

编译原理

qhy

2018 年 1 月 14 日

目录

1 介绍	3
1.1 词法分析	5
1.2 语义分析	5
1.3 语义分析	5
1.4 中间代码生成 intermediate code generation	6
1.5 代码优化	6
1.6 目标代码生成	6
1.7 表格管理与出错处理	6
1.8 编译程序划分	6
2 文法和语言	6
2.1 语法	7
2.2 语义	7
2.3 文法	7
2.3.1 符号和符号串	7
2.3.2 闭包	8
2.3.3 文法	8
2.3.4 简化表示	9
2.4 归纳与推导	9
2.5 句型、句子、语言	10
3 上下文无关文法及其语法树	11
3.1 规范推导和规范句型	11
3.2 句型的分析	12
4 词法分析	13
4.1 正规文法与正则式	13
4.2 正规文法和正则式的等价性	14
4.3 有穷自动机	15
4.3.1 确定的有穷自动机(DFA)	15
4.3.2 不确定的有穷自动机(NFA)	16
4.3.3 NFA转换为等价的DFA	16

4.3.4 DFA的化简	17
4.3.5 正规式与有穷自动机的等价性	19
4.3.6 正规文法和有穷自动机的等价性	20
4.4 词法分析程序的构造	20
5 自顶向下的语法分析	22
5.1 确定的自顶向下的语法分析	22
5.1.1 非LL(1)文法转LL(1)文法	24
5.2 不确定的自顶向下分析	24
5.3 确定的自顶向下算法-实现	25
5.3.1 递归子程序法	25
5.3.2 预测分析法	25
6 LR分析法	27
6.1 LR(0)文法	29
6.2 SLR(1)文法	34
6.3 LR(1)文法	34
6.4 LALR(1)文法	36
7 中间代码生成	36
7.1 属性文法	36
7.2 语法制导的翻译	36
7.3 中间代码	38
7.4 中间代码生成	38
7.4.1 简单赋值语句的翻译	39
7.4.2 布尔表达式的翻译	40
7.4.3 if,while,repeat 语句的翻译	43
8 符号表	46
9 代码优化	46
9.1 局部优化	47
9.2 循环优化	49

1 介绍

预期收货:

- 通过学习编译原理,写出更高效的代码

- 针对目标,自编写编译器

对某个模型机的编译器进行设计

编译程序:将高级语言书写的程序翻译成等价的低级语言的程序

源程序:编译程序的输入对象

目标程序:编译程序的输入对象

编译器和解释器的区别?

- 编译器是转换器,解释器是执行系统。
- 解释器是源程序的一个执行系统,工作结果得到源程序的执行结果
- 编译器是源程序的转换系统,工作结果得到等价于源程序的某种目标程序

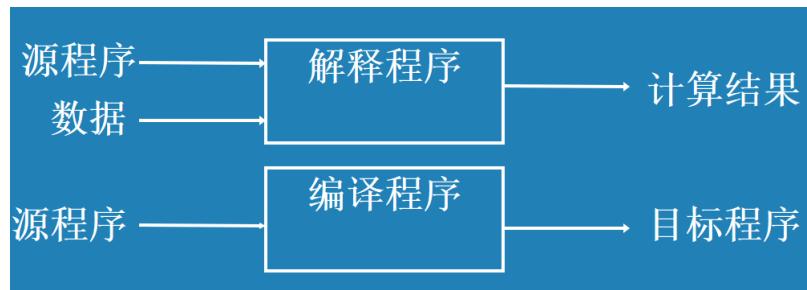


图 1: 编译器和解释器

程序编译的过程主要分为两大阶段:

1. 查错
 - 词法分析
 - 语法分析
 - 语义分析
2. 综合(翻译)
 1. 中间代码生成
 2. 代码优化
 3. 目标代码生成

高级程序处理过程:初始源程序→预处理→源程序→编译→目标汇编→机器代码

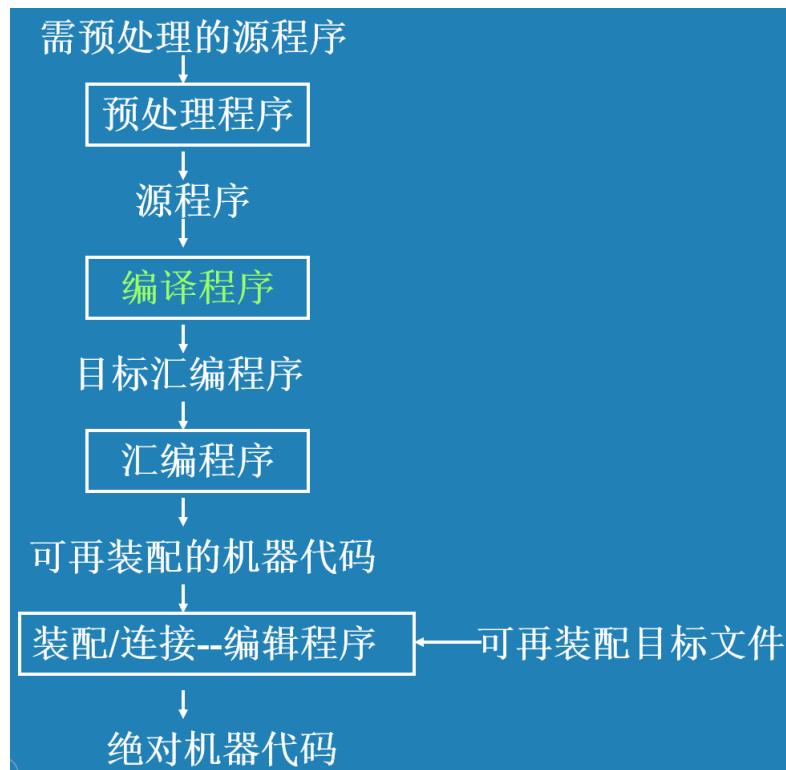


图 2: 高级语言程序的处理过程

注:生成机器代码的时候,并不是直接生成,而是先生成对应的汇编代码,再生成机器代码。

编译过程:每个阶段的输出作为下一个阶段的输入(即数据从一种形式转换成另一种形式)。

编译过程的每个阶段都包括两个相同的处理:表格管理和出错处理。

表格管理是保存编译过程每个阶段的数据和结果,出错处理则是对编译过程遇到的语法,词法等错误进行处理。

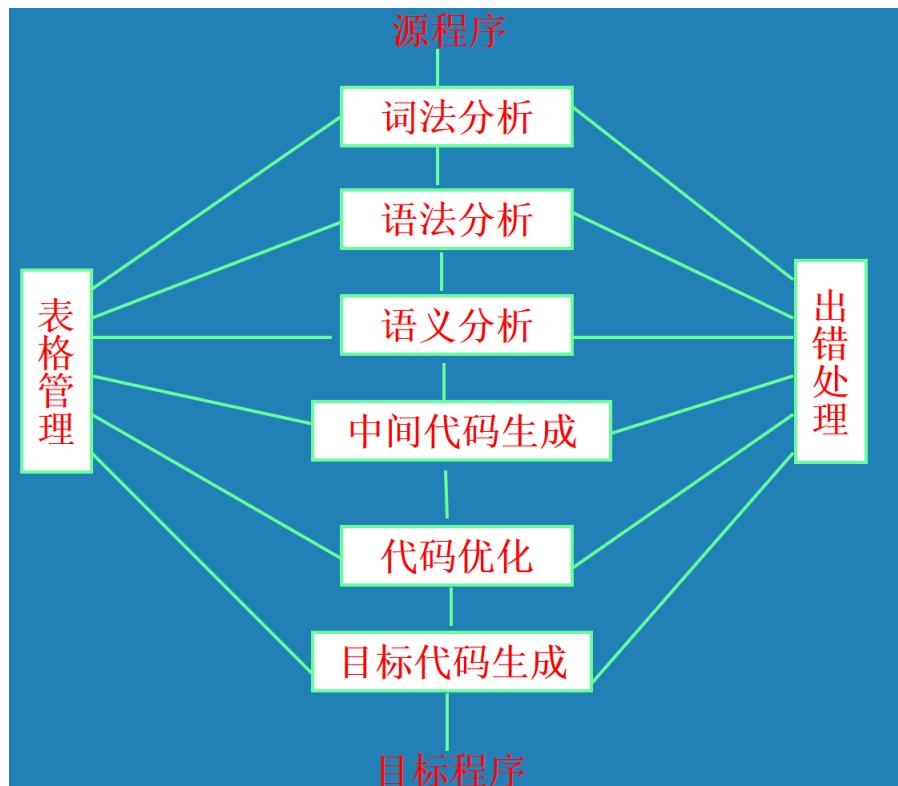


图 3: 编译的各个阶段

1.1 词法分析

词法分析的任务:从左到右一个字符一个字符地读入源程序,对构成源程序的字符流进行扫描和分解,从而识别出一个个单词(**token**)

主要分为两大步骤,扫描和分解

扫描为从左到右扫描

分解为以界符对语句进行分割(注:双引号内的介符不可划分)

最后结果使用一个二元组保存,格式为 (**种类,值**)

1.2 语义分析

语义分析:不断构造语法树

如:类型不对,数组越界等

过程:单词符号串→语法分析→语法短语

识别规则:描述程序结果的规则,通常由递归规则表示。

输出:合法的语法树

1.3 语义分析

任务:审查源程序是否有语义错误,为代码生成阶段收集类型信息

语义分析:包括静态语义和动态语义

常见的错误包括类型不匹配,数组越界等

输出:生成中间代码(把中间代码生成合并到语义分析部分,拆分开也可以)

结果使用四元式表示,格式为:(运算符,运算对象1,运算对象2,结果)

静态语义:

- 上下文相关性
- 类型匹配:每个算符是否具有语言规范允许的运算
- 类型转换

1.4 中间代码生成 intermediate code generation

任务:将源程序生成一种内部表示形式,这种内部表示形式叫 **中间代码**

一般使用四元式表示,格式为:(运算符,运算对象1,运算对象2,结果)

1.5 代码优化

任务:对中间代码进行等价变换,以便生成更高效的目标代码。

1.6 目标代码生成

任务:把中间代码转换成特定机器上的绝对指令代码或可重定位的指令代码或汇编指令代码,它的工作与硬件系统和指令含义有关

目标代码生成:与目标机器紧密相关,先生成对应的汇编代码,再生成机器代码

1.7 表格管理与出错处理

表格管理与出错处理:每个阶段都执行这一操作

表格管理:编译过程中源程序的各种信息被保存在种种不同的表格里,编译各阶段的工作涉及到构造、查找或更新有关的表格,因此需要有表格管理工作。

出错处理:编译过程中,发现源程序有错误(词法错误,语法错误,语义错误),编译程序应报告错误的性质和错误的地点,并将错误所造成的影响限制在尽可能小的范围内,使得源程序的其余部分继续被编译下去。这些工作称为出错处理(**error handling**)。

1.8 编译程序划分

编译程序划分:分析与综合(翻译)两个阶段

按是否与目标机器相关:前端(无关)和后端(相关)

2 文法和语言

掌握自下而上和自上而下的分析方法。

程序设计的定义:语言是一个记号系统。

分析程序设计语言的两个阶段: $\left\{ \begin{array}{l} \text{每个程序的构成规律} \\ \text{每个程序的含义} \end{array} \right.$

2.1 语法

语法:是一组规定,用它可以形成和产生一个合适的程序。

描述工具:文法

语法只可以判断结构是否合法。

2.2 语义

语义:
 | 静态语义
 | 动态语义

静态语义:一系列限定的规则,确定哪些合乎语法的程序是合适的。

动态语义:运行或动态链接时进行的判断。

描述工具:指称语义,操作语义

作用:检查类型匹配,变量作用域等。

静态语义:

- 上下文相关性
- 类型匹配:每个算符是否具有语言规范允许的运算
- 类型转换

2.3 文法

如何描述语句?

