编译原理复习提纲

# 编译程序模型

输入：源程序

分析阶段：词法分析、语法分析、语义分析

综合阶段：中间代码优化、优化、目标代码生成器

两个阶段均有：错误处理、符号表管理

输出：目标代码

## 词法分析

内码：二元式

## 语法分析

语法规则用BNF表示

## 语义分析和中间代码生成

中间代码常用：逆波兰式、三元式、四元式、抽象机代码表示

## 代码优化

等价变换原则

## 目标代码生成

绝对指令代码→可重定位指令代码或汇编指令代码

## 表格管理

常见符号表：名字特性表、常数表、标号表、分程序入口表、中间代码表

## 前端和后端

前端：编译逻辑结构中的分析部分

后端：与目标机器有关的部分

# 文法和语言

## 产生式

## 文法

## 句型

## 规范推导

最右推导为规范推导，规范推导产生的句型为规范句型。

## 递归规则

### 直接左递归的产生式

### 间接左递归的产生式

### 递归文法

至少含一个递归的非终结符号。

## 短语

有句型：

### 简单（直接）短语

### 句柄

句型的**最左**简单短语

## 上下文无关文法（2型、CFG）

左边只有一个非终结符。

## 正则文法（3型）

## CFG的化简

### 消除无用符号

#### 可达

#### 产生

#### 有用

若X同时是产生的和可达的，则称X是有用的，否则为无用符号。

#### 消除算法

##### 计算“产生的”符号集算法

1. 每个T中的符号都是产生的
2. 若有产生式且中符号都是产生的，则A是产生的

##### 计算“可达的”符号集算法

1. 符号S是可达的
2. 若有产生式且是可达的，则中的符号都是可达的

##### 示例

消除非产生的

消除非可达的

必须先消除非产生的，再消除非可达的。

### 消除

形如的产生式为空产生式或。

#### 算法

先确定全部可空的变元：

1. 若，则A可空
2. 若且中每个符号都是可空的，则B可空

再替换带可空符号的产生式，若是产生式，那么用所有的替代，其中：

1. 若不是可空的，则
2. 若是可空的，则或
3. 但不能全部为

#### 示例

消除后

### 消除单元产生式

形如。

代入消除即可。

### 简化的可靠顺序

1. 消除
2. 消除单元产生式
3. 消除非产生的无用符号
4. 消除非可达的无用符号

# 词法分析

## 有穷自动机

### 确定的有穷自动机（DFA）

### 非确定的有穷自动机（NFA）

### NFA→DFA

#### move

#### 算法

计算起始状态的，记为，依次计算。

### DFA最小化

标记状态对。（即终点和非终点不能为一类）

若被标记了，则标记，重复该步骤，直到无可标记。（即目的地不在一类的点对不在一类）

#### 定理

对于有同一接受集的FA，与之等价且具有最小状态数的DFA在同构意义下是唯一的。

## 正则式（正规式）

与FA等价。

# 自顶向下语法分析

## 下推自动机

## LL(1)文法

### 构造

#### FIRST集（开头集，可能的开头的终结符集合）

若，则

#### FOLLOW集（跟随集，后面可能跟随的终结符集合）

若，则

#### SELECT集（产生式预测集）

### 判别

相同左部的产生式的集的交集都为空，则是LL(1)文法。

### 非LL(1)转换为LL(1)

* 提取左公共因子
* 消除左递归
  + 直接左递归
  + 间接左递归

### 预测分析表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

#### 分析过程

|  |  |  |  |
| --- | --- | --- | --- |
| 步骤 | 分析栈 | 剩余输入串 | 所用产生式或匹配 |
| *1* |  |  |  |
| *2* |  |  |  |
| *3* |  |  |  |
| *4* |  |  |  |
| *5* |  |  |  |
| *…* | *…* | *…* | *…* |

