第1章 引论

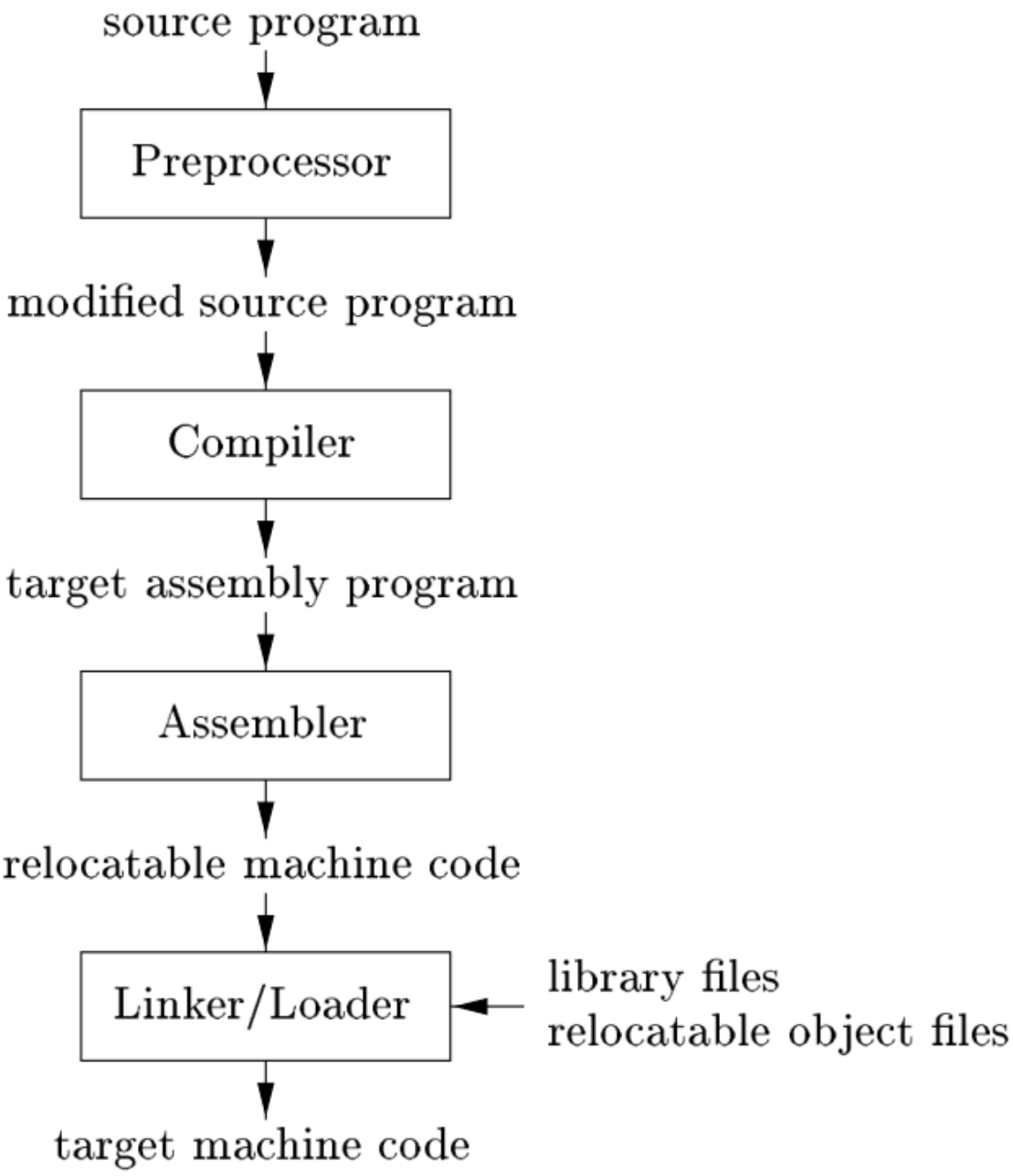
概念

解释器：

- 解释器直接利用用户提供的输入执行源程序中指定的操作。
- 解释器的错误诊断通常比编译器更好，因为它逐语句执行源程序。

JIT(just in time，及时)java编译器：在处理输入的前一刻将字节码翻译为本机的机器语言。

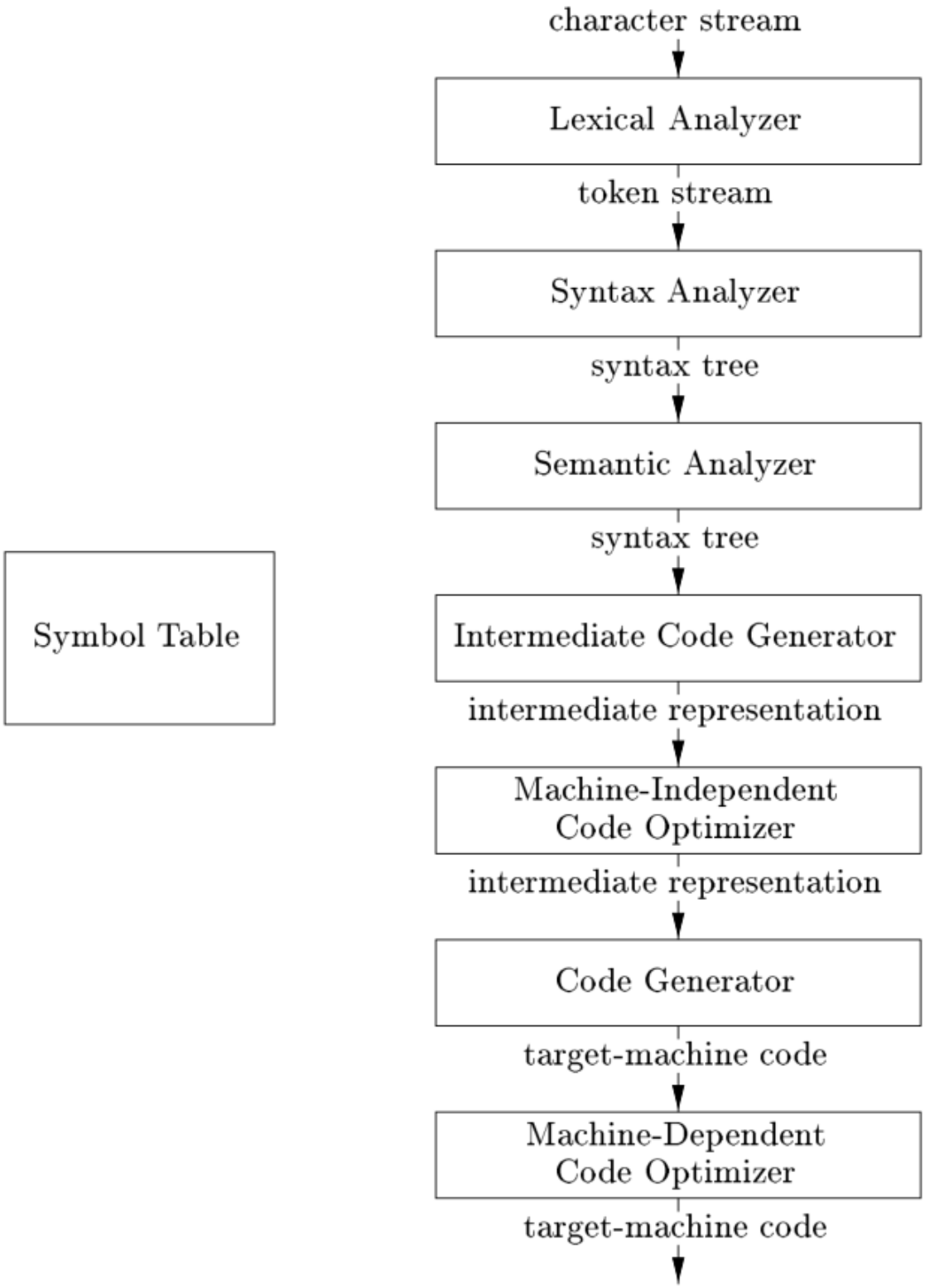
静态策略：这个问题在编译时刻决定的 动态策略：在运行时期做出决定的策略

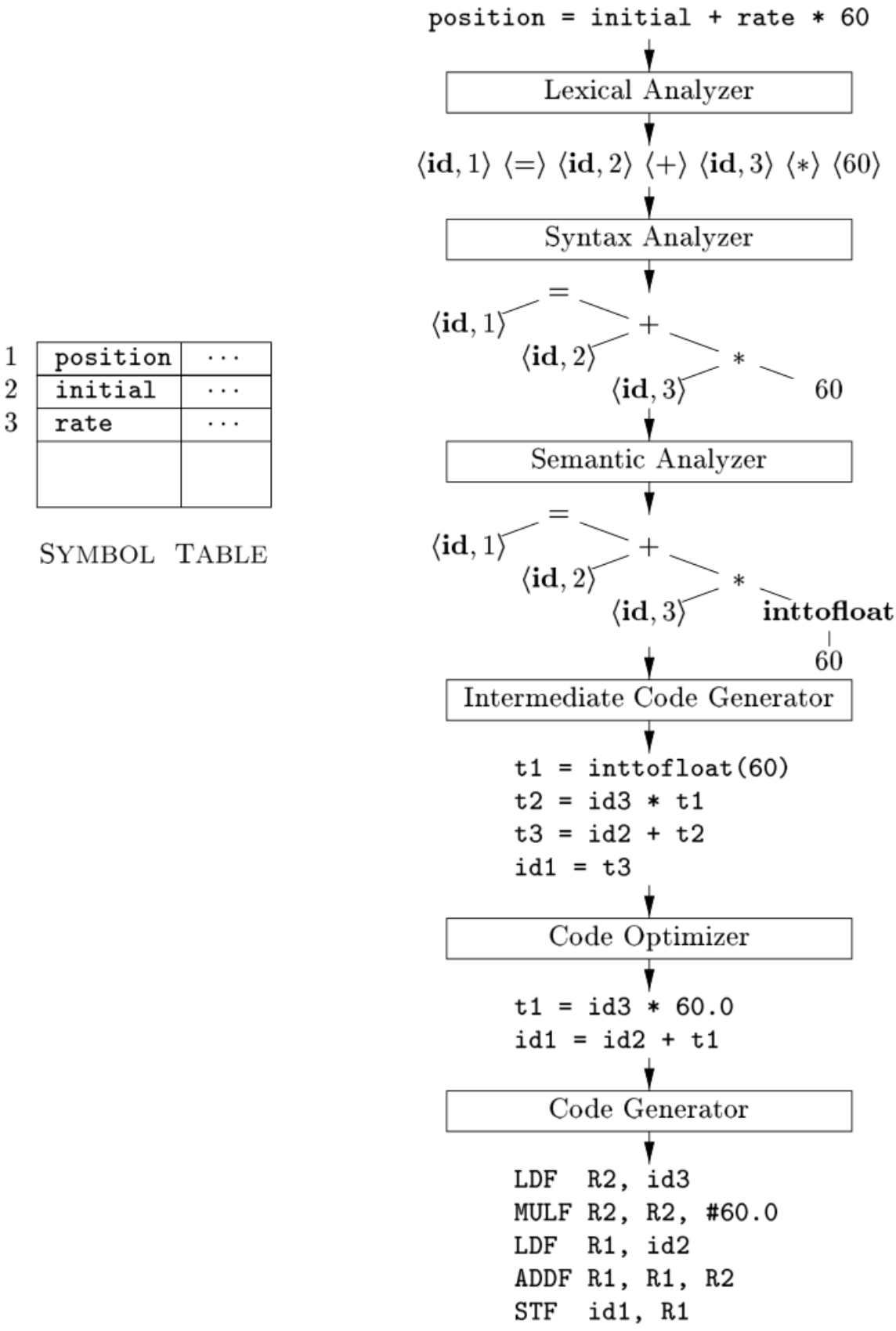


编译器的结构

编译器：一个语言到另一个语言的翻译 分析(analysis)：把源程序分解成多个组成要素，并在这些要素上面加上语法结构。 综合(synthesis)：根据中间表示和符号表中的信息来构建用户期待的目标程序。

编译器的前端(front end)：词法分析，语法分析，语义分析，中间代码生成 编译器的后端(back end)：中间代码优化，目标代码生成，目标代码优化





