

1. 编程语言的发展

1.1 语言的发展阶段

机器语言 第一台电子计算机 ENIAC 出现于 1946 年。它通过设置开关和线路，使用机器语言（0 和 1 的序列）进行编程。

汇编语言 使用助记符来代表机器指令。使用宏指令来代表频繁使用的机器指令序列。显式地操作内存地址和内容。

高级语言 人类可以理解，需要“翻译器”才能被计算机理解。

1.2 典型高级语言的特征

Fortran: 用于科学计算；Cobol: 用于商业数据处理；Lisp: 用于符号计算。

2. 编译器的结构

2.1 前端与后端的划分

前端 将源程序分解为组成部分，并在其上施加语法结构。利用语法结构创建源程序的中间表示（IR）。收集关于源程序的信息并将其存储在称为符号表的数据结构中。

组成部分：词法分析器 | 语法分析器 | 语义分析器 | 中间代码生成器 后端根据 IR 和符号表中的信息构建目标程序（通常是机器语言）。在此过程中执行代码优化。

组成部分：机器无关代码优化器 | 代码生成器 | 机器相关代码优化器

2.2 符号表管理

由前端维护，符号表随中间代码一起传递给后端。记录变量名称和各种属性（分配的存储空间、类型、作用域）。记录过程名称和各种属性。

3. 编译器 V.S. 解释器

执行方式：编译器将高级语言转换为目标计算机上的机器码；解释器执行每一条语句，不需要转换成机器码。

性能与分析：编译器分析语句关系流并进行优化；解释器在分析、解析和执行上花费的时间较少（但整体语言执行较慢，因为没有优化）。

错误反馈：编译器在成功编译后才能执行；解释器执行直到遇到第一个错误时停止。

1. 核心概念 (Core Concepts)

1.1 基本术语

词素 (Lexeme): 源程序中的字符序列，是编程语言中最低级的语法单位 (实例). 例如: if, count, 123, >.

词法单元 (Token): 代表一类词素的语法类别 (抽象分类)。形式: (*token* - *name*, *attribute* - *value*). 其中 token-name 是语法分析使用的抽象符号; attribute-value (可选) 通常指向符号表中的条目，用于语义分析和代码生成。

模式 (Pattern): 描述某一类 Token 的词素可能采取的形式。通常使用正则表达式来描述。

串 (String): 基于某个固定字母表的符号的有穷序列。长度: 串中符号的数量。记作 |*s*|. 空串: 长度为 0 的串，记作 ϵ 。

语言 (Language): 基于某个固定字母表的串的可数集合。{ ϵ } 是包含空串的语言，不同于空集 \emptyset 。

1.2 语言的运算 (Operations on Languages)

假设 *L* 和 *M* 是两个语言:

并 (Union): $L \cup M = \{s \mid s \in L \text{ or } s \in M\}$ 。

连接 (Concatenation): $LM = \{st \mid s \in L \text{ and } t \in M\}$ 。

Kleene 闭包 (Kleene Closure): $L^* = \cup_{i=0}^{\infty} L^i$ 。表示 *L* 连接 0 次或多次（包含空串 ϵ ）。

正闭包 (Positive Closure): $L^+ = \cup_{i=1}^{\infty} L^i$ 。表示 *L* 连接 1 次或多次。

2. 正则表达式 (Regular Expressions)

2.1 定义与规则

正则表达式是用于指定 Token 模式的重要符号表示法。定义在字母表 Σ 上的规则如下:

基础 (Basis): ϵ 是正则表达式，表示语言 { ϵ }。如果 *a* 是 Σ 中的符号，则 *a* 是正则表达式，表示语言 {*a*}。

归纳 (Induction):

并: $r|s$ 表示语言 $L(r) \cup L(s)$ 。

连接: rs 表示语言 $L(r)L(s)$ 。

闭包: r^* 表示语言 $(L(r))^*$ 。

括号: (r) 表示语言 $L(r)$ 。

2.2 优先级 (Precedence)