1. 生成方式(文法)

语言中每个句子可以用严格定义的规则进行构造

2. 识别方式(自动机)

用一个过程,经有限次计算后会停止回答“是”,若属于句子,要么回答“否”,要么永远持续下去

语言:可以是有穷的也可以是无穷的,我们要做的是,找出语言的有穷表示。

文法的作用:

1. 使用有穷的规则描述无穷的语言
2. 严格定义句子的结构,是判断句子结构是否合法的依据

方法:

::=表示定义一条规则

⇒表示应用这条规则,完成句子的变换(即由...推导出...的意思)

2.3.1 符号和符号串

字符表(符号集,字母表):由字母、数字和若干专用字符组成的非空有限集合。

符号串:字母表中的符号组成的任何有穷序列。

比如: a, ac, aca 是 $A : \{a, b, c\}$ 的符号串。

空串:不含任何符号的符号串 , 用 ϵ 表示

符号串的长度:符号串含有符号的个数, $|\epsilon| = 0$

符号串的运用:

- 连接

定义: $x = "ST"$, $y = "aby"$

则 $xy = "STaby"$

- 方幂

$$a^n = \underbrace{aa \cdots aa}_{n \text{ 个 } a}$$

注: $a^0 = \epsilon$

- 集合的乘积

定义: $A = \{a, b\}$, $B = \{0, 1\}$

则 $AB = \{a0, a1, b0, b1\}$

注:A,B是符号串的集合,并且AB的结果中A的符号串在前面

- 集合的方幂

$$A^2 = AA$$

2.3.2 闭包

闭包: Σ 的闭包为 Σ 上所有元素组合的集合 Σ^* 。即

$$\Sigma^* = \Sigma^0 \bigcup \Sigma^1 \bigcup \Sigma^2 \dots \quad (1)$$

正闭包:闭包去掉空集元素。即

$$\Sigma^+ = \Sigma^* - \{\epsilon\} \quad (2)$$

有:

$$\Sigma^+ = \Sigma \Sigma^* = \Sigma^* \Sigma \quad (3)$$

字母表上的一个语言是 Σ 上的一些符号串的集合,即该语言是 Σ^* 的一个子集

注: $\{\epsilon\}$ 也是一个语言 ,即 Φ (空集)是一个语言

2.3.3 文法

产生式(规则):一组有序对 (α, β) 表示 $\alpha \rightarrow \beta$ 或 $\alpha ::= \beta$

文法的定义:四元组 (V_N, V_T, P, S)

其中, V_N 表示非终结符 , V_T 表示终结符 , P 表示产生式集合, S 表示文档起始符号。

注:

- S 为文档的其实符号,从具体情况上看,我们就是从 S 起始的产生式对句子进行解析。

- $V_N \bigcup V_T =$ 字母表

- S 是一个非终结符,并且至少在一条产生式的左部出现

例子:

文法 $G = (V_N, V_T, P, S)$

$V_N = \{S\}$, $P = \{S \rightarrow 0S1, S \rightarrow 01\}$, $V_T = \{0, 1\}$

开始符号为 S

文法 $G = (V_N, V_T, P, S)$

$V_N = \{\text{标识符}, \text{字母}, \text{数字}\}$

$V_T = \{a, b, \dots, z, 0, 1, \dots, 9\}$

$$P = \left\{ \begin{array}{l} <\text{标识符}> \rightarrow <\text{字母}>, \\ <\text{标识符}> \rightarrow <\text{标识符}><\text{字母}>, \\ <\text{标识符}> \rightarrow <\text{标识符}><\text{数字}>, \\ <\text{字母}> \rightarrow a, \\ \dots, \\ z, <\text{字母}> \\ <\text{数字}> \rightarrow 0, \\ \dots, \\ 9, <\text{数字}> \end{array} \right\}$$

$S = <\text{标识符}>$

2.3.4 简化表示

简化表示:只使用产生式来表示文法,四元组的其他三元在产生式中表示。

- 第一条产生式的左部表示 S
- 用大写字母或者尖括号包围表示非终结符集合
- 用小写字母表示终结字符集合
- 左部相同的产生式的多个,可以用 |(或) 来简化表示。
比如: $A \rightarrow \alpha, A \rightarrow \beta$ 可以简记为 $A \rightarrow \alpha|\beta$

注意: $A \rightarrow \alpha|\beta$ 表示的是两条产生式而不是一条,其中 α, β 为候选式。

2.4 归纳与推导

直接归纳与直接推导:

若 $v \Rightarrow w$ 则称 v 直接推导 w 或 w 直接归纳为 v

归纳与推导: 使用了多条产生式, 记为

$$S \stackrel{+}{\Rightarrow} \alpha \tag{4}$$

若 $S \Rightarrow^+ \alpha$ 或 $S = \alpha$ 则记作:

$$S \stackrel{*}{\Rightarrow} \alpha \tag{5}$$

注: 如果使用了一条产生式, 进行了多次 \Rightarrow 也为直接推导, 推导是使用了多条产生式。推导指的是多次直接推导, 无论是否使用同一条产生式

直接推导: 就是用产生式的右部替换产生式的左部的过程

直接归约: 就是用产生式的左部替换产生式的右部的过程

2.5 句型、句子、语言

句型与句子的定义:

对于文法 $G[S]$,若 $S \xrightarrow{*} x$,则称 x 为文法 G 的句型。

若 x 仅由终结符组成,则称 x 为文法 G 的句子

注:

- 句型可以表示多个句子,句子是句型的一个情况之一。句子也是一个句型。
- 起始符也为句型,即
隐含条件: $S \rightarrow S$,所以 S 也是文法 G 的句型。

语言 $L[G]$ 的定义:语言 $L[G]$ 是文法 $G[S]$ 的所有句子的集合。

注:

- $+, (*,)$ 等出现在句子中的成分,也是终结符。
- 可以通过产生式的具体表达,判断出某个操作的优先级。

文法的等价:

若 $L[G_1] == L[G_2]$,则称文法 G_1 和文法 G_2 等价。

文法的种类:

- 0型文法(短语文法)

对任一产生式 $\alpha \rightarrow \beta$,都有 $\alpha \in (V_N \cup V_T)$,且至少含有一个终结符, $\beta \in (V_N \cup V_T)$

即产生式左侧只要不是句子,就是0型文法

对产生式基本无限制

- 1型文法(上下文相关文法)

在0型文法条件下,对任一产生式 $\alpha \rightarrow \beta$,都有 $|\beta| \leq |\alpha|$,仅仅 $S \rightarrow \epsilon$ 除外(| α |指的是句型的长度)

即产生式左侧短于右侧,就是1型文法

上下文体现在是对产生式左部的扩展,拆分成多个部分,当扩展其中一个部分的时候,就需要考虑其他部分(上下文)

$\alpha A \gamma \rightarrow \alpha \beta \gamma, \gamma \in V^*, A \in V_N, \beta \in V^+$,将 A 替换成 β 时,必须考虑 A 的上下文 α, γ

- 2型文法(上下文无关文法)

在1型文法条件下,对任一产生式 $\alpha \rightarrow \beta$,都有 $\alpha \in (V_N), \beta \in (V_N \cup V_T)$

即产生式左侧只有非终结符

大部分程序设计语言是2型文法

- 3型文法(正规文法)

在2型文法的条件下,,对任一产生式 $\alpha \rightarrow \beta$,都有形如 $A \rightarrow aB$ 或 $A \rightarrow a$,其中 $A, B \in V_N, a \in V_T$

即产生式的右侧必须是以终结符开头

一般用来定义一个单词

4种类型的文法约束依次为:左侧不能是句子,左侧长度小于右侧,左侧只有非终结符,右侧以终结符开头

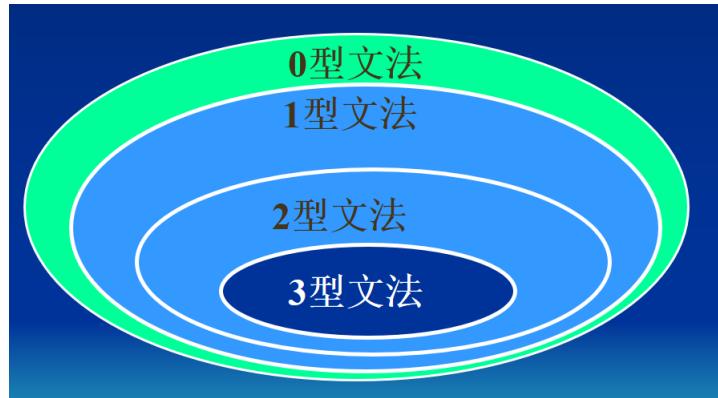


图 4: 四种文法之间的包含关系

3 上下文无关文法及其语法树

上下文无关文法:2型文法,有足够的能力描述现今程序设计语言的语法结构

例子:算数表达式: $E \rightarrow i | E + E | E * E | (E)$

$<\text{赋值语句}> \rightarrow i := E$

$<\text{条件语句}> \rightarrow if <\text{条件}> then <\text{语句}>$

$| if <\text{条件}> then <\text{语句}> else <\text{语句}>$

3.1 规范推导和规范句型

最左/右推导:

在推导任何一步 $\alpha \rightarrow \beta$ 时,其中 α, β 为句型,都是对 α 中的最左/右的非终结符进行替换,则成为最左/右推导。

推导到底是多次直接推导,还是多条产生式?前者

规范推导:最右推导

规范规约:最左规约

规范句型:由规范推导所得的句型

语法树:推导的过程,可以展开成语法树,但是从语法树是看不出语法树的构造顺序的(即推导的过程)

在语法树上,从左到右读取叶子节点,可以得到句型或句子。

语法树的定义:满足下面几个条件的树,则称为文法G的语法树。

- 每个节点都有标记,是终结符或者非终结符中一个符号
- 根节点的标记是起始符(S)
- 若一个节点至少有一个除它自己之外的子孙,则这个节点的标记一定是非终结符
- 语法树得到的一定是文法G的产生式

文法的二义性:

一个文法存在某个句子对应2个不同的语法树,则称文法有二义性。

对于程序设计语言来说,希望它的文法是无二义性的,因为希望对它的每个语句的分析是唯一的。因此,如果出现了有二义性的文法,可以尝试把它转换成等价的无二义性的文法。

3.2 句型的分析

句型的分析:识别给定串是否是某文法的句型。

句型的分析主要有两种方法:

- 自上而下的分析法(推导)
- 自下而上的分析法(归约)

自上而下的算法:从开始符出发,反复使用文法中的产生式,寻找匹配的推导。(从根节点开始不断构造语法树,找出与给定串匹配的语法树)