词法分析(lexical analysis)

也称扫描(scan)，将字符流组织成有意义的词素(lexeme)的序列。每个词素的输出形式是词法单元(token)，形式为<token-name, attribute-value>

语法分析(synatax analysis)

也成为解析(parsing)，将token转为树形的中间的语法表示，常为语法树(syntax tree)

语义分析(semantic analysis)

使用语法树和符号中的信息来检查源程序是否和语言定义的语义一致。例如类型检查(type checking)

中间代码生成(Intermediate Code Generator)

生成类机械语言的中间表示（与机器无关语言）

与机器无关的代码优化(Machine-Independent Code Optimizer)

代码生成(Code Generator)

目标机器代码优化

编译器优化

原则：

- 正确：不改变原含义
- 改善性能
- 时间必须在合理的时间内
- 所需要的工程方面的工作（设计和维护）必须是可管理的

第2章 一个简单的语法制导(syntax-directed)翻译器

简介：这一章是对整个编译器前端的简单介绍，我将这一章各部分的笔记放在相应的部分。

第3章 词法分析

语言(language)：某个给定的字母表上一个任意的可数的字符串集合。

字符串的运算定义：P75 语言上的运算：P75

正则表达式

字母表用Σ表示 正则表达式r表示的语言记为L(r)，也称为正则集合(regular set)，或正则语言 正则表达式的运算（并、连接、闭包）对应的语言运算：P76

正则表达式的等价：表示的语言一样

正则定义(regular definition)：P77，给某些正则表达式命名，类似上下文无关文法

状态转化图(transition diagram)

中文课本P82

有穷自动机(finite automata)

特点：对每个输入只能简单回答“是”或“不是” 分类：

- 不确定有穷自动机(Nondeterministic Finite Automata, NFA)：一个标号（输入符号）离开同一状态可能有多条边，空串也可以作为标号

- 确定的有穷自动机(Deterministic Finite Automata, DFA)

