下一个输入符号是 a，那么 N 可以立即移动到集合 $move(T,a)$ 中的任何状态。然而，N 可以在读入 a 后再执行几个 ϵ 转换，因此 N 在读入 xa 之后可位于 ϵ-$closure(move(T,a))$ 中的任何状态上。根据这些思想，我们可以得到如下的方法，该方法构造了 D 的状态集合 $Dstates$ 和 D 的转换函数 $Dtran$。

Algorithm: 算法 1

Input: 一开始，ϵ-$closure(s-0)$ 是 $Dstates$ 中的唯一状态，且它未加标记

Output: None

while 在 $Dstates$ 中有一个未标记状态 T **do**
  给 T 加上标记; **for** 每个输入符号 a **do**
    U = ϵ-$closure(move(T,a))$; **if** U 不在 $Dstates$ 中 **then**
      将 U 加入到 $Dstates$ 中，且不加标记;
    end
    $Dtran[T,a]=U$;
  end
end

D 的开始状态是 ϵ-$closure(S_0)$，D 的接受状态是所有至少包含了 N 的一个接受状态的状态集合。我们只需要说明如何对 NFA 的任何状态集合 T 计算 ϵ-$closure(T)$，就可以完整地描述子集构造法。这个计算过程显示在下述算法中。它是从一个状态集合开始的一次简单的图搜索过程，不过此时假设这个图中只存在标号为 ϵ 的边。

Algorithm: 算法 2

Input: None

Output: None

将 T 的所有状态压入 $stack$ 中;

将 ϵ-$closure(T)$ 初始化为 T;

while $stack$ 非空 **do**

  将栈顶元素 t 弹出栈中;

  for 每个满足如下条件的 u: 从 t 出发有一个标号为 ϵ 的转换到达状态 u **do**

    if u 不在 ϵ-$closure(T)$ 中 **then**

      将 u 加入到 ϵ-$closure(T)$ 中;

      将 u 压入栈中;

    end

  end

end

第 2 章 一些概念和问题

First 集和 Follow 集的作用

算法实现

Algorithm: 求 nullable、FIRST 集和 FOLLOW 集

将所有的 FIRST 和 FOLLOW 集合初始化为空，将所有的 nullable 初始化为 false

for 每一个终结符 Z **do**
  $FIRST[Z] \leftarrow \{Z\}$
end
repeat
  for 每个产生式 $X \rightarrow Y_1 Y_2 \cdots Y_k$ **do**
    for 每个 i 从 1 到 k, 每个 j 从 $i+1$ 到 k **do**
      if 所有 Y_i 都是可为空的 **then**
        $nullable[X] \leftarrow true$
      end
      if $Y_1 \cdots Y_{i-1}$ 都是可为空的 **then**
        $FIRST[X] \leftarrow FIRST[X] \cup FIRST[Y_i]$
      end
      if $Y_{i+1} \cdots Y_k$ 都是可为空的 **then**
        $FOLLOW[Y_i] \leftarrow FOLLOW[Y_i] \cup FOLLW[X]$
      end
      if $Y_{i+1} \cdots Y_{j-1}$ 都是可为空的 **then**
        $FOLLOW[Y_i] \leftarrow FOLLOW[Y_i] \cup FIRST[Y_j]$
      end
    end
  end
until $FIRST$ 、$FOLLOW$ 和 $nullable$ 在此轮迭代中没有改变;

词法分析和语法分析的关联体现在什么地方

左递归带来的问题是什么

要搞清楚这个问题，首先需要知道什么是左递归。左递归的定义如下：

对于上下文无关文法的一个规则来说，如果其右侧第一个符号与左侧符号相同或者能够推导出左侧符号，那么称该规则为左递归的。前一种情况称为直接左递归，后一种情况成为间接左递归。

以一个经典的包含左递归的表达式语法为例。

Listing 2.1: 经典表达式文法

```
1 Goal   -> Expr
```

```
Expr    -> Expr + Term
Expr    |  Expr - Term
        |  Term
Term    -> Term * Factor
Term    |  Term / Factor
        |  Factor
Factor -> ( Expr )
        |  num
        | name
```

左递归是自顶向下语法分析中需要专门处理的一个问题。自定向下语法语法分析是指从语法分析树的根开始，系统地向下拓展树，直至树的叶结点与词法分析器返回地已归类单词相匹配。在过程的每一点上，都需要考虑一个部分完成的语法分析树。过程在树的下边缘选择一个非终结符，选定某个适用于该非终结符的产生式，用与产生式右侧相对应的子树来拓展该结点。终结符是无法拓展的。这个过程会一直持续下去，直到语法分析树的下边缘只包含终结符，且输入流已经耗尽。

从上面的描述中可以看出，自顶向下语法分析的关键就是选择一个合适的非终结符进行拓展。以句子 a+b*c 为例，了解左递归对自顶向下语法分析带来的问题。

Listing 2.2: 自顶向下语法分析 a+b*c

```
规则0  Expr
规则1  Expr + Term
规则1  Expr + Term + Term
规则1  Expr + Term + Term + Term
...
```

在第二行开始，语法分析器知道非终结符 Expr 和输入单词 a，它选择规则 1 进行匹配，在第三行，语法分析器还是面临非终结符 Expr 和输入单词 a，他继续选择规则 1 进行匹配，导致了无限循环。这也就是左递归给自顶向下语法分析带来的问题。在这个例子中，语法分析器向前多看一个单词在这里可以解决问题，但是如果语法足够复杂，左递归仍然会导致这个问题。所以，需要有一个算法来消除左递归。对于直接左递归，可以采用如下的方式。

Listing 2.3: 消除直接左递归

```
Fee -> Fee α
    |  β

转换为如下产生式

Fee  -> β Fee'
Fee' -> α Fee'
     |  ϵ
```

对于间接左递归的消除算法就不在这里描述了。

什么是递归下降分析器