这个算法的关键之处是**如何选择产生式?**(剧透:LL(1)文法只有一个待选择的产生式,多个选择的时候可以使用回溯法(即多个可能都进行搜索尝试))

注:这里生成语法树的是为后面的语法分析和词法分析所用,对于句型分析(词法分析)是没有作用的。

这里的描述的意思是:如果能正确构造出这个串的语法树那么它就是合法的句子(而构造的过程则是不断推导的过程,通过推导来确定语法树的创建过程,而不是由语法树来决定推导的过程)

语法树的末端结点符号串正好是输入符号串。

自下而上的算法:从给定串出发,归约到起始符(从叶子节点出发,向上构造语法树)

这个算法的关键之处是**如何确定可归约串?**(剧透:通过构造出语法各个状态的转移过程来确定归约串,当输入归约串的同时,状态也不断发生变化,而下一个动作(状态转移还是归约)则由当前的输入和当前的状态决定,问题转变成**如何构造这个分析表?**)

自上而下的话,感觉就是凭空想象,直到推出想要的结果,而自下而上的话,则是从句子本身出发,带有明确的方向性,更为简单。胡说八道,虽然自上而下只知道回溯法

归约的描述: $abc \rightarrow S$,表示串 abc 被归约成 A

这里的语法树也是和上面一样,是随着归约的过程构造,而不是由语法树来进行归约。

要确定可归约串,首先要了解短语,直接短语,句子的概念。**为什么?回头需要去看算法的原理,辅助分析建立的概念**

短语:若 $S \xrightarrow{*} \alpha A \delta \xrightarrow{*} \beta$,则称 β 是句型 $\alpha \beta \delta$ 相对于 A 的短语。(从语法树上来看,短语就是语法树的每棵子树的叶子组成的句子)

到底是句型 $\alpha A \delta$ 还是句型 $\alpha \beta \delta$?后者

直接短语:若 $S \xrightarrow{*} \alpha A \delta \Rightarrow \beta$,则称 β 是句型 $\alpha \beta \delta$ 相对于 A 的直接短语。(从语法树上来看,每个叶子都是直接短语)

句柄:一个句型的最左直接短语为句柄(从语法树上来看,最左的叶子就是句柄)

可归约串:句柄即可归约串。

那么自下而上的算法则是,每次选取语法树的句柄归约。

这里为方便理解,从语法树的角度来短语,直接短语,句柄这些概念,但实际并不是从语法树来确定可归约串,而是从句柄在语法树的位置,我们能更加清晰的明白为什么在这个句柄的情况下,为什么要使用这个句型进行归约。

后面建立分析表的时候,实际上就是枚举出了所有句型的语法树(状态转移图),从图上得到是归约还是继续状态转移(未到语法树的叶子),从而确定每个句型的句柄。当多个句型有相同的句柄的时候,就出现了冲突,就需要结合其他技术来解决问题(LR(1), SLR(0), LALR文法等)

后面补上一个求直接短语,短语,句柄以及归约的例子

多余规则:

- 不可到达
- 不可终止

4 词法分析

单词描述的工具:正规文法(3型文法)和正则式(正则表达式)

4.1 正规文法与正则式

正规文法也称三星文法 $G = (V_N, V_T, S, P)$,其P中的一条规则都有以下形式: $A \rightarrow aB$ 或 $A \rightarrow a$,其中 $A, B \in V_N, a \in V_T^*$ 。

即(一句话概括)文法的产生式的右侧以终结符开头。

正则式:也称正则表达式。

既然有了正规文法,为什么还需要正则式?(作用)

因为正则式可以直观地看出单词的构成

给定集合 Σ

- ϵ 和 Φ^1 都是某个集合 Σ 上的正规式。
- 集合内的任意元素都是集合 Σ 的正规式
- 正规式经过以下运算之后,还是集合 Σ 的正规式
 - “|” 或
 - “.” 连接
 - “*” 闭包
 - 优先级 $* > . > |$

例子:令 $\Sigma = \{a, b\}$, Σ 上的正规式和相应的正规集的例子如下:

¹ $\Phi = \{\epsilon\}$

正规式	正规集
a	$\{a\}$
$a b$	$\{a, b\}$
ab	$\{ab\}$
$(a b)(a b)$	$\{aa, ab, ba, bb\}$
a^*	$\{\epsilon, a, aa, \dots\}$, 即任意个a的串
$(a b)^*$	$\{\epsilon, a, b, aa, ab, \dots\}$, 即所有a,b组成的串

4.2 正规文法和正则式的等价性

将正规式转换成正规文法:

选择一个非终结符 S 生成类似产生式的形式: $S \rightarrow r$, 并将 S 定为 G 的识别符号。

具体转换规则如下图, 原产生式右边为正规式

	原产生式	变换后产生式
规则1	$A \rightarrow xy$	$A \rightarrow xB \quad B \rightarrow y$
规则2	$A \rightarrow x^*y$	$A \rightarrow xA \quad A \rightarrow y$
规则3	$A \rightarrow x y$	$A \rightarrow x \quad A \rightarrow y$

图 5: 正规式转换成正规文法

例子: 将 $r = a(a|d)^*$ 转换成相应的正规文法。书本上貌似写多了, 以下为自己观点, PPT是下面的写法

- $S \rightarrow a(a|d)^*$
- $S \rightarrow aA$
 $A \rightarrow (a|d)^*$
- $S \rightarrow aA$
 $A \rightarrow (a|d)A$
 $A \rightarrow \epsilon$
- $S \rightarrow aA$
 $A \rightarrow aA$
 $A \rightarrow dA$
 $A \rightarrow \epsilon$

将正规文法转换成正规式: emmm, 这个转换看直觉吧, 懒得写了
为什么最右推导, 最左规约是合理的? 有什么直观的理解?

4.3 有穷自动机

有穷自动机:也称有限自动机,是一种识别装置,能准确识别正规集,即识别正规文法所定义的语言和正规式所表示的集合。

有穷自动机本质上是一个状态转移图。

4.3.1 确定的有穷自动机(DFA)

定义:一个确定的有穷自动机 M 是一个五元组

$$M = (K, \Sigma, f, s, z)$$

其中,

- (1) K 是一个有穷集,它的每个元素称为一个状态
- (2) Σ 是一个有穷字母表,它的每个元素称为一个输入符号,所以也称为 Σ 为输入符号表。
- (3) f 是转换函数, 是 $K \times \Sigma \rightarrow K$ 上的映像。
- (4) $S \in K$,是唯一的一个初态
- (5) $Z \subseteq K$,是一个终态集,终态也称为可接受状态或结束状态。

确定性体现在:

- 初态是唯一的
- 每个状态对应的唯一的下一个状态

DFA可以使用状态图或状态矩阵表示:图6和图7分别表示状态图和状态矩阵。

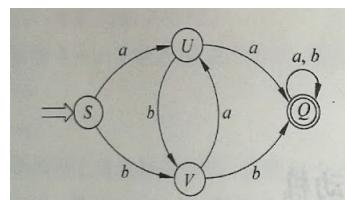


图 6: 状态转移图:初态结点冠以” \rightarrow ”或标以”-”,终态结点用双圈表示或标以”+”

输入

状态 \ 符号	a	b	
s	U	V	0
U	Q	V	0
V	U	Q	0
Q	Q	Q	1

图 7: 状态转移矩阵:可以用 \rightarrow 标明初态;否则第一行即初态,相应终态行在表的右端标以1,非状态标以0

4.3.2 不确定的有穷自动机(NFA)

定义:一个不确定的有穷自动机 M 是一个五元组

$$M = (K, \Sigma, f, s, z)$$

其中,

- (1) K 是一个有穷集,它的每个元素称为一个状态
- (2) Σ 是一个有穷字母表,它的每个元素称为一个输入符号,所以也称为 Σ 为输入符号表。
- (3) f 是转换函数, 是 $K \times \Sigma^*$ 到 K 的全体子集的映像,即 $K \times \Sigma^* \rightarrow 2^K$,其中 2^K 表示 K 的幂集^{2 3} 这一段是什么鬼?见脚注
- (4) $S \subseteq K$,是一个非空初态集
- (5) $Z \subseteq K$,是一个终态集

确定性体现在:

- 初态不唯一(有初态集)
- 每个状态对应的同一个输入,有多个可能的状态转移。(即映射不唯一,一个状态映射到所有状态的某个子集)

4.3.3 NFA转换为等价的DFA

定理:设 L 为一个由NFA接受的集合,则存在一个接受 L 的DFA。

子集法:将NFA转换成接受同样语言的DFA

为一个NFA构造相应的DFA的基本想法是让DFA的每一个状态对应NFA的一组状态。(即把可能转移的多个不确定的状态的组合表示成一个状态来看) 具体算法见8在进行子集法的时候,也会有无用状态去掉

- ϵ 合并
经过 ϵ 弧得到的状态可以合并成一个状态
- 状态合并
当前状态所能直接转移到的所有状态合并成一个状态
- 状态集合I的 ϵ 闭包
 $\epsilon - closure(I)$ 状态集I中的任何状态S经过任一条 ϵ 弧而能到达的状态
- 状态集合I的 a 弧转换
 $move(I, a)$,所有那些可从I中的某个状态经过一条a弧而达到的状态的全体。

² $K \times \Sigma^*$ 指的是两者的笛卡尔积,表示一个状态与每个输入的组合,然后这里取闭包 Σ^* ,表示每个状态与多个可能输入的组合(也可以分解成一个状态一个输入,这样写应该是为了简化,因为画图的时候,采取前者而不是逐个画)

³所谓幂集(Power Set),就是原集合中所有的子集(包括全集和空集)构成的集族

```

① 开始,令  $\epsilon$ -closure( $K_0$ )为 C 中唯一成员,并且它是未被标记的。
② While(C 中存在尚未被标记的子集 T) do
{标记 T;
  for 每个输入字母 a do
    {U :=  $\epsilon$ -closure (Move(T,a));
     if U 不在 C 中 then
       将 U 作为未被标记的子集加在 C 中
    }
}

```

图 8: 子集构造算法

- C 为 DFA 的状态集合,初始元素为起始状态的空闭包
- while 所做的就是不断以 C 中的状态转移出新的状态加入 C 中,直到所有的装备都被使用过
- 所谓的转移,指的就是 $\epsilon - closure(Move(T, a))$
其中, T 为 C 中的一个状态(它的值为 NFA 中某些状态的集合), a 表示经过的弧, 结果可以简记为 I_a
- 终态的确定
上面的算法确定的 NFA 对应到 DFA 有哪些状态,但是还未确定终态。
DFA 的终态就是 DFA 中和 NFA 中的终态有交集的状态。

4.3.4 DFA 的化简

一个 DFA 可以通过消除无用状态和合并等价状态来转换成一个等价的最小状态的 DFA。
也就是化简主要分两个部分。

所谓最简的 DFA 指的是它没有多余的状态,并且它的状态中没有两个是互相等价的。

无用状态:

- 从该自动机的开始状态出发,任何输入也不能到达的那个状态
- 从这个状态没有通路到达终态

有什么算法能解决上面这个问题呢? 从起点开始 DFS, 在能到达终点的前提下, 所能访问到的所有点都是有效的。其他都是无效的。注: 在进行子集法的时候, 也会有无用状态去掉

等价状态:

- (1) 一致性条件
状态 s 和 t 的必须同时为可接受状态或不可接受状态。⁴
- (2) 蔓延性
对于所有输入符号, 状态 s 和 t 必须转换到等价的状态里。

⁴ 可接受状态指的是终态, 不可接受状态指的是非终态。

那么如何找出等价的状态进而化简呢?