NFA

中文课本：P93 定义的语言：从开始状态到某个接受状态所有路径上的标号串的集合

DFA

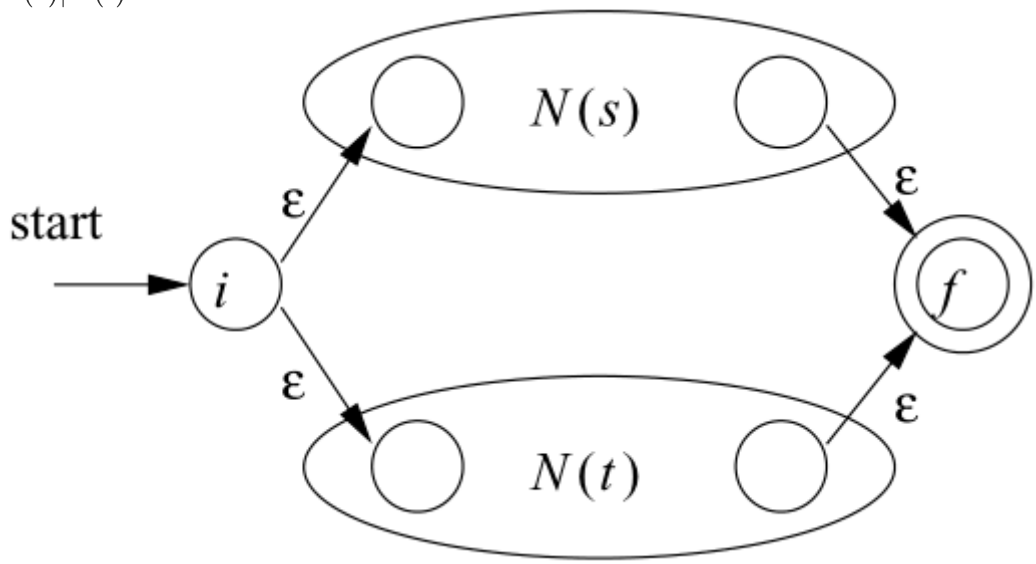
中文课本P95

转化

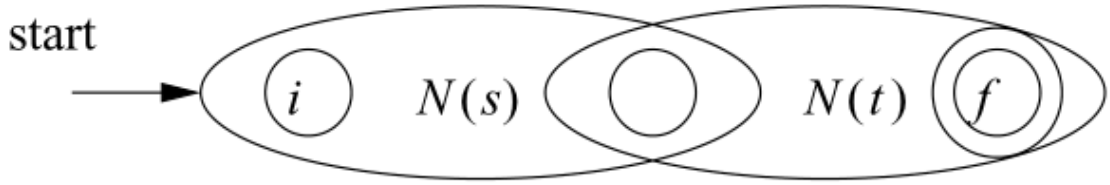
正则表达式到NFA

方法：McMaughton-Yamada-Thompson算法 算法步骤：P101

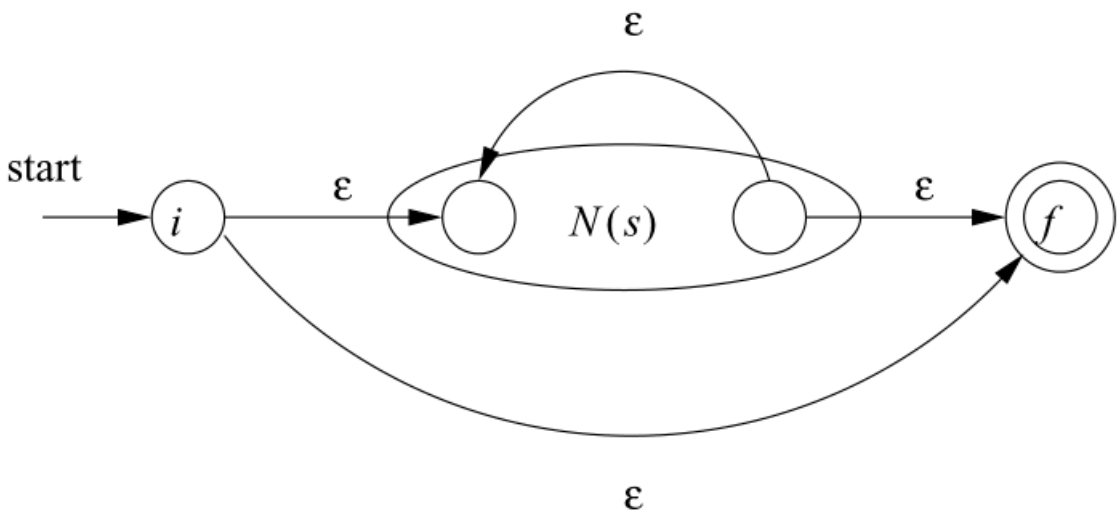
- $N(s)|N(t)$



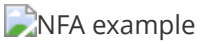
- $N(s)N(t)$



- $N(s)^*$



例子： 正则表达式：$(a|b)*abb$

 NFA example

NFA到DFA

方法：子集构造(subset construction)算法 算法步骤：P97

形式：

1. 画Dtran表
2. 画DFA

例子： Transition table Dtran for DFA D：  Transition table Dtran for DFA D
 Result of applying the subset construction