# 自底向上优先分析

## 算符优先分析法

### 优先关系

### 构造优先关系矩阵

#### 集

#### 集

#### 算法

对每个产生式

1. 若和都是终结符：
2. 若是终结符是非终结符：
3. 若是非终结符是终结符：

规定#优先级比相邻任何运算符都低。

不可以同时出现任意两种。

### 素短语

* 至少包含一个终结符
* 除他自身，不再包含其他素短语

#### 最左素短语

满足

### 规约过程

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 栈 | 优先关系 | 当前符号 | 剩余输入串 | 移进或规约 |
|  |  |  |  | 移进 |
|  |  |  |  | 规约 |
|  |  |  |  | 移进 |
|  |  |  |  | 移进 |
|  |  |  |  | 规约 |
|  |  |  |  | 规约 |
|  |  |  |  | 接受 |

## LR(0) 分析

### 拓广文法

增加产生式

### 活前缀

规范推导（最右推导）：

其逆过程为最左规约（规范规约）。

每次规约前句型的前部，称为可归前缀。

把形成可归前缀之前，包括可归前缀的所有前缀，称为活前缀。

#### 计算不包含句柄的活前缀

即，若有产生式

则

#### 计算包含句柄的活前缀

#### 例

求不包含句柄在内的活前缀方程组：

所以包含句柄的活前缀为：

### 项目集规范族

#### LR(0) 项目

在产生式的右部每个空隙加一个圆点。

#### 构造NFA/DFA

#### LR(0) 分析表构造

*若*

*若*

*若*

最终表形如：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态 |  | | | | |  | | |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

### LR(0) 分析器工作

如上例中的分析表，分析

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态栈 | 符号栈 | 输入串 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

遇到规约后，状态栈弹出个状态，即第个产生式右部符号数。

然后以此时栈顶状态为当前状态，当前输入为规约后的非终结符，找到对应的。

## SLR(1) 分析

### 基本思路

利用非终结符号的FOLLOW集，判断“规约”还是“移进”。

### 解决冲突

若LR(0)的规范族含有如下项目集

存在移进-移进冲突和移进-规约冲突。

若

则当状态面临输入符号时，

## LR(1) 分析

### LR(1) 项目集族的构造

起始项目为

### LR(1) 分析表构造

*项目*

其余不变

## LALR(1) 分析

### 基本思路

合并同心集而不产生冲突。超前搜索符为之前的合集。

## 优先关系解决二义性文法

例如状态：

由于，所以遇移进，而符合左结合，所以遇规约。

又例如状态：

由于，所以无论如何都规约。

# 语法制导翻译和中间代码生成

## 属性文法

对每个产生式，有一套与之相关的语义规则。规则形式为：

其中，和

即综合属性是自下而上的，继承属性是自上而下的。

## 中间代码

### 逆波兰记号

运算对象在前，运算符号在后。

例：

### 三元式

例：

写为

#### 间接三元式

操作表和三元式表。

操作表为三元式表的索引，表示执行顺序。

例：

写为

三元式表

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

操作表：(1)(2)(3)(4)(1)(5)

### 四元式

普遍采用。

例：

写为

### 抽象语法树

节点序号为自底向上分析器创建该树的节点的次序。

## 翻译

newtemp为生成临时变量。

### 简单赋值语句

(op, arg1, arg2, result)

S -> id:=E {

p := lookup(id.name);

if p != null then emit(p ‘:=’ E.place) else error;

}

E -> E1+E2 {

E.place := newtemp;

emit(E.place ‘:=’ E1.place ‘+’ E2.place);

}

E -> -E1 {

E.place := newtemp;

emit(E.place ‘:=’ ‘uminus’ E1.place);

}

E -> (E1) {

E.place := E1.place;

}

E -> id {

E.place := newtemp;

p := lookup(id.name);

if p != nil then E.place := p else error;

}

### 布尔表达式

E -> id1 rop id2 {

E.place := newtemp;

emit(‘if’ id1.place ‘rop’ id2.place ‘goto’ nextstat+3);

emit(E.place ‘:=’ 0);

emit(‘goto’ nextstat+2);

emit(E.place ‘:=’ ‘1’);