分割法:把一个DFA(不含多余状态)的状态分成一些不相交的子集,使得任何不同的两个子集的状态都是可区别的,而同一子集中的任何两个状态都是等价的。

- 终态与非终态能初始划分出初始的两个不等价的子集。
- 对于每个集合,尝试所有的输入,如果集合内的两个状态不能转移到相同集合,那么这两个状态是不等价的,需要把这两个状态拆分到两个新的子集中。
- 尝试所有的输入可能,最终得到的集合内的元素就是相互等价的元素
(因为尝试的所有输入都是转移到相同的状态(可蔓延性),而一致性在初始的时候已经明确)。

例子:把图9表示的NFA确定化和最小化。

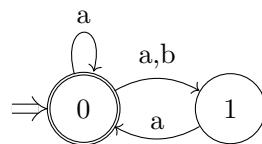


图 9: 例子:子集法、分割法例子

- 确定化,见表1

set	input a	input b
$A = \{0\}$	$\{0, 1\}(B)$	$\{1\}(C)$
$B = \{0, 1\}$	$\{0, 1\}(B)$	$\{1\}(C)$
$C = \{1\}$	$\{1\}(C)$	Φ

表 1: 例子:子集法表

对应状态图为:见图10

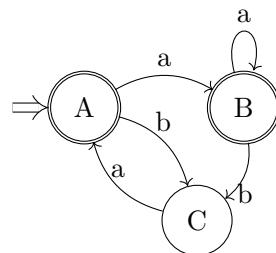


图 10: 例子:子集法图

- 最小化,见表2

input	sets
init	$\{C\}, \{A, B\}$
a	$\{C\}, \{A, B\}$
b	$\{C\}, \{A, B\}$

表 2: 例子:分割法表

因此 A, B 两个状态是等价的。对应的状态图为 11

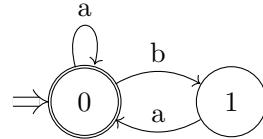
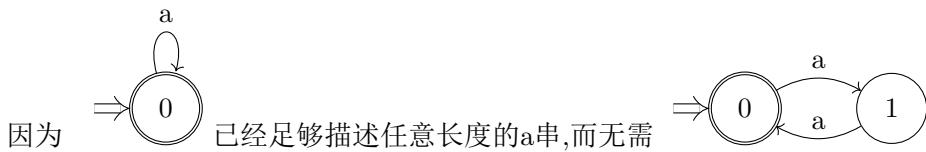


图 11: 例子: 分割法图

另外,可以观察出,NFA的图9和最小化后的DFA的图11 只是相差的一个 a 而已,实际上他们的确是等价的。



4.3.5 正规式与有穷自动机的等价性

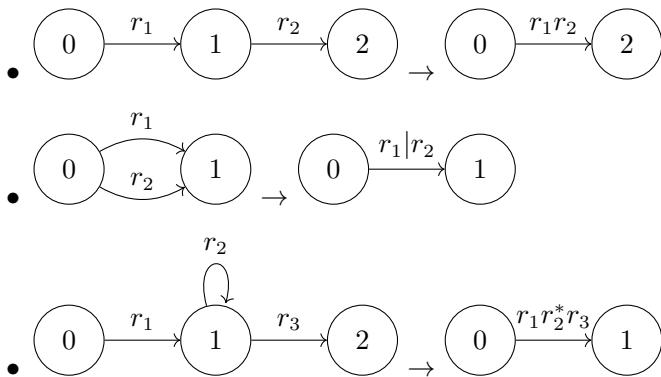
正规式和有穷自动机的等价性由以下两点说明:

- 对于字母表(输入表) Σ 上的NFA M ,可以构造一个 Σ 上的正规式 r ,使得 $L(r) = L(M)$
- 对于 Σ 上的每个正规式 r ,可以构造一个 Σ 上的NFA M ,使得 $L(r) = L(M)$

正规式 \rightarrow NFA:分两个步骤

- 建立 x 节点,使用 ϵ 连接到初态,建立 y 节点,使用 ϵ 弧连接终态
- 不断消去弧,直到只剩下 x, y 两个节点

消去规则:



NFA \rightarrow 正则式:

- 环对应闭包

注意,闭包是包括空的

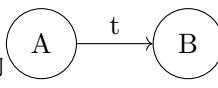
- 连续输入对应与

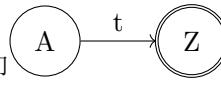
- 分支对应或

4.3.6 正规文法和有穷自动机的等价性

主要有以下几点性质:

- 弧的输入(字母表)就是终结符
- 非终结符就是状态
- 新增一个状态Z作为终态

- 对于形如 $A \rightarrow tB$ 的产生式,得到 $f(A, t) = B$ 的转换函数。即 

- 对于形如 $A \rightarrow t$ 的产生式,得到 $f(A, t) = Z$ 的转换函数,即 

正则式,正规文法,有穷自动机的关系?

正则式和正规文法是描述工具,有穷自动机是识别工具,三者可等价相互转换。

4.4 词法分析程序的构造

词法分析程序的构造主要分为4个步骤:

- 接口类型确定

语法分析程序可以有3种利用方式:

一是作为单独的子程序,其结果作为语法分析程序的输入。

二是作为语法分析程序的一部分

- 确定单词的分类和Token串结构

单词常见的类型有:关键字,标识符等

token串一般为2元组(单词的种别,单词自身的值) (需要自己对种类进行编号)

- 特殊问题

- 标识符和保留字的区分

事先构造好保留字表,在拼出标识符的单词下能查询保留字表,如果是保留字,则作为保留字处理,否则才是标识符

- 空格、制表符及换行符的处理

无用空格和制表符要删除

字符串内的空格不能删

换行符不能删,对于错误处理起作用

- 复合型Token的处理

比如”!=”,读取到”!”的时候,还必须读取下一个字符进行判断

- 括号,引号等成对符号的匹配问题

这一个本应是语法分析的工作,但是可以词法分析程序中完成

处理方式是,设置计数器,每遇到左括号计数器+1,右括号计数器-1,最终结果不为0,说明括号不匹配

- 使用状态转换图构造语法分析程序

文法or正规式 \rightarrow NFA \rightarrow DFA \rightarrow 最简DFA

后面补上正规式,正规文法与有穷自动机相互转换的例子

综合例题: 为正规文法 $G[S]$

$$S \rightarrow aA|bQ$$

$$A \rightarrow aA|bB|b$$

$$B \rightarrow bD|aQ$$

$$Q \rightarrow aQ|bD|b$$

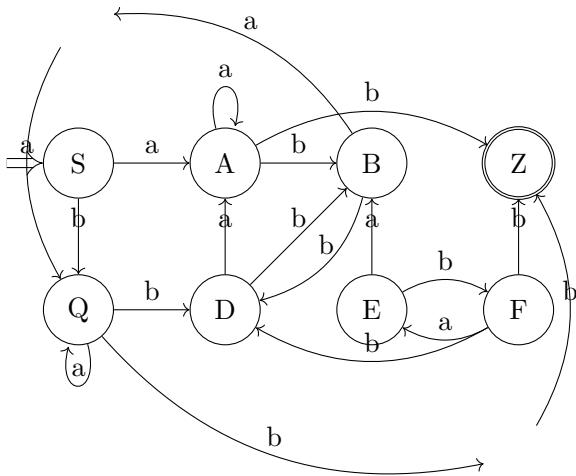
$$D \rightarrow bB|aA$$

$$E \rightarrow aB|bF$$

$$F \rightarrow bD|aE|b$$

构造出相应的最小的DFA。

- 正规文法→NFA



5 自顶向下的语法分析

5.1 确定的自顶向下的语法分析

确定分析的条件:从文法的开始符号出发,如果能根据当前的输入符号(单词符号)唯一地确定选用那个产生式进行推导,则分析是确定的。

确定的分析有以下3种可能情况:

- $S \rightarrow pA|bA$

产生式的右部由终结符开头,同一个非终结符的不同的产生式由不同的终结符开头,我们可以直接地,唯一确定选用那个产生式

- $S \rightarrow Ap|Bq$

产生式的右部以(非空)终结符开头,同一个非终结符的右部由不同的符号开头

这种情况,我们无法确定选用哪个产生式,关键是 判断产生式右部推出的开始符号集 ,若A和B的开始符号集不相交分析过程是确定的

- $A \rightarrow \epsilon$

对于空产生式左部的非终结符,关键是 判断该非终结符的后跟符号集,分析过程可能是确定的。

要进行确定的自顶向下分析,文法要满足是 $LL(1)$ 文法

开始符号集:

设 $G = (V_T, V_N, P, S)$ 是上下文无关文法。(产生式左侧只有非终结符)

$$FIRST(\alpha) = \{a | \alpha \xrightarrow{*} a\beta, a \in V_T, \alpha, \beta \in V^*\}$$

则称 $FIRST(\alpha)$ 为 α 的开始符号集 或 首符号集。(可以包含空弧 ϵ)

也就是求产生式的一个可能的终结符

后跟符号:

设 $G = (V_T, V_N, P, S)$ 是上下文无关文法, $A \in V_N, S$ 是开始符号

$$FOLLOW(A) = \{a | S \xrightarrow{*} \mu A \beta \text{ 且 } a \in FIRST(\beta), \mu \in V_T^*, \beta \in V^+\} \text{ (特别的, # 表示输入串的结束符)}$$

也就是求产生式第二个可能的终结符,但是只有在有空弧的情况下才考虑即($\epsilon \in FIRST(A)$)

划重点:# 不要忘了,初始状态:# $\in FOLLOW(S)$

选择符号集:

一个产生式的选择符号集合SELECT。给定上下文无关文法的产生式

$$A \rightarrow \alpha, A \in V_N, \alpha \in V^*$$

若 $\alpha \xrightarrow{*} \epsilon$,

则 $SELECT(A \rightarrow \alpha) = FIRST(\alpha)$

若 $\alpha \xrightarrow{*} \epsilon$,

则 $SELECT(A \rightarrow \alpha) = (FIRST(A) - \{\epsilon\}) \cup FOLLOW(A)$

也就是可选的非终结符,在没空弧的情况下就是 $FIRST$,在有空弧的情况下还要考虑 $FOLLOW$ (同时去掉空弧)

LL(1)文法的充要条件

- 上下文无关文法
- 对于每个非终结符 A 的两个产生式 $A \rightarrow \alpha, A \rightarrow \beta$,满足

$$SELECT(A \rightarrow \alpha) \cap SELECT(A \rightarrow \beta) = \emptyset$$

其中, α, β 不同时能 $\xrightarrow{*} \epsilon$

也就是同一个非终结符的产生式的第一个可能终结符不能相同

LL(1)文法的判别: 一个上下文无关文法是**LL(1)**文法的充要条件 \rightarrow SELECT集 \rightarrow FIRST+FOLLOW \rightarrow ϵ
1的含义:1表明只需要向前看一个符号便可以决定选哪个产生式进行推导,类似的LL(k)文法需要向前看 k 个符号才可以确定选用哪个产生式