为了避免过多的括号，规定运算优先级从高到低为：一元运算符: 闭包 * (Closure) 连接: (Concatenation) 并: | (Union) 所有运算都是左结合的。

2.3 正则定义 (Regular Definitions)

为了方便，给某些正则表达式命名并在后续定义中重用。形式: $d_i \rightarrow r_i$ 例子 (C 语言标识符): $letter_non \rightarrow A|B|...|Z|a|b|...|z|_ digit \rightarrow 0|1|...|9 \ id \rightarrow letter_non(letter_non digit)^*$ 。

3. 有穷自动机 (Finite Automata)

3.1 NFA (非确定有穷自动机)

定义: 一个五元组 $(S, \Sigma, \delta, s_0, F)$ 。S: 有穷状态集， Σ : 输入符号集。 δ : 转换函数，给出给定状态和输入符号（或 ϵ ）的下一状态集合。 s_0 : 开始状态。 F : 接受状态集。

特点: 同一个符号可以标记离开同一状态的多条边。允许空串 (ϵ) 转换。接受: 只要存在一条从开始状态到接受状态的路径，使得路径上的符号组成输入串，该串即被接受。

3.2 DFA (确定有穷自动机)

特点: 没有 ϵ 转换。对于每个状态 *s* 和每个输入符号 *a*，恰好有且仅有一条标号为 *a* 的边离开 *s*。

优势: 模拟 DFA 识别模式非常高效（无需回溯或并行搜索）。

4. 关键算法 (Key Algorithms)

4.1 Thompson 构造法 (正则表达式 \rightarrow NFA)

一种基于语法制导的算法，将正则表达式递归地转换为 NFA。

输入: 字母表 Σ 上的正则表达式 *r*。

输出: 接受 *L*(*r*) 的 NFA。

构造规则:

基础: 为 ϵ 和符号 *a* 构造简单的 NFA。

并 ($|$): 引入新开始状态和新接受状态。新开始状态通过 ϵ 转换分别连接到 *N*(*s*) 和 *N*(*t*) 的开始状态; *N*(*s*) 和 *N*(*t*) 的结束状态通过 ϵ 转换连接到新接受状态。

连接 (\cdot): 将 *N*(*s*) 的接受状态与 *N*(*t*) 的开始状态合并 (或用 ϵ 连接)。

闭包 (s^*): 引入新开始状态和新接受状态。新开始 $\xrightarrow{\epsilon}$ 旧开始旧接受 $\xrightarrow{\epsilon}$ 新接受旧接受 $\xrightarrow{\epsilon}$ 旧开始 (循环) 新开始 $\xrightarrow{\epsilon}$ 新接受 (匹配 0 次)。

4.2 子集构造法 (NFA \rightarrow DFA)

将 NFA 转换为等价的 DFA。核心思想是 DFA 的每个状态对应 NFA 的一个状态集合。

核心概念: ϵ -closure(*T*): 从集合 *T* 中的状态出发，仅通过 ϵ 转换可达到的所有 NFA 状态集合。move(*T*, *a*): 从集合 *T* 中的状态出发，通过输入符号 *a* 可达到的所有 NFA 状态集合。

算法流程: 初始状态: DFA 的初始状态 $A = \epsilon$ -closure($\{s_0\}$)。将 *A* 标记为未处理。While 存在未标记的 DFA 状态 *T*: 标记 *T*。For 每个输入符号 *a*: 计算 $U = \epsilon$ -closure(move(*T*, *a*))。如果 *U* 不在 DFA 状态集

中，将 *U* 加入并标记为未处理。添加 DFA 转换 $T \xrightarrow{a} U$ 。

合并 NFA: 为了识别多个模式（如 if, id, number），将每个模式的正则表达式转换为 NFA，然后引入一个新的开始状态，通过 ϵ 转换连接到各个 NFA 的开始状态。

冲突解决: 当转换后的 DFA 的一个接受状态包含多个 NFA 的接受状态时（即输入前缀匹配多个模式）:

优先顺序: 选择定义在前的模式。

最长匹配: 通常词法分析器会匹配尽可能长的词素。

1. 上下文无关文法 (Context-Free Grammars, CFG)

核心概念: 描述编程语言语法的形式化方法，比正则表达式表达能力更强。

1.1 基本定义

CFG 的组成 ($G = (T, N, P, S)$):

终结符 (Terminals): 组成串的基本符号 (Token)。

非终结符 (Nonterminals): 表示串集合的语法变量。

开始符号 (Start Symbol): 一个特殊的非终结符。

产生式 (Productions): 形式为 $head \rightarrow body$ ，其中 *head* 是非终结符，*body* 是终结符和非终结符的序列。

1.2 推导 (Derivation)

定义: 从开始符号出发，反复使用产生式将非终结符替换为产生式体，直到生成仅包含终结符的串。

最左推导 (Leftmost Derivation): 每一步都替换句型中最左边的非终结符（对应自顶向下分析）。

最右推导 (Rightmost Derivation): 每一步都替换句型中最右边的非终结符（对应自底向上分析的逆过程，规范推导）。

句型 (Sentential Form): 推导过程中出现的符号序列（包含终结符或非终结符）。

句子 (Sentence): 不包含非终结符的句型。

1.3 语法分析树 (Parse Tree) 与二义性 (Ambiguity)

语法分析树: 推导的图形化表示，根是开始符号，叶子是终结符，内部节点是非终结符。

二义性: 如果某个文法的某个句子存在多于一棵语法分析树（或多于一种最左推导），则称该文法是二义性的。

消除二义性: 重新设计文法（如规定优先级和结合性）或使用消歧规则（如 else 匹配最近的 if）。

2. 自顶向下分析 (Top-Down Parsing)

核心理由: 从语法的根节点开始，通过输入串构造语法树（寻找最左推导）。

2.1 递归下降分析 (Recursive-Descent Parsing)

原理: 为每个非终结符编写一个递归过程。

回溯 (Backtracking): 如果产生式选择错误，需要回退并尝试其他产生式。通过递归下降可能需要回溯，效率较低。

2.2 LL(1) 文法与预测分析

LL(1) 含义: 从左向右扫描输入 (*L*)，产生最左推导 (*L*)，向前看 1 个符号 (*1*)。

(1) 关键集合构造算法 (FIRST & FOLLOW) 这是构造预测分析表的基础。

FIRST(*X*): 可从文法符号 *X* 推导出的串的首个终结符集合。若 *X* 是终结符， $FIRST(X) = \{X\}$ 。若 $X \rightarrow \epsilon$ ，则 $\epsilon \in FIRST(X)$ 。若 $X \rightarrow Y_1 Y_2 \dots Y_k$ ，若 $FIRST(Y_1) \neq \epsilon$ ，加入 $FIRST(X)$ 。若 $FIRST(Y_1)$ 包含 ϵ ，则继续查看 Y_2 ，依此类推。若所有 Y_i 都包含 ϵ ，则 $\epsilon \in FIRST(X)$ 。

FOLLOW(*A*): 在某些句型中紧跟在非终结符 *A* 之后的终结符集合。将 \$ (输入结束标记) 加入 FOLLOW(*S*)。若有 $A \rightarrow \alpha \beta B$ ，将 $FIRST(\beta) \setminus \{\epsilon\}$ 加入 FOLLOW(*B*)。若有 $A \rightarrow \alpha \beta$ 或 $A \rightarrow \alpha \beta B$ 且 $\epsilon \in FIRST(\beta)$ ，将 FOLLOW(*A*) 加入 FOLLOW(*B*)。

(2) LL(1) 文法的条件对于同一非终结符 *A* 的任意两个产生式 $A \rightarrow \alpha | \beta$:

不相交: $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$ 。

无冲突: 若 $\epsilon \in FIRST(\beta)$ ，则 $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$ 。