DFA到min-DFA

任何正则语言都有一个唯一的状态最少的DFA，记为min-DFA 可区分的：存在某一个输入串，使得状态s和状态t只有一个到达可接受状态。 工作原理：将一个DFA划分为多个组，每个组中的各个状态不可区分 具体：P115

例子：{A, C}, {B}, {D}, {C}

正则表达式到DFA

正则表达式 -> 抽象语法树 -> DFA 正则表达式 -> 抽象语法树：P110 抽象语法树 -> DFA：P113（有nullable, firstpos, lastpos, followpos等概念）

第4章 语法分析

引入

语法分析：接受一个终结符号串的输入，找出从文法的开始符号推到出这个串的方法。

语法分析器分类：

- 通用型
- 自顶向下
- 自底向上

文法：给出一个程序设计语言的精确易懂的语法规范 中文课本“文法”一词对应的英文：

- syntax of language
- grammar

句型(sentential form)：如果$S \Longrightarrow {}^*\alpha$，其中S是G的开始符号，那么$\alpha$就是G的一个句型，句型既可能包含终结符号又可能包含非终结符号。

- 如果α是最右推导来的，α也被称为最右句型
- 如果α是最左推导来的，α也被称为最左句型

句子(sentence)：不含非终结符号的句型

文法定义的语言：可以从开始符号推到得到的所有终结符号串（也就是句子）的集合。 文法的分类（按文法可以按哪种形式来翻译（处理）的分类）：

- LL文法：适合使用自顶向下，第一个L表示从左到右边扫描字符串，第二个L表示最左推导，LL(k)表示向前看k个输入符号
- LR文法：适合使用自底向上，L表示从左到右边扫描字符串，R表示反向构造最右推导

语法与语义的区别： 语法(syntax)：语言的正确形式 语义(semantics)：定义了程序的含义，即程序在运行时做什么事情。

各种树：

1. 语法分析树(syntax parse tree)：
    - 中文课本P28
    - 可以视为推导的图形表示形式（忽略了推导过程中对非终结符号应用产生式的顺序）。
    - 也称为具体语法树(concrete syntax tree)，其对应的文法称为该语言的具体语法(concrete syntax)
2. （抽象(abstract)）语法树：P43，内部节点表示的是程序构造，而语法分析树内部节点表示的是非终结符号
3. 注释(annotated)（语法）分析树：P196，语法分析树的节点加上属性值

上下文无关文法(context-free grammar, CFG)

简介

定义：

- 终结符号(terminal)集合
- 非终结符号(nonterminal)集合
- 产生式(production)集合
- 指定一个非终结符号为开始符号（一般是第一个产生式的头）

缺陷：

- 无法表达程序中那种先声明后使用的文法

推导

定义：从开始符号开始，不断将某个非终结符号替换为非终结符号的某个产生体

分类：

- 最左推导(leftmost derivation)：总是选择最左边的非终结符号进行替换， $\Longrightarrow_{lm}$
- 最右推导(rightmost derivation)：有时称为规范推导(canonical derivation)

具体内容：中文课本P126

二义性

二义性定义（如何判断）：一个终结符号串是多个语法分析树的结果，也就是说存在多个最左推导或最右推导。 如何消除二义性：P133

结合性

左结合的文法：P27

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

右结合的文法：P30

```
right → letter = right | letter
letter → a | b | ... | z
```

上述第一个式子表明在right展开，所以改成左结合的话，如下

```
left → left = letter | letter
letter → a | b | ... | z
```

优先级

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

推广到n层优先级，需要n+1个非终极符号，中文课本P31

CFG 与 RE

语言可以用RE描述，肯定可以用CFG描述，反之不一定成立 RE转CFG：P130, RE -> NFA -> CFG

既然CFG比RE强大，为什么要用RE来表示词法分析器：P132-P133

语法分析

任何上下无关文法，都可以构造出一个时间复杂度为$O(n^3)$的语法分析器

自顶向上和自底向上语法分析器的构造都可以使用和文法G相关的两个函数FIRST和FOLLOW来实现。$FIRST(\alpha)$：α可以推导得出的串的首个字符的集合 $FOLLOW(A)$：紧跟在A右边的终结符号的集合，记$为结束标记，$FOLLOW$集合中不包含$\epsilon$。 如何计算上面两个函数：P140-141

自顶向下(top-down)