- 求能推导出 ϵ 的非终结符
 - 标记每个非终极符的状态
 - 使用直接推导能推导出空的,标记是
 - 循环推导,直到标记表不再发生变换

有些不确定能推导出空的非终结符后来确定可以的时候,它所在的产生式也可能推导出空,因此循环处理
- 求 $FIRST$ $a \rightarrow X_1 X_2 \cdots X_n$

- X_1 不为空, $FIRST(a) = FIRST(X_1)$
- 对于前j个可达空, $FIRST(a) = FIRST(A) + FIRST(X_J) - \{\epsilon\}$
- 循环,直到FIRST不变
- 求FOLLOW

$$A = aBC$$
 - C不能达到空,FOLLOW(B) = FOLLOW(C)
 - 可以达空,FOLLOW(B) = FOLLOW(A) + FIRST(C)
 - 循环直到FOLOW不再增加
 - 初始 FOLLOW(S) = #
- 求SELECT
根据定义进行即可

后面补上一个例子

5.1.1 非LL(1)文法转LL(1)文法

不是LL(1)文法:

- 消除左递归
- 提取左公因子

提取左公因子:

对于含有 $A \rightarrow ab|ac$ 的文法,显然不是LL(1)文法

我们可以提取左公因子转换成LL(1)文法: $A \rightarrow aB, B \rightarrow b|c$

直接左递归: $A \rightarrow Ab$,含有左递归的产生式,称为直接左递归

间接左递归: $A \rightarrow Ba, B \rightarrow Ab$,即 $A \Rightarrow^+ A$,称为间接左递归

以直接左递归为例子: $A \rightarrow Aa, A \rightarrow b$

有: $Select(A \rightarrow Aa) \supseteq First(Aa) \supseteq First(b)$

$Select(A \rightarrow b) \supseteq First(b)$ 因此,含有左递归的文法都不是LL(1)文法

消除直接左递归:

$S \rightarrow Sa|b$

改写成

$S \rightarrow bS', S' \rightarrow aS'|\epsilon$

消除间接左递归:通过代入等方法把间接左递归变成直接左递归,对于间接左递归下,还嵌套有直接左递归,那么先消除直接左递归,再代入

5.2 不确定的自顶向下分析

对于非LL(1)文法,就不能确定选择哪个产生式规则,对于待选的产生式,采用回溯搜索,逐条试探的方式进行

其实就是回溯算法啦

含有空弧的情况有什么特殊处理的地方吗?

5.3 确定的自顶向下算法-实现

- 递归子程序算法
- 预测分析法

5.3.1 递归子程序法

实现思想:对文法中的每个非终结符编写一个递归过程,识别由该终结符推出的串。当非终结符含有许多条产生式时,按当前输入属于哪条产生式的SELECT集可唯一确定(LL(1)文法)哪个产生式进行匹配。

- 当识别到终结符时与输入符号匹配并读取下一输入符。
- 当识别到非终结符时,则调用该终结符相应的过程

特点:

- 优点:简单直观,易于构造
- 缺点
 - 对文法要求高,必须满足LL(1)文法
 - 递归调用多
- 实用性:许多高级语言,如Pascal、c等编译系统常常采用此方法

5.3.2 预测分析法

一个预测分析器由3个部分组成:

- 预测分析程序
控制分析过程的进行
- 分析栈
存放从文法开始符号出发的自顶向下的推导过程中等待匹配的文法符号。
开始时放入#和文法开始符号
结束时栈应该是空的。
- 预测分析表
是一张二维表,元素 $M[A, a]$ 的内容是当前非终结符面临输入符号a时应该选取的产生式
当无产生式时,元素内容为转向出错处理

构造预测分析表:

- 把文法转变为LL(1)文法
- 求出每条产生式的SELECT集
- 依照SELECT集把产生式填入分析表
- 用a表示终结符或#,若 $a \in SELECT(A \rightarrow b)$,则把 $A \rightarrow b$ 放入 $M[A, a]$ 中,把所有无定义的 $M[A, a]$ 标上出错标记

例 算术表达式文法G

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid i$$

(1) 消除G的左递归得到文法 G'

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid i$$

(2) 求出每个产生式的select集, G'是LL(1)文法

$$\text{SELECT}(E \rightarrow TE') = \{ (, i \}$$

$$\text{SELECT}(E' \rightarrow +TE') = \{ + \}$$

$$\text{SELECT}(E' \rightarrow \epsilon) = \{), \# \}$$

$$\text{SELECT}(T \rightarrow FT') = \{ (, i \}$$

$$\text{SELECT}(T' \rightarrow *FT') = \{ * \}$$

$$\text{SELECT}(T' \rightarrow \epsilon) = \{ +,), \# \}$$

$$\text{SELECT}(F \rightarrow (E)) = \{ (\}$$

$$\text{SELECT}(F \rightarrow i) = \{ i \}$$

(3) 依照选择集合把产生式填入分析表

	i	+	*	()	#
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow i$			$F \rightarrow (E)$		

注: 表中空白处为出错

步骤	分析栈	剩余输入串	所用产生式
1	#E	i+i*i#	E→TE'
2	#E'T	i+i*i#	T→FT'
3	#E'T'F	i+i*i#	F→i
4	#E'T'i	i+i*i#	i匹配
5	#E'T'	+i*i#	T'→ ε
6	#E'	+i*i#	E'→+TE'
7	#E'T+	+i*i#	+匹配

图 12: 描述'i+i*i'的分析过程

8	#E'T	i*i#	T→FT'
9	#E'T'F	i*i#	F→i
10	#E'T'i	i*i#	i匹配
11	#E'T'	*i#	T'→ *FT'
12	#E'T'F*	*i#	*匹配
13	#E'T'F	i#	F→i
14	#E'T'i	i#	i匹配
15	#E'T'	#	T'→ ε
16	#E'	#	E'→ ε
17	#	#	接受

递归子程序和预测分析法的对比:

两者的思想是一致的,根据当前的状态来选取合适的产生式进行推导

区别是递归下降子程序是利用递归程序的便利免除对环境的保存操作,而预测分析法则是手写栈保存过程中的记录。

6 LR分析法

LR分析法:根据当前分析栈种的符号串(通常以状态表示)和向右顺序查看输入串的K个($K \geq 0$)符号就可以唯一地确定句柄。

LR(k)的含义:

- L表示从左到右扫描输入串
- R表示最左规约(即最右推到的逆过程)
- k表示向右查看输入符号串的个数

当 $k = 1$ 时, 能满足当前绝大多数高级语言编译程序的需求

LR分析器的特点:

- 是规范规约
- 适用范围广, 适合于大多数上下文无关文法描述的语言
- 分析速度快, 能准确定位错误
- 缺点: LR分析器的构造工作量大

LR分析器的构成:

- 总控程序
所有LR分析器总共程序相同
- 分析表
 - 不同文法有不同的分析表
 - 同一文法采用的LR分析器不同, 分析表也不同
 - 分析表分为 **action表**(动作表) 和 **goto表**(状态转移表)
 - GOTO表表示当前状态面临文法符号时应转向的下一个状态
 - ACTION表表示当前状态面临输入符号时应采取的动作
- 分析栈
包括 状态栈S 和 文法符号栈X

行标题为状态, 列标题为文法符号

	ACTION						GOTO								
	a	c	e	b	d	#	a	c	e	b	d	#	S	A	B
0	S					#	2						1		
1						acc									
2		S		S						1		3			
3	r2	r2	r2	r2	r2	r2									

图 13: LR分析表

为节省空间:

	ACTION						GOTO		
	a	c	e	b	d	#	S	A	B
0	S2						1		
1						acc			
2		S1		S3					
3	r2	r2	r2	r2	r2	r2			

图 14: LR分析表2

ACTION表中的动作有4种:

- 移进 s_k
把状态 k 移入状态栈;
若当前状态(未移入时的栈顶)是 i ,且 $k = GOTO[i, a]$, 把 a 移入符号栈
注:状态栈栈顶为当前状态; k 为转移的状态,如果 k 为空的话,则报错
- 归约 r_k
用第 k 条产生式进行归约,此时栈顶形成了句柄 β ,文法中第 k 条产生式为 $A \rightarrow \beta$, 且 $|\beta| = r$
归约时从状态栈和符号栈弹出 r 个符号,把 A 移入符号栈, $j = GOTO[i, A]$ 移入状态栈,其中状态 i 为修改后的栈顶状态
- 接受(acc)
当符号栈只剩下文法开始符 S ,并且当前输入符为 $\#$, 则分析成功
- 报错
当状态栈顶的状态遇到了不应该出现的文法符号,则报错,说明输入串不是该文法的句子

6.1 LR(0)文法

- LR(0) 分析器在分析的过程中只根据符号栈的内容就能确定句柄,不需要向右查看输入符号
- 对文法的限制较大, 对绝大多数高级语言的语法分析器不适用
- 构造LR(0)分析表的思想和方法是构造其他LR分析表的基础

LR(k)分析法通过 活前缀来帮助确定句柄

- 规范句型的可归约前缀和活前缀
- 构造文法的识别活前缀及可归前缀的DFA
- 按DFA构造相应分析表
- 按进行表进行 $LR(k)$ 分析

可归前缀:规范句型中 句柄之前 包括 句柄在内 的串

活前缀:形成 可归前缀之前(包括可归前缀在内)的所有规范句型的前缀,即规范句型的不含句柄右边符号的前缀。

比如:规范句型 $aAbcde$ 中,若 Ab 为句柄,则可归前缀为 aAb ,活前缀为: ϵ, a, aA, aAb
可归前缀和活前缀在LR分析中的作用:

- 在LR分析过程中,实际上是把 活前缀 列出放在符号栈中
- 一旦在栈中出现 可归前缀 , 即 句柄 已经形成,就用相应的产生式进行归约
- 在分析的过程中,只要符号栈中的符号串是一个活前缀,就可保证已被分析过得部分是该文法规范句型的正确部分

使用 DFA 来识别前缀,为了构造这个DFA,需要扩广文法

扩广文法:使用 S' 作为文法的唯一开始,只要归约到 S' 即表示归约结束

扩广的原因:文法开始符 S 可能出现在产生式的右部,在归约过程中,不能判断是归约到最后开始符,还是归约到在产生式右部出现的开始符, S' 只出现在产生式的左部,确保不会混淆

利用LR分析表分析步骤:

- 查找产生式的右部
- 左部将归约后的结果入栈
- ACC动作,分析结束
- 空白部分表示出错

问题:怎么得到LR分析表?LR分析表→可归前缀→DFA→扩广文法+项目

LR(0)项目:在产生式的右侧适当位置添加一个小圆点,构成一个项目,小圆点后面部分表示待分析项
比如:产生式 $U \rightarrow XYZ$ 对应4个项目: $U \rightarrow \cdot XYZ, U \rightarrow X \cdot YZ, U \rightarrow XY \cdot Z, U \rightarrow XYZ \cdot$

注: $A \rightarrow \epsilon$ 只有一个项目 $A \rightarrow \cdot$

项目的出现是为了标明当前所处的位置,以便后续的输入顺利进行。

圆点的含义:

- 圆点在最左边,表示希望有产生式的右部归约
- 圆点在最右边,表示句柄已经形成,可以进行归约
- 圆点左部的内容,表示分析过程中已经识别过的部分
- 圆点右部的内容,表示待识别部分

项目分类:

- 移进项目,圆点后面为 终结符
这个项目输入 a ,表示移进操作
- 待约项目,圆点后面为 非终结符
- 归约项目,圆点在最右端
表示句柄已经形成,在该项目下应该进行归约操作
- 接受项目: $S' \rightarrow S \cdot$
- 开始项目: $S' \rightarrow \cdot S$

构造识别活前缀的NFA:

- 扩广文法
- 构造新状态,产生 $LR(0)$ 项目
所有项目为NFA的一个状态,开始项目为初态,其他项目为活前缀的识别态(其中,归约项目为终态,接受项目为接受态)
- 确定状态的转换关系
对于项目 $i : A \rightarrow a \cdot X b, j : A \rightarrow aX \cdot b$
 - i和j连一条标记为X的箭弧
 - 若 X 是非终结符($X \in V_N$),对于 X 的所有产生式圆点在最左边的项目k, i和所有k都连一条标记为 ϵ 的箭弧
- $NFA \rightarrow DFA$

整理:

- 考虑分析过程需要,引入项目的概念
- 随着分析过程,待归约串会发生变化,因而输入非终结符 X 需 X 和 ϵ 两种弧(未识别出 X 的时候,走 ϵ ,当识别出 X 时,走 X 弧)

上述描述太过麻烦,工作量大,不实用,能否直接构造出DFA?解:可以,使用项目集作为DFA的状态运算;项目集闭包,空串引发的子节点均合并到父节点上(实际上就是把上面 $NFA \rightarrow DFA$ 的过程合并到创建过程上)

例: $S' \rightarrow E, E \rightarrow aA|bB, A \rightarrow cA|d, B \rightarrow cB|D$

$I = \{S' \rightarrow \cdot E\}$

则 $CLOSURE(I) = \{S' \rightarrow \cdot E, E \rightarrow \cdot aA, E \rightarrow \cdot bB\}$,为DFA的一个状态(因为后两条·后面是非终结符,因此停止)

拓广文法 \mathbf{G}' :

- (0) $S' \rightarrow E$
- (1) $E \rightarrow aA$
- (2) $E \rightarrow bB$
- (3) $A \rightarrow cA$
- (4) $A \rightarrow d$
- (5) $B \rightarrow cB$
- (6) $B \rightarrow d$

识别活前缀的DFA:

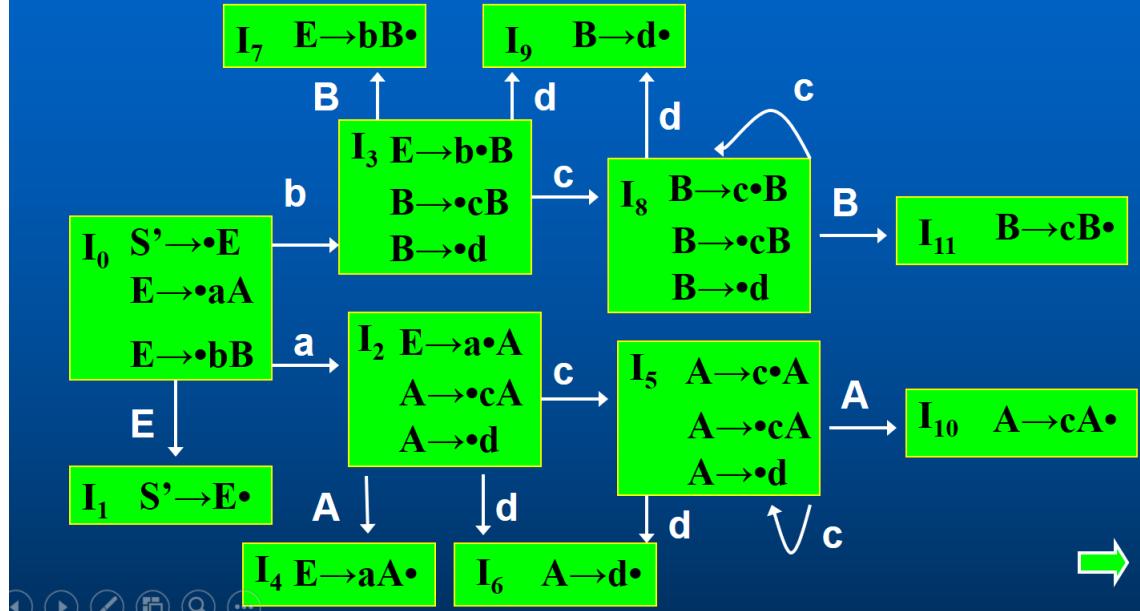


图 15: 识别活前缀的DFA

LR(0)分析表的构造:由DFA生成LR(0)分析表($S' \rightarrow \cdot S$ 表示初始状态)

- 输入终结符(LR(0)分析法要求这移进和归约操作不发生冲突)
 - 如果项目集有归约项目,那么执行归约操作
 - 否则执行移进操作
- 输入非终结符,直接进行状态转移

问:如果当前的项目集既有归约项目进行归约,又能后续进行移进操作,怎么处理?SLR(0), LR(1), LALR

状态	ACTION					GOTO		
	a	b	c	d	#	E	A	B
0	S_2	S_3				1		
1					acc			
2			S_5	S_6		4		
3			S_8	S_9			7	
4	r_1	r_1	r_1	r_1	r_1			
5			S_5	S_6		10		
6	r_4	r_4	r_4	r_4	r_4			
7	r_2	r_2	r_2	r_2	r_2			
8			S_8	S_9			11	
9	r_6	r_6	r_6	r_6	r_6			
10	r_3	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5	r_5			

图 16: 从DFA构造LR(0)分析表

由LR(0)分析表分析过程:查表→状态转移→acc或报错

- 移进 s_k
把状态 k 移入状态栈,若当前状态是 i ,且 $k = GOTO[i, a]$, 把 a 移入符号栈
- 归约 r_k
用第 k 条产生式进行归约,此时栈顶形成了句柄 β ,文法中第 k 条产生式为 $A \rightarrow \beta$, 且 $|\beta| = r$
归约时从状态栈和符号栈弹出 r 个符号,把 A 移入符号栈, $j = GOTO[i, A]$ 移入状态栈,其中状态 i 为修改后的栈顶状态
- 接受(acc)
当符号栈只剩下文法开始符 S ,并且当前输入符为 $\#$, 则分析成功
- 报错
当状态栈顶的状态遇到了不应该出现的文法符号,则报错,说明输入串不是该文法的句子

← (0) $S' \rightarrow E$ (1) $E \rightarrow aA$ (2) $E \rightarrow bB$ (3) $A \rightarrow cA$
 (4) $A \rightarrow d$ (5) $B \rightarrow cB$ (6) $B \rightarrow d$

输入串bccd# 的分析过程

步骤	状态栈	符号栈	输入串	ACTION	GOTO
1	0	#	bccd#	S_3	
2	03	#b	ccd#	S_8	
3	038	#bc	cd#	S_8	
4	0388	#bcc	d#	S_9	
5	03889	#bccd	#	r_6	11
6	0388(11)	#bccB	#	r_5	11
7	038(11)	#bc B	#	r_5	7
8	03 7	#b B	#	r_2	1
9	0 1	#E	#	acc	

图 17: 使用LR(0)分析表进行分析

6.2 SLR(1)文法

文法可能存在‘移进→归约’和‘归约→归约’冲突

LR(0)文法:项目集中既不存在‘移进→归约’也不存在‘归约→归约’冲突的文法

解决冲突的方法:使用SLR(1)文法:向前看一个输入符号,对于归约项目 $A \rightarrow r\cdot$ 只有当后面的输入符号 a ,满足 $a \in FOLLOW(A)$ 时才使用A的产生式进行归约

例:对于 $I = \{A \rightarrow a \cdot bc, A \rightarrow r \cdot, B \rightarrow e\}$

若 $\{b\}, FOLLOW(A), FOLLOW(B)$ 两两不相交,当状态I面临输入符 a 时可采取以下动作:

- 若 $a = b$, 移进
- 若 $a \in FOLLOW(A)$, 则用 $A \rightarrow r$ 归约
- 若 $a \in FOLLOW(B)$, 则用 $B \rightarrow e$ 归约
- 否则,报错

例子:P156 练习题T1 ,手机图片

6.3 LR(1)文法

SLR文法并不能完全解决冲突问题,因此需要进一步判断(LR(1)文法)

基本思想:

LR(1)方法按每个具体的句型设置展望信息。

例: 如果存在如下的一些句型

$\dots\alpha A a\dots, \dots\beta A b\dots, \dots\gamma A c\dots$, 则

$FOLLOW(A)=\{a,b,c\}$

处理到句型 $\dots\alpha A$, 只当输入符号为**a**时归约;

处理到句型 $\dots\beta A$, 只当输入符号为**b**时归约;

处理到句型 $\dots\gamma A$, 只当输入符号为**c**时归约;

图 18: LR(1)分析的基本思想

LR(1)文法: 对于产生式 $A \rightarrow \alpha B \beta$, 除了看 $FOLLOW(B)$ 之外, 还要看 $FIRST(\beta)$, 并把 $FIRST(\beta)$ 作为项目集的一个部分, 通过这样进一步缩小可归约的范围