(3) 预测分析表 (Predictive Parsing Table) 构造对于产生式 $A \rightarrow \alpha$: 对于 $\forall a \in FIRST(\alpha)$, $M[A, a] = A \rightarrow \alpha$ 。若 $\epsilon \in FIRST(\alpha)$ ，对于 $\forall b \in FOLLOW(A)$, $M[A, b] = A \rightarrow \alpha$ 。

3. 自底向上分析 (Bottom-Up Parsing)

核心理由: 从叶子节点（输入串）开始，归约为根节点（开始符号）| 寻找最右推导的逆过程 |

3.1 移入-归约分析 (Shift-Reduce Parsing)

栈操作: 使用一个栈来保存文法符号。

四个动作:

移入 (Shift): 将输入符号压入栈。

归约 (Reduce): 将栈顶的句柄 (Handle) 替换为产生式左部的非终结符。

接受 (Accept): 分析成功。

报错 (Error): 发现语法错误。

3.2 LR 分析技术

LR 分析器是表驱动的，由 Action 表和 Goto 表组成。核心是构建识别可行前缀的 DFA。

(1) LR(0) 项与项集闭包

LR(0) 项: 右部带有圆点的。的产生式 (如 $A \rightarrow X \cdot YZ$)，表示分析进度。

CLOSURE(*I*) 算法: 初始将 *I* 中所有项加入。若 $A \rightarrow \alpha \cdot \beta B \in I$ ，则将所有 $B \rightarrow \gamma \cdot$ 加入 *I*。重复直到不再增加。

GOTO(*I*, *X*) 算法: *I* 中所有形如 $A \rightarrow \alpha \cdot X \beta$ 的项，移动点得到 $A \rightarrow \alpha X \cdot \beta$ ，对结果集合求闭包。

(2) 三种 LR 分析表的对比

• SLR (简单 LR): LR(0) 项集 + FOLLOW 集合。弱。当出现归约时，仅当输入符号 *a* $\in FOLLOW(A)$ 时才归约 *A*。无法解决很多移入-归约冲突。

• CLR (LR(1)) (规范 LR): LR(1) 项集 (带向前看符号)。最强。状态数非常多。利用由 CLOSURE 传播的精确向前看符号解决冲突。

• LALR (向前看 LR): 合并 CLR 中核心 (Core) 相同的状态。中等 (实际最常用)。状态数与 SLR 相同。可能产生归约-归约冲突，但不会产生移入-归约冲突。

(3) 冲突类型

移入-归约冲突 (Shift-Reduce Conflict): 既可以移入下一个符号，也可以按当前栈顶进行归约。

归约-归约冲突 (Reduce-Reduce Conflict): 栈顶字符串可以按两个不同的产生式进行归约。

4. 补充知识点 (Supplement)

4.1 文法变换

为了使文法适用于自顶向下分析 (LL(1))，通常需要:

消除左递归 (Eliminate Left Recursion): 直接左递归 $A \rightarrow A\alpha | \beta$ 替换为 $A \rightarrow \beta, A' \rightarrow \alpha A' | \epsilon$ 。

提取左公因子 (Left Factoring): $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ 替换为 $A \rightarrow \alpha\alpha', A' \rightarrow \beta_1 | \beta_2$ 。

4.2 LR(1) 项集构造细节

在 CLR 中，CLOSURE 算法有所不同: 若 $[A \rightarrow \alpha \cdot \beta \beta, a] \in I$ ，则对 $\forall b \in FIRST(\beta a)$ ，将 $[B \rightarrow \gamma \cdot b]$ 加入 *I*。这里的 *b* 是精确的向前看符号，来源于 β 和 *a*。

1. 核心概念 (Core Concepts)

1.1 语法制导定义 (Syntax-Directed Definitions, SDD)

定义: SDD 是一个上下文无关文法 (CFG) 加上属性 (attributes) 和语义规则 (semantic rules)。

特点: 它是一个高层次的规范 (High-level specification)，只定义了“做什么” (what to do)，而不指定“怎么做” (how to do)，即不指定求值顺序。