定义：输入串构造语法树的最左推导的过程。从根节点开始，逐步向叶子节点方向进行 向前看(lookahead)符号：输入中当前被扫描的终结符号

递归下降分析法(recurise-descent parsing)

递归下降分析法：一种自顶向下的分析方法 递归下降分析法不能处理左递归(left recursive)的文法，如$A \to A\alpha$，因为左递归会让递归下降分析进入无限循环，即使该分析器带了回溯。

左递归

左递归的分类

- 立即左递归，如$A \to A\alpha$
- 非立即左递归，如$S \to Aa|b, A \to Ac|Sd|\epsilon$，可推导出$S \implies Aa \implies Sda$

消除左递归（只适用于不存在环的推导，即$A \implies A$，或$A \implies \epsilon$）：P135

提取公因子

中文课本：P135

预测分析法（LL(1)语法分析器）

预测分析法(predictive parsing)：

- 一种简单的递归下降分析法
- 非终结符号后可以由向前看符号无二义确定，即$A \to \alpha, A \to \beta, c$，换句话说，它只需要检查当前输入符号就可以为一个非终结符号选择正确的产生式，不需要进行回溯来寻找真正匹配的产生式。
- 作用：用以构造LL(1)文法的语法分析器

LL(1)的严格判断：$A \to \alpha|\beta$

- $FIRST(\alpha) \bigcap FIRST(\beta) = \emptyset$
- $\exists \epsilon \in \alpha, FOLLOW(A) \bigcap FIRST(\beta) = \emptyset$
- $\exists \epsilon \in \beta, FOLLOW(A) \bigcap FIRST(\alpha) = \emptyset$

如何使用：

1. 文法存在左递归，则消除左递归
2. 构造预测分析表：表示输入哪些终结符号可以使用哪些产生式，需要用到FIRST，算法：P143
3. 使用预测分析表进行语法分析：如P144的图，栈顶初始为（上下文无关）文法的开始符号

错误恢复：P145

非递归预测分析法

内容：手动维护一个栈来模拟递归预测分析 P144

自底向上(bottom-up)

一个自底向上的语法分析过程对应一个输入串构建语法分析树的过程，它从叶子底部开始逐渐向上到达根部 目标（本质）：反向构造一个最右推导过程

句柄(handle)：P149，和某产生式体匹配的字串，对它的规约代表了相应的最右推导中的一个反向步骤。句柄的右边肯定是终结符号，不然不满足反向的最右推导

可行前缀(viable prefix)：P163，移入-归约语法分析器的栈中最右句型前缀(right sentential form)，它没有越过最右句柄的右端，也就是后面的LR(0)等算法在分析时符号栈的符号。

移入-归约 分析方法(shift-reduce parsing)

一种自底向上的分析方法

归约：一此归约是一次推导的反向操作。

LR(0)