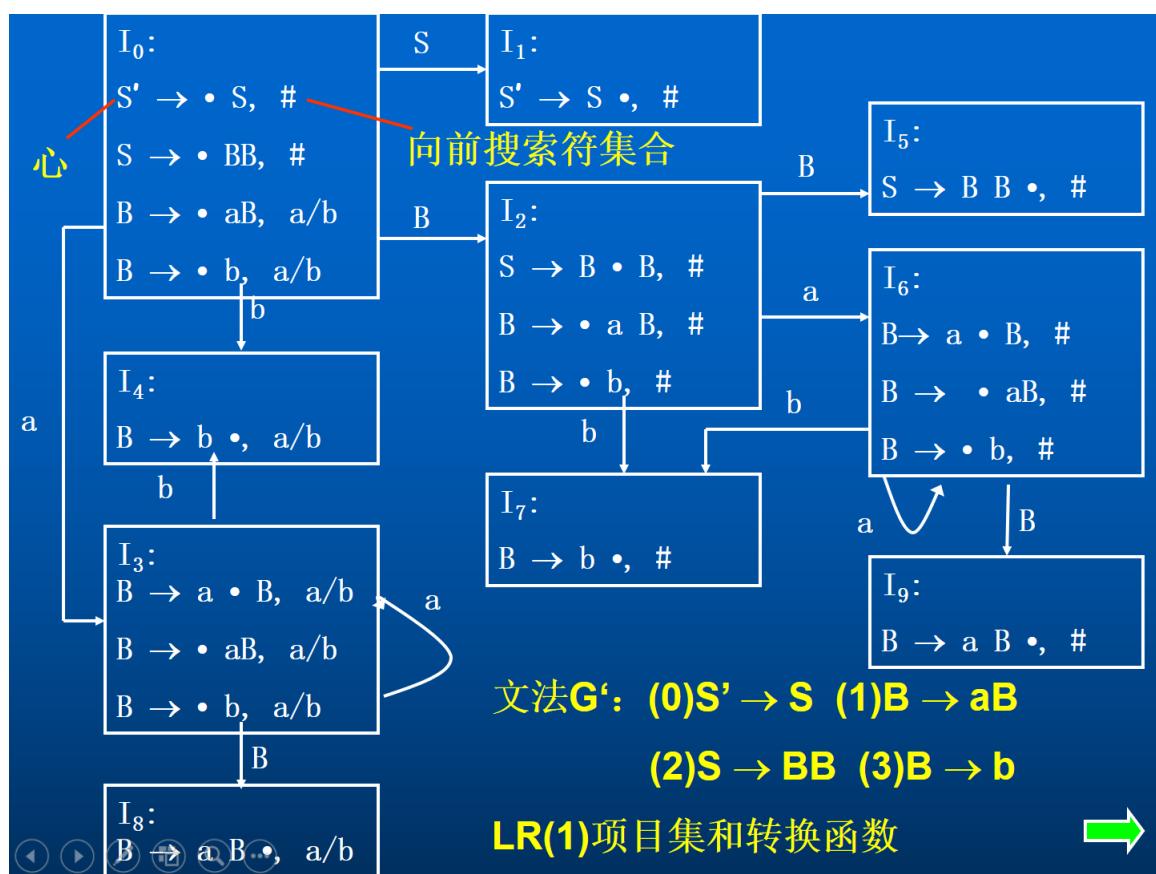


图 19: LR(1)分析的DFA

LR(1)文法的优点: 使用广度大

LR(1)文法的缺点: LR(1)文法的状态数目庞大

6.4 LALR(1)文法

LALR(1)文法:LR(1)的状态数目太过庞大,不好计算,只有在SLR(1)文法无法解决问题的部分,才采用LR(1)文法进行解决。

7 中间代码生成

7.1 属性文法

属性:对文法的每个符号,引进一些属性,这些属性代表与文法符号相关的信息,如类型、值、存储位置等

语义规则:为文法的每个产生式配备的计算属性的计算规则

属性文法:带属性的文法

例: $E \rightarrow A + B \{A.t = int \quad AND \quad B.t = int\}$

(花括号内为 E 的语义规则)

属性分类:

- 综合属性

属性值由该节点的子节点的属性值计算来

用于自下而上传递信息

- 继承属性

属性值由该节点的兄弟节点或父节点的属性值计算而来

用于自上而下传递信息

终结符只有综合属性,他们由词法分析器提供

非终结符既有综合属性也有继承属性,但文法开始符没有继承属性

7.2 语法制导的翻译

基本思想:在语法分析过程中,随着分析的步步进展,每当使用一条产生式进行推导或归约,就执行该产生式所对应的语义动作,完成相应的翻译工作

语法制导翻译法不论对自上而下还是自下而上的分析都适用

一旦语法分析确认输入符号串是一个句子,它的值也同时由语义规则计算出来

对于LR分析(自下而上)的语法分析来说,在归约后调用语义规则的功能需要增加予以栈,语义值放到与符号栈同步操作的语义栈中,多项语义值可设多个语义栈

栈结构格式为:

状态栈	符号栈	语义栈
S_m	X_m	$X_m.val$
...
S_1	X_1	$X_1.val$
S_0	#	-

图 20: LR分析栈结构

例:

- 1) $L \rightarrow E$
 $\{ \text{print}(E.val) \}$
- 2) $E \rightarrow E^1 + T$
 $\{ E.val := E^1.val + T.val \}$
- 3) $E \rightarrow T$
 $\{ E.val := T.val \}$
- 4) $T \rightarrow T^1 * \text{digit}$
 $\{ T.val := T^1.val * \text{digit.lexval} \}$
- 5) $T \rightarrow \text{digit}$
 $\{ T.val := \text{digit.lexval} \}$

状态	ACTION				GOTO	
	d	+	*	#	E	T
0	S_3				1	2
1		S_4		acc		
2		r3	S_5	r_3		3
3		r5	r5	r5		
4	S_3					7
5	S_6					
6		r4	r4	r4		
7		r2	S_5	r_2		

图 21: 简单算数表达式求值的属性文法

分析并计算 $2+3*5$ 的过程如下:

步骤	状态栈	语义栈	符号栈	剩余输入串	Action	GOTO
0	0	-	#	$2+3*5\#$	S_3	
1	03	--	#2	$+3*5\#$	r_5	2
2	02	-2	#T	$+3*5\#$	r_3	1
3	01	-2	#E	$+3*5\#$	S_4	
4	014	-2-	#E+	$3*5\#$	S_3	
5	0143	-2--	#E+3	*5#	r_5	7
6	0147	-2-3	#E+T	*5#	S_5	
7	01475	-2-3-	#E+T*	5#	S_6	
8	014756	-2-3-5	#E+T*5	#	r_4	7
9	0147	-2-15	#E+T	#	r_2	1
10	01	-17	#E	#	acc	

图 22: 计算 $2+3*5$

7.3 中间代码

中间代码: 中间代码是一种复杂性介于源程序语言和机器语言之间的一种表示形式,与具体机器无关,可以进行与机器无关的优化

中间代码的形式:

- 逆波兰记号
也叫后缀式,运算对象写在前面
- 三元式
- 四元式
- 树

7.4 中间代码生成

属性和语义规则中用到的变量、过程和函数:

属性:

- id.name
表示单词id的名字
- E.place
表示存放E值的变量名在符号表的入口地址或临时变量编码

变量、函数和过程:

- nextstat
给出在输出序列中下一个四元式的序号
- lookup(id.name)
查询id.name是否出现在符号表中,是则返回id的入口地址,否则返回nil
- emit
向输出序列输出一个四元式
emit每调用一次,nextstat的值增加1
- newtemp
生成一个新的临时变量
- error
进行错误处理

7.4.1 简单赋值语句的翻译

语义主要审查标识符是否已经声明

(1) $S \rightarrow id := E$
 $\{ p := \text{lookup}(\text{id.name});$
 $\quad \text{if } p \neq \text{nil} \text{ then emit}(:=, E.\text{place}, -, p)$
 $\quad \text{else error} \}$

•

(2) $E \rightarrow E^1 + E^2$
 $\{ E.\text{place} := \text{newtemp};$
 $\quad \text{emit}(+, E^1.\text{place}, E^2.\text{place}, E.\text{place}) \}$

•

(3) $E \rightarrow E^1 * E^2$
 $\{ E.\text{place} := \text{newtemp};$
 $\quad \text{emit}(*, E^1.\text{place}, E^2.\text{place}, E.\text{place}) \}$

•

(4) E \rightarrow -E¹

```
{ E.place:=newtemp ;
  emit ( @ , E1.place , - , E.place ) }
```

•

(5) E \rightarrow (E¹)

```
{ E.place:=E1.place }
```

•

(6) E \rightarrow id

```
{ p:=lookup ( id.name ) ;
  if p $\neq$ nil then E.place:=p
  else error }
```

•

7.4.2 布尔表达式的翻译

直接翻译:把结果保存在临时变量,对于比较运算,则采用赋值语句进行结果的赋值

(1) E \rightarrow E¹ or E²

```
{ E.place := newtemp ;
  emit ( or , E1.place , E2.place , E.place ) }
```

(2) E \rightarrow E¹ and E²

```
{ E.place := newtemp ;
  emit ( and , E1.place , E2.place , E.place ) }
```

(3) E \rightarrow not E¹

```
{ E.place := newtemp ;
  emit ( not , E1.place , __ , E.place ) }
```

(4) E \rightarrow (E¹)

```
{ E.place := E1.place }
```

(5) E \rightarrow id₁ rop id₂

```
{ E.place := newtemp ;
  emit ( jrop , id1.place , id2.place , nextstat+3 ) ;
  emit ( :=, 0 , __ , E.place ) ;
  emit ( jump , __ , __ , nextstat+2 ) ;
  emit ( :=, 1 , __ , E.place ) }
```

(6) E \rightarrow true

```
{ E.place:=newtemp;emit(:=,1,-,E.place)}
```

(7) E \rightarrow false

```
{E.place:=newtemp;emit(:=,0,-,E.place)}
```

短路翻译:对于布尔表达式,一旦确定了真假,就无需进行后续的判断,直接进行跳转即可
主要有以下3种代码:

1. **(jnz , A , - , p)**
若A为真, 则转向四元式p
2. **(jrop , A , B , p)**
若A rop B为真, 则转向四元式p
3. **(jump , - , - , p)**
无条件转向四元式p

对于 $a \text{ } rop \text{ } b$ 形式,生成以下代码:

(jrop , a , b , E.true)
(jump , - , - , E.false)

但是,在翻译E的时候,我们并不能确定两个跳转语句将要跳转的位置,因此我们需要保存这两条四元式的序号,以便生成完后续的代码之后进行回填

- E.true
是一个链表,E的中间代码中,判断E为”真”时发生跳转的四元式的编号
- E.false
是一个链表,E的中间代码中,判断E为”假”时发生跳转的四元式的编号

为了完成翻译过程:我们定义以下过程和变量:

- merge(p1,p2)
把p1和p2为链首的两条链,并返回合并后的链首
- backpatch(p,t)
把链首p所连接的每个四元式的第四区段填为转移目标t
- E.codebegin
表示E的第一个四元式的序号

1) E→E¹ or E²

```
{ E.codebegin:=E1.codebegin ;  
  backpatch ( E1.false , E2.codebegin ) ;  
  E.true:=merge ( E1.true , E2.true ) ;  
  E.false:=E2.false }
```

2) E→E¹ and E²

```
{ E.codebegin:=E1.codebegin ;  
  backpatch ( E1.true , E2.codebegin ) ;  
  E.true:=E2.true ;  
  E.false:=merge ( E1.fasle , E2.false ) }
```

3) E→not E¹

```
{ E.codebegin:=E1.codebegin ;  
  E.true:=E1.false ;  
  E.false:=E1.true }
```

4) E→(E¹)

```
{ E.codebegin:=E1.codebegin ;  
  E.true:=E1.true ;  
  E.false:=E1.false }
```

5) E→id₁ rop id₂

```
{ E.codebegin:=nextstat ;  
  E.true:=nextstat ;  
  E.false:=nextstat+1;  
  emit ( jrop , id1.place , id2.place , 0 ) ;  
  emit ( jump , −, −, 0 ) }
```

6) E→true

```
{ E.codebegin:=nextstat ;
  E.true:=nextstat ;
  E.false:=0;
  emit ( jump ,—,—,0 ) }
```

7) E→false

```
{ E.codebegin:=nextstat ;
  E.false:=nextstat ;
  E.true:=0;
  emit ( jump ,—,—,0 ) } ↗
```

7.4.3 if,while,repeat 语句的翻译

C→if E then

```
{ backpatch ( E.true , nextstat ) ;
  C.CHAIN:=E.false }
```

S→C S¹ /* if E then S¹ */

```
{ S.CHAIN:=merge ( C.CHAIN , S1.CHAIN ) }
```

T^p→C S¹ else /* if E then S¹ else */

```
{ q:=nextstat ;
  emit ( jump,—,—,0 ) ;      /*S1执行完，跳离整个if语句*/
  backpatch ( C.CHAIN , nextstat ) ;
  Tp.CHAIN:=merge ( q , S1.CHAIN ) }
```

S→T^p S² /* if E then S¹ else S² */

```
{ S.CHAIN:=merge ( Tp.CHAIN , S2.CHAIN ) }
```

W→while

```

{ W.codebegin:=nextstat }

Wd→W E do /*while E do*/
    { Wd.codebegin:=W.codebegin ;
      backpatch ( E.true , nextstat ) ;
      Wd.CHAIN:=E.false }

S→ Wd S3 /*while E do S3*/

    { backpatch ( S3.CHAIN , Wd.codebegin ) ;
      emit ( jump ,—,—,Wd.codebegin) ;
      /*S3执行完, 跳至While语句开头*/
      S.CHAIN:=Wd.CHAIN }
  
```

C.chain的含义:表示C的四元式中,跳转到C的代码块末尾(出口)的四元式的集合

实际上,E.true和 E.false已经能够翻译出语句了,但是为什么还需要C.chain?