组成:

属性 (Attributes): 关联到每个文法符号上，用于存储信息 (如值、类型、代码片段等)。

语义规则 (Semantic Rules): 关联到每个产生式，描述如何计算属性值。

1.2 语法制导翻译方案 (Syntax-Directed Translation Schemes, SDT)

定义: SDT 是在产生式体部中嵌入语义动作 (semantic actions) 的上下文无关文法。

特点: 它不仅定义了语义，还指定了动作的执行时机 (实现细节)。语义动作是具体的程序代码片段，用花括号 {} 包围。

位置: 语义动作可以出现在产生式体部的任何位置。例如: $E \rightarrow E_1 + T \{ \text{print}('++') \};$ 。

1.3 属性的分类 (Classification of Attributes)

合成属性 (Synthesized Attributes):

定义: 节点 *N* 上的属性值仅由 *N* 的子节点或 *N* 本身的属性值决定。

特点: 信息从下往上流动 (自底向上)。

应用: 表达式求值、S-属性定义。

继承属性 (Inherited Attributes):

定义: 节点 *N* 上的属性值由 *N* 的父节点、*N* 的兄弟节点或 *N* 本身的属性值决定。

特点: 信息从上往下或从左往右流动。

应用: 类型声明 (将类型信息分发给变量列表)、上下文信息传递。

2.1 依赖图 (Dependency Graph)

作用: 描绘特定语法的分析树中属性实例之间的信息流。

构建: 如果属性 *b* 的计算依赖于属性 *c*，则画一条从 *c* 指向 *b* 的边。

原则: 如果依赖图中存在环 (cycle)，则无法找到求值顺序，该 SDD 是不可计算的。

2.2 求值顺序 (Evaluation Orders)

拓扑排序 (Topological Sort): 依赖图的拓扑排序给出了属性计算的有效顺序。在拓扑排序中，任何属性的计算都先于依赖它的属性。

3. 两类重要的 SDD (Classes of SDD)

3.1 S-属性定义 (S-Attributed SDD)

定义: 仅仅包含合成属性的 SDD。

求值顺序: 可以按照语法分析树节点的后序遍历 (Postorder Traversal) 顺序进行计算 (自底向上)。

实现: 非常适合在自底向上的分析 (如 LR 分析) 过程中实现。当发生归约 (Reduce) 时执行相应的语义动作。通常使用一个栈来保存属性值。

3.2 L-属性定义 (L-Attributed SDD)

定义: 允许包含合成属性和受限的继承属性。对于产生式 $A \rightarrow X_1 X_2 \dots X_n$ ，右部符号 X_j 的继承属性只能依赖于: 产生式体中 X_j 左边的符号 X_1, \dots, X_{j-1} 的属性。产生式头 *A* 的继承属性。(注: *L*

代表 Left-to-right，即信息从左向右流动)

求值顺序: 可以按照深度优先遍历 (Depth-First Traversal) 的顺序计算 (通常结合从左到右的扫描)。

实现: 适合在自顶向下的分析 (如 LL 分析) 中实现。

4. 实现与算法 (Implementation & Algorithms)

4.1 SDD 转换为 SDT 的规则 (Converting SDD to SDT)

如何将 L-属性 SDD 转换为可在解析过程中执行的 SDT?

基本原则: 将语义动作放置为属性计算所需的所有信息都可用的最早位置。

转换规则:

继承属性: 将计算非终结符 *X* 继承属性的动作，插入到产生式体部中 *X* 的紧前面。

合成属性: 将计算产生式头部合成属性的动作，放置在产生式体部的最末尾。

4.2 递归下降解析中的实现 (L-属性)

在递归下降分析器中实现 L-属性 SDD 的通用模板:

函数参数: 用于传递继承属性 (将信息传递给子节点)。

返回值: 用于返回合成属性 (将信息传回父节点)。

算法逻辑:

```
// 产生式 A -> X Y
ReturnType A(InheritedType a_inh) {
    // ... 匹配 X ...
    // 计算 X 的继承属性 x_inh (依赖 a_inh)
    x_syn = X(x_inh);
    // ... 匹配 Y ...
    // 计算 Y 的继承属性 y_inh (依赖 a_inh 和 x_syn)
    y_syn = Y(y_inh);
    // 计算 A 的合成属性 a_syn
    return a_syn;
}
```

4.3 自底向上解析中的实现 (S-属性/后缀 SDT)

后缀翻译方案 (Postfix Translation Schemes): 所有语义动作都位于产生式的右最末。

实现: 使用 LR 分析器。维护一个与分析栈平行的语义栈（或在同一个栈中存储符号和属性）。当执行归约 $A \rightarrow \beta$ 时，执行对应的语义动作。属性值通过栈索引访问 (例如 stack[top-1].val)。

5. 补充知识 (Supplement)

5.1 标记非终结符 (Marker Nonterminals)

问题: 在自底向上分析 (LR) 中，如果语义动作嵌入在产生式中间 (非后缀 SDT)，直接实现会破坏 LR 分析的逻辑 (因为归约发生在该产生式完全识别之后)。

解决方案: 引入标记非终结符 *M* 将其转换为后缀 SDT。将嵌入的动作替换为 *M*。添加产生式 $M \rightarrow \epsilon$ 。将原动作关联到 $M \rightarrow \epsilon$ 上。例子: $A \rightarrow a\{action\}b \Rightarrow A \rightarrow aMb, M \rightarrow \epsilon\{action\}$ 。

5.2 消除左递归时的属性处理

当为了适应 LL(1) 分析而消除左递归时 ($A \rightarrow A\alpha | \beta$ 转为 $A \rightarrow \beta A', A' \rightarrow \alpha A' | \epsilon$)，属性的计算逻辑也需要相应转换。通常涉及将原综合属性转换为新非终结符 *A'* 的继承属性进行传递。

5.3 常见应用场景

类型检查: L-属性。通过继承属性将类型环境传入，通过合成属性返回检查结果。

抽象语法树 (AST) 构建: S-属性。子节点返回子树指针，父节点创建新节点并连接子树。

中间代码生成: 往往混合使用。S-属性用于生成表达式代码，L-属性用于回填控制流标签。

1. 中间表示 (Intermediate Representation, IR)

1.1 概念与作用

定义: 编译器前端分析源程序后生成的、独立于具体机器的表示形式。

好处:

复用性: 使得 *M* 种语言和 *N* 种目标机器只需要 *M* 个前端和 *N* 个后端 (共 *M* + *N* 个编译器)，而不是 *M* × *N* 个。

规则示例 ($E \rightarrow E_1 + E_2$):

```
E.addr = new Temp();
gen(E.addr, '+=', E1.addr, '+', E2.addr);
```

5.2 数组元素寻址 (Addressing Array Elements)

目标是计算数组元素的相对地址 (偏移量)。

一维数组 ($A[i]$):

$$addr = base + (i - low) \times w$$

通常假设 $low = 0$ 或在编译时合并常数项 $c = base - low \times w$, 则运行时只需计算 $i \times w$ 。

二维数组 ($A[i_1][i_2]$) - 行优先 (Row-major):

$$addr = base + (i_1 \times w_1 + i_2 \times w_2)$$

w_2 : 元素宽度。 w_1 : 行宽度。 $n_2 \times w_2$ (n_2 是一行的元素个数)。

k 维数组 (递归公式):

$$addr(A[i_1] \dots [i_k]) = addr(A[i_1] \dots [i_{k-1}]) + i_k \times w_k$$

6. 控制流与布尔表达式 (Control Flow)

6.1 布尔表达式的作用

计算逻辑值: 结果为 true 或 false。

改变控制流: 用于 if, while 等语句的条件判断。

6.2 短路代码 (Short-Circuit Code)