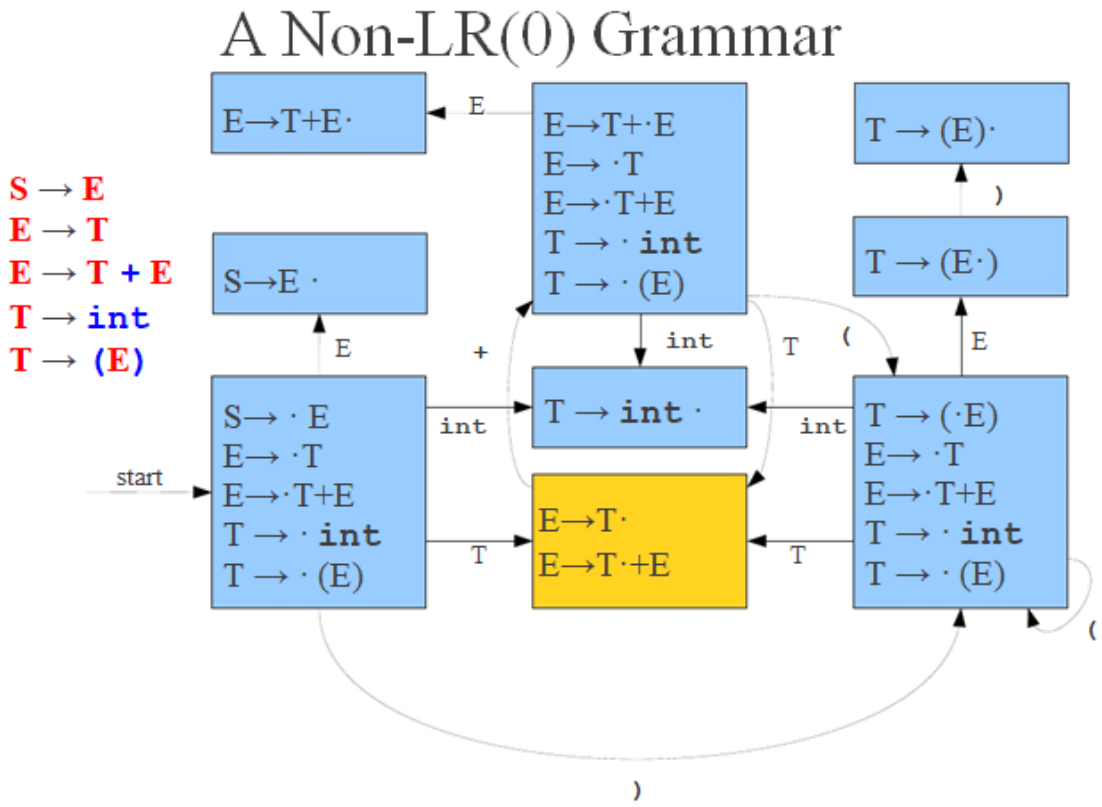
如何使用LR(0)进行语法分析：

1. 构造文法G的增广文法：开始符号为S，加入产生式$S \to S'$
2. 构造LR(0)自动机（该自动机属于DFA）
    - GOTO：P156
    - CLOUSRE：P156 图4-32
    - 构建：P155 图4-31，accept只出现在开始符号归约后
3. 构造分析表：如P160 图4-37
4. 进行分析：如P160 图4-38，另一种做法是将栈的数字放在相应的符号列的符号后，以便归约弹栈方便

LR(0)自动机中一些概念：

- （LR(0)）项(iterm)的定义：P154
- 项集：由项组成的集合，记为I
- 项集族：由项集组成的集合

LR(0)的归约冲突—例子



黄色方框输入T进行归约后又回到黄色方框

若对一个文法符号，项集里面有两个产生式要进行归约的情况，LR(0)也不能处理

SLR(1)（常记为SLR）

S表示Simple

与LR(0)区别：

- 利用向前看符号(lookahead)来避免移入/归约冲突
- SLR(1)自动机跟LR(0)自动机一样，不同的是当我们选择归约的时候
  - 假设$A-\rightarrow \alpha\cdot$在状态集$I_i$
  - LR(0)：输入符号a不能进行移入都进行归约
  - SLR(1)：输入符号a必须在FOLLOW(A)才进行归约

规范LR（LR方法，LR(1)）

LR(1)的强大

- Any LR(0) grammar is LR(1).
- Any LL(1) grammar is LR(1).
- Any deterministic CFL (a CFL parseable by a *deterministic pushdown automaton*) has an LR(1) grammar.
- Any LL(*k*) *language* is LR(1), though individual LL(*k*) *grammars* might not be.
- Any LR(*k*) *language* is LR(1), though individual LR(*k*) *grammars* might not be.

步骤：

1. 构造自动机：类似P169 图4-41，算法在P167
2. 构造分析表：类似P170 图4-42

LALR(look ahead LR)

太难了，不考

其他

LR语法分析器的优点：P153

语法错误的恢复 P124 P145

- 恐慌模式(panic mode)
- 短语层次(phrase-level recovery)
- 错误产生式(Error Pro ductions)
- 全局纠正(Global Correction)

第5章 语法制导的翻译(语义分析)

语法制导定义

语法制导定义（Syntax-Directed Definition, SDD）：一个上下文无关文法和属性及规则的结合。

- 将每个文法符和一个属性集合相关联
- 每个产生式与一组语义规则相关联

属性分类 综合属性(synthesized attribute)：属性由子节点的属性确定 继承属性：属性由本身、父节点和兄弟节点的属性决定

两类重要的SSD：P200

- S属性(S-attributed)SSD：S代表综合
- L属性SSD：L代表从左到右

语法制导的翻译方案(syntax-directed tranlstation scheme, SDT)

定义：一个（上下文无关）文法在产生式体中嵌入了语义动作（程序片段）

SDT的实现：

- 构建一棵语法分析树
- 从左到右深度优先执行这些语义动作

两种SDT实现SDD的方案：

- 后缀翻译方案(Postfix Translation Schemes)：文法是LR的，SSD是S属性的，P207
- 文法是LL的，SSD是L属性的，P212

第6章 中间代码生成

本章内容：中间代码表示、静态类型检查、中间代码生成

中间代码的形式：

- 树形结构：语法分析树、抽象语法树
- 线性结构：三地址码，静态单赋值形式(SSA，P237)

静态检查

- 语法检查
- 类型检查

DAG(Directed Acyclic Graph)

有向无环图(DAG)，P230 值编码(value number)：DAG的节点的存放与查找

三地址码

形式：P233

三地址码的数据结构

- 四元式(quadruple)：P235
- 三元式(triple)：P235
- 间接三元式(indirect triple)：P236

翻译到三地址码

流控制

1. if-else

```
if ( c ) {
    stmt1;
} else{
    stmt2;
}
stmt3;
```

```
if False c goto L1:
stmt1
goto L2;
L1:
stmt2(marked after)
L2:
stmt3;
```