}

E -> true {

E.place := newtemp;

emit(E.place ‘:=’ ‘1’);

}

E -> false {

E.place := newtemp;

emit(E.place ‘:=’ ‘0’);

}

#### 控制语句中的布尔表达式

E -> E1 or E2 {

backpatch(E1.false, E2.codebegin);

E.codebegin := E1.codebegin;

E.true := merge(E1.true, E2.true);

E.false := E2.false;

}

E -> E1 and E2 {

backpatch(E1.true, E2.codebegin);

E.codebegin := E1.codebegin;

E.false := merge(E1.false, E2.false);

E.true := E2.true;

}

E -> not E1 {

E.true := E1.false;

E.codebegin := E1.codebegin

E.false := E2.true;

}

E -> id1 rop id2 {

E.true := nextstat;

E.codebegin := nextstat;

E.false := nextstat+1;

emit(‘if’ id1.place ‘rop’ id2.place ‘goto’ \_\_\_);

emit(‘goto’ \_\_\_);

}

E -> true {

E.true := nextstat;

E.codebegin := nextstat;

emit(‘goto’ \_\_\_);

}

E -> false {

E.false := nextstat;

E.codebegin := nextstat;

emit(‘goto’ \_\_\_);

}

其中backpatch为回填，merge为合并真/假链。

扫描到then…else…再回填真/假链出口的空。

# 符号表

## 作用

* 收集符号属性
* 检查语义正确性
* 辅助生成代码

## 符号表的内容

### 标识符的名称

两种存储方法：

* 定长：标识符名称域规定最大长度
* 集中：开辟一个存放所有标识符的缓冲区，标识符名称域只存放偏移地址和长度

### 信息区

#### 不同种属名字，建立不同的符号表

如常数表、变量名表、过程名表

#### 最大单一符号表+

建立各类符号所有的属性项，形成一个单一的大符号表。

#### 折中结构

将属性信息类似的符号分在一组，为每组建立单一符号表。

#### 统一符号表的链接式结构

为多数符号的定长属性项确定“基本长度”，对需要空间多的符号，占用多个“基本长度”，用指针相连。

### 非分程序结构语言的符号表的组织

#### 无序表

插入简单、查找效率低

#### 有序表

插入需要附加查找

查找效率高于无序表

#### 散列（Hash）表

平均查找次数本质上与表长无关。

### 分程序结构语言的符号表的组织（下推链）

#### 分表结构

为每个分程序建立一张符号表。

#### 单表结构

设一个属性域登录符号所在层次。

#### 栈式符号表

遇变量声明，压入堆栈；

到达分程序结尾，弹出。

栈式符号表无序，查询效率低。

## 符号存储

### 类别

静态存储、寄存器存储

外部变量（公共存储变量）、内部变量（私有存储变量）

### 存储区

静态存储区

动态存储区

# 目标程序运行时的存储组织

## 分配方式

### 静态存储分配

编译时确定存储大小和位置。

### 动态存储分配

允许递归、可变数组、自由申请释放空间。

#### 栈式动态存储分配

##### 活动记录（AR：Activation Record）

存放过程的一次执行所需信息。

1. 临时工作单元：计算表达式临时存放中间结果
2. 局部变量
3. 机器状态信息：PC、寄存器
4. 存取链（非必需）：非局部变量
5. 控制链（非必需）：指向调用该过程的那个过程的活动记录
6. 实参
7. 返回地址

##### 嵌套过程

* 增设存取链，指向包含该过程直接外层过程的最新活动记录的起始位置。
* display栈
  + 指针数组，自顶向下存放现行层、直接外层，直至最外层0层的过程的最新活动记录的地址。