- 对于语句嵌套的时候,如果没有回填chain,那么就需要经过 多次跳转才能跳到出口
- 在 if 语句中,then 代码块结束之后,需要跳过else的代码块,如果使用chain之后,就不需要在if的子程序里面对这个jmp进行填写

比如:以下代码 while E do if A then B

在while结束的时候,会有一条 jmp1语句跳转到while语句的开头

在 if 语句为false的时候,会有 jmp2 跳转到 if语句块的结束,也就是 jmp1 语句,再由jmp1跳转到while开头(注意,这里jmp 了两次)

由chain的定义,我们可以知道 IF.chian = jmp2 , 也就是前图中的 S³.chain = jmp2

我们 回填 while.codebegin 到 jmp2的话,那么执行 jmp2的时候,就直接到了while 的开头,而无需再经过 jmp1 (划重点:回填chain后,只jmp了一次)

例:

■ **while A<B do if C<D then X:=Y+Z 的最终翻译结果为:**

```
100 ( j<, A , B , 102 )
101 ( jump ,—,—, 0 )
102 ( j<, C , D , 104 )
103 ( jump ,—,—,100 )
104 ( +, Y , Z , t1 )
105 ( :=, t1 ,—, X )
106 ( jump ,—,—,100 )
S.CHAIN=101
```

while A<B do if C<D then X:=Y+Z 的翻译过程:

(1) 把**while**归为**W**, 记住**while**语句的入口为**100**

(2) 把**A<B**归为**E¹**, 产生:

```
100 ( j<, A , B , 0 )      E1.true=100
101 ( jump,—,—,0 )      E1.false=101
```

(3) 把**W E do**归为**W^d**, 回填**E¹.true** 得到

```
100 ( j<, A , B , 102 )
Wd.CHAIN=E1.false=101
```

(4) 把**C<D**归为**E²**, 产生:

```
102 ( j<, C , D , 0 )      E2.true=102
103 ( jump,—,—,0 )      E2.false=103
```

(5) 把 **if E² then** 归为 C, 回填 E².true 得

102 (j<, C, D, 104)

C.CHAIN = E².false = 103

(6) 把 **X := Y + Z** 归为 S¹, 产生:

104 (+, Y, Z, t₁)

105 (:=, t₁, -, X)

S¹.CHAIN = 0

(7) 把 C S¹ 归为 S², S².CHAIN = merge (103, 0) = 103

(8) 把 W^d S² 归为 S, 回填 S².CHAIN 得

103 (jump, -, -, 100)

产生四元式 **106 (jump, -, -, 100)**

S.CHAIN := W^d.CHIAN = 101

8 符号表

符号表: 符号表是存放标识符信息的一种表, 其中的信息表示是标识符的属性(语义)

符号表的作用: 符号表是连接声明和引用的桥梁。

- 一个名字在声明时, 相关信息被写进符号表
- 在引用时, 根据符号表中的信息生成相应的可执行语句

符号表的生存周期: 编译过程中, 每当遇到标识符时, 就要查填符号表

- 若是新的标识符, 就向符号表中填入一个新的表项
- 否则, 根据情况向符号表中的已有表项 增填 信息(如填入分析的存储地址)或者 查获信息(如语义检查)

9 代码优化

代码优化: 对代码进行等价变换, 使得变换后的代码效率更高(节省运行时间、存储空间或者两者兼而有之)

- 中间代码优化(不依赖具体计算机)
- 目标代码优化(依赖具体计算机)

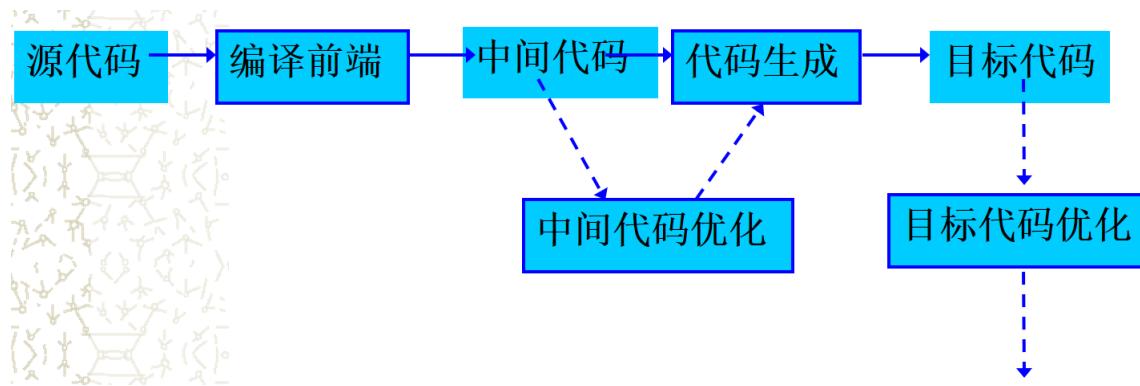


图 23: 编译的优化工作阶段

优化的分类:

- 局部优化
在只有一个入口、一个出口的基本快上进行优化
- 循环优化
对循环中的代码进行优化
- 全局优化
在整个程序范围内进行的优化

中间代码优化 中间代码常用计数:

- 删除多余运算(删除公共子表达式)
如果子表达式E在前面计算过,且之后E中的变量值都未改变,那么E的重复出现称为 **公共子表达式**,可避免重复计算
- 合并已知量与复写传播
如果运算量都是已知量,则在编译时就算出它的值,称 **合并已知量**
若有 $A := B$,称为把B **复写**到A,如果其后有引用A的地方,且其间A、B的值都未改变,则可把对A的引用改为对B的引用,称为 **复写传播**
- 删除无用赋值
有些变了的赋值从未被引用,称为无用赋值,应删除
 - 变量被赋值,但在程序中从未被引用
 - 变量赋值后未被引用又重新赋值,则前面赋值是无用的
 - 变量的赋值只计算变量自己引用,其他变量都不引用它

9.1 局部优化

局部优化:指基本块内的优化

基本块:是指一个程序中顺序执行的语句序列,其中只有一个入口语句和出口语句。执行时 **只能从入口语句进入,从其出口语句退出**

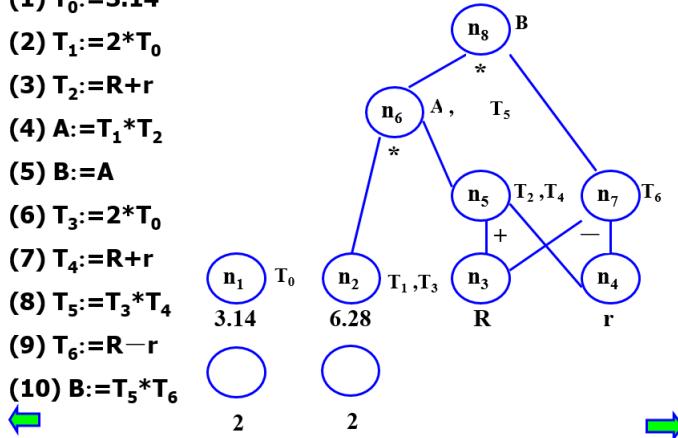
划分基本块:

- 求基本块的入口语句:
 - 程序的第一个语句
 - 条件转移或无条件转移语句的转移目标语句
 - 紧跟在条件转移语句后面的语句
- 对每一入口语句,构造其所属的基本块
 - 它是由该入口语句到下一入口语句
 - 或到转移语句
 - 或到停止语句
- 凡未被纳入某一基本块的语句,是不会被执行到的语句,可以把它们删除

利用DAG进行优化

例: 构造以下基本块的DAG

- (1) $T_0 := 3.14$
- (2) $T_1 := 2 * T_0$
- (3) $T_2 := R + r$
- (4) $A := T_1 * T_2$
- (5) $B := A$
- (6) $T_3 := 2 * T_0$
- (7) $T_4 := R + r$
- (8) $T_5 := T_3 * T_4$
- (9) $T_6 := R - r$
- (10) $B := T_5 * T_6$



原基本块的四元式序列G 按DAG重新写成的四元式序列G'

(1) $T_0 := 3.14$	(1) $T_0 := 3.14$
(2) $T_1 := 2 * T_0$	(2) $T_1 := 6.28$
(3) $T_2 := R + r$	(3) $T_3 := 6.28$
(4) $A := T_1 * T_2$	(4) $T_2 := R + r$
(5) $B := A$	(5) $T_4 := T_2$
(6) $T_3 := 2 * T_0$	(6) $A := 6.28 * T_2$
(7) $T_4 := R + r$	(7) $T_5 := A$
(8) $T_5 := T_3 * T_4$	(8) $T_6 := R - r$
(9) $T_6 := R - r$	(9) $B := A * T_6$
(10) $B := T_5 * T_6$	> G中(2)(6)的已知量已合并 > G中(5)的无用赋值已删除 > G中(3)(7)的公共子表达式R+r 只计算一次, 删除了多余运算

✿ 利用DAG进行优化

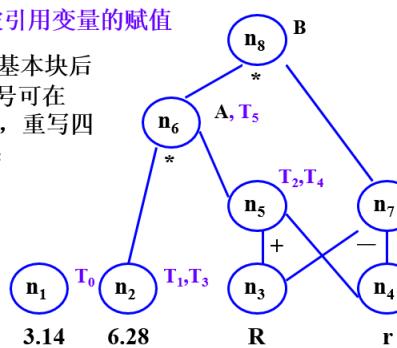
删除在基本块后不被引用变量的赋值

假如 T_0, T_1, \dots, T_6 在基本块后都不被引用，则这些符号可在DAG附加标识符中删去，重写四元式得到进一步的优化：

- (1) $S_1 := R + r$
- (2) $A := 6.28 * S_1$
- (3) $S_2 := R - r$
- (4) $B := A * S_2$

其中 S_1 和 S_2 是临时变量。

T_0, T_1, \dots, T_6 被赋值的代码被优化掉



[

9.2 循环优化

- 代码外提

把循环不变运算,即结果独立于循环执行次数的表达式,提到循环外面,使之只在循环外计算一次

- 强度削弱

把程序中强度大的运算替换成强度小的运算

比如把乘法运算换成加法运算

- 变换循环控制条件(删除归纳变量)