2. while

```
while(c) {
    stmt1;
}
stmt2;
```

```
L1:
if False c goto L2;
stmt1;
goto L1;
L2:
stmt2;
```

3. do-while

```
do{
    stmt1;
}while(c);
stmt2;
```

```
L1:
stmt1;
if True c goto L1;
stmt2;
```

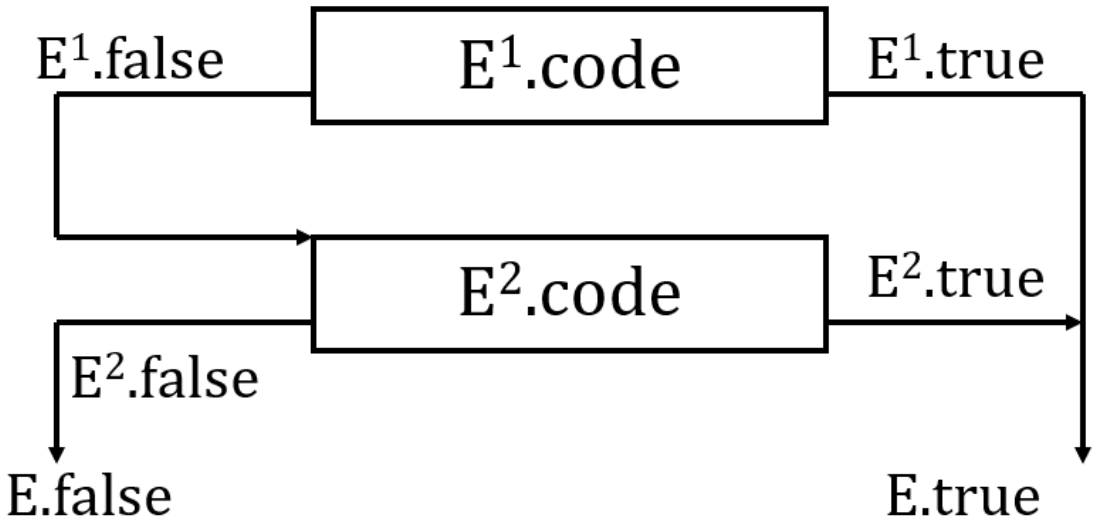
布尔

例子：a || b && !c（注意优先级）

```
t1 = !c
t2 = b && t1
t3 = a || t2
```

复合布尔的流控制

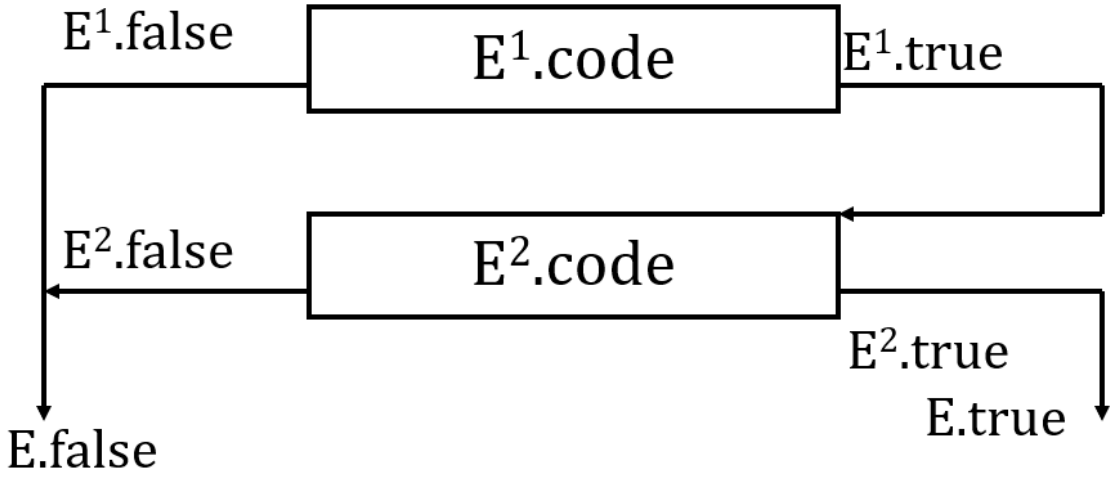
1. $E = E_1 || E_2$




```
if(E1 || E2){
    stmt1;
}
stmt2;
```

```
if True E1 goto E.true;
if True E2 goto E.true;
goto E.false;
E.ture:
stmt1;
E.false:
stmt2;
```

2. $E = E_1 \&\& E_2$




```
if(E1 && E2){
    stmt1;
}
stmt2;
```

```
if false E1 goto E.false;
if false E2 goto E.false;
stmt1;
E.false:
stmt2;
```