#### 堆式动态存储分配

new、delete

# 代码优化

## 分类

### 按对象

* 中间代码优化
* 目标代码优化

### 按范围

* 局部优化
* 循环优化
* 全局优化

## 优化类型

### 删除多余运算

删除公共子表达式

### 代码外提

循环不变运算

### 强度削弱

乘法变加法等

### 变换循环控制条件

通过改变循环控制条件，减少使用的变量

### 合并已知量与复写传播

例如：T4=T1，T5=T4，则删除T4，直接T5=T1

### 删除无用赋值

对T4赋值但未引用

## 局部优化

### 基本块的划分

#### 入口语句

* 程序的第一个语句
* 条件转移语句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

#### 基本快

1. 求各个入口语句
2. 对每个入口语句，构造所属的基本块。  
   由该入口到下一入口，或到一转移语句、停语句之间的语句序列
3. 未被纳入某一基本块的语句，是不可到达的，可删

### 基本块的变换

#### 保结构变换

* 删除公共子表达式
* 删除无用代码
* 重新命名临时变量
* 交换语句次序

#### 代数变换

* 简化表达式
* 较快运算替代较慢运算

### 基本块的有向无环图（DAG）表示

节点下部是值，右部是附加节点，节点里面是节点编号。

#### 0型

#### 1型

#### 2型

#### 构造0、1、2型四元式的DAG

* 若NODE(B)无定义，则构造NODE(B)
* 若是0型
  1. 令NODE(B)=n
* 若是1型
  1. 若NODE(B)是常数叶节点
     1. 执行（合并已知量），令得到的新常数为P
     2. 若NODE(B)是处理当前四元式才创建的，则删除它
     3. 若NODE(P)无定义
        1. 构造一个用P做标记的叶节点n
        2. 置NODE(P)=n
  2. 否则，不是常数叶节点
     1. 检查是否有一节点，其唯一后继为NODE(B)，且标记为（即公共子表达式）
     2. 若没有，构造该节点n
     3. 若有，把已有的节点作为它的节点并设该节点为n
* 若是2型
  1. 若NODE(C)无定义
     1. 构造标记为C的叶节点，定义NODE(C)为该节点
  2. 若NODE(B)和NODE(C)都是标记为常数的叶节点
     1. 执行（合并已知量），令得到的新常数为P
     2. 若NODE(B)或NODE(C)是处理当前四元式才创建的，删除
     3. 若NODE(P)无定义
        1. 构造一个用P做标记的叶节点n
        2. NODE(P)=n
  3. 否则，其中之一不是常数叶节点
     1. 检查是否有一节点，其左后继为NODE(B)，右后继为NODE(C)，且标记为（公共子表达式）
     2. 若没有，构造该节点n
     3. 若有，把已有节点作为它的节点并设该节点为n

最后统一：

* 若NODE(A)无定义
  + 1. A附加在节点n上
    2. 令NODE(A)=n
* 否则，若NODE(A)有定义
  + 1. 把A从NODE(A)节点上的附加标识符集中删除（若NODE(A)是叶节点则不删除）
    2. 把A附加到新节点n上
    3. 令NODE(A)=n

## 控制流分析和循环优化

### 程序流（程）图（流图）

节点即基本块。

有向边，建立边：

或

#### 例

1. read x
2. read y
3. r := x mod y
4. if r = 0 goto (8)
5. x := y
6. y := r
7. goto(3)
8. write(y)
9. halt
10. read x
11. read y
12. r := x mod y
13. if r = 0 goto (8)
14. x := y
15. y := r
16. goto (3)
17. write y
18. halt

### 循环的查找

必经节点：若从流图首节点出发，到达n的任意通路，都经过m，则m是n的必经节点。

记为m DOM n。因此，有a DOM a。

节点n的所有必经节点的集合，为n的必经节点集，记为D(n)。

回边：是一条有向边，若b DOM a，则是回边。

循环：是回边，则节点a、b，有通路到达a而不经过b的所有节点组成。

### 循环的优化

#### 代码外提

**循环所有出口节点的必经节点**中的**循环不变运算**前置外提。

#### 强度削弱与删除归纳变量