布尔运算符 &&, ||, ! 被翻译为跳转指令。运算符本身不出现在代码中, 表达式的值由代码执行的位置 (跳转到哪儿) 决定。

示例: if $x < 100$ || $x > 200$ && $x != y$ 翻译为一系列 if...goto 和 goto 指令。

6.3 回填技术 (Backpatching)

问题: 在单遍 (One-pass) 代码生成中, 生成跳转指令 (如 goto) 时, 目标标号 (Target Label) 往往还未确定 (例如跳转到 else 或循环结束处)。

解决方案: 生成跳转指令时, 目标留空。将这些未完成的跳转指令的索引保存在列表中。当目标位置确定时, 回填 (Backpatch) 这些指令的目标标号。

关键函数: makelist(i): 创建一个只包含指令索引 i 的列表。merge(p1, p2): 合并两个列表 p1 和 p2。backpatch(p, i): 将列表 p 中所有指令的目标回填为标号 i。

属性: B.truealist: 当 B 为真时跳转的指令列表。B.falsealist: 当 B 为假时跳转的指令列表。S.nextlist: S 执行完后跳转的指令列表 (用于跳过 else 或跳出循环)。M.quad: 标记非终结符 (Marker Nonterminal), 用于记录当前指令的索引 (即下一条指令的地址), 以便后续回填跳转目标。

6.4 控制流语句翻译逻辑

$S \rightarrow \text{if } (B) \text{ M } S_1: \text{backpatch}(B.\text{truealist}, \text{M.quad}): B$ 为真跳转到 $S1$ 开始处。 $S.\text{nextlist} = \text{merge}(B.\text{falsealist}, S1.\text{nextlist}): B$ 为假或 $S1$ 结束都去 $S.\text{next}$ 。

$S \rightarrow \text{while } M_1 (B) \text{ M}_2 \text{ S}_1: \text{backpatch}(S1.\text{nextlist}, M1.\text{quad}): S1$ 结束后跳回循环开始断处。 $\text{backpatch}(B.\text{truealist}, M2.\text{quad}): B$ 为真跳转到 $S1$ 。 $S.\text{nextlist} = B.\text{falsealist}: B$ 为假跳出循环。生成 goto $M1.\text{quad}$: 再次循环。

1. 存储组织 (Storage Organization)

1.1 运行时内存的逻辑划分

编译器将运行时内存逻辑上划分为以下区域, 以满足不同数据的生命周期需求:

代码区 (Code Area): 存放生成的目标代码 (机器指令)。特点: 大小在编译时确定, 通常是只读的。

静态区 (Static Area): 存放全局变量、常量等在编译时即可确定大小且在整个程序运行期间都存在的变量。特点: 大小在编译时确定。

堆区 (Heap Area): 存放生命周期不依赖于过程调用的动态分配数据 (如 C 中的 malloc 或 Java 中的 new 创建的对象)。特点: 大小动态变化, 向高地址增长。

栈区 (Stack Area): 存放过程调用的活动记录 (Activation Records), 用于管理局部变量、参数等。特点: 大小动态变化, 通常向低地址增长 (与堆相反增长)。

1.2 静态与动态存储分配

静态分配 (Static Allocation):

原理: 在编译时确定数据对象的存储位置。

适用: 全局变量、Fortran 77 等不支持递归的语言。

局限: 不支持递归调用, 不支持动态数据结构。

动态分配 (Dynamic Allocation):

栈式分配: 也就是栈空间分配, 用于处理过程调用。

堆式分配: 用于处理生命周期不确定的数据。

2. 栈空间分配 (Stack Space Allocation)

2.1 核心概念: 活动与活动树

过程的活动 (Activation): 过程的一次执行称为一次活动。生存期 (Lifetime): 从过程的第一步执行到最后一步执行的时间段。

活动树 (Activation Tree):

定义: 用于描述程序运行期间过程调用关系的树结构。

节点: 每个节点代表一个过程的活动。

根节点: main 过程的活动。

性质: 过程调用序列对应树的先序遍历 (Pre-order)。过程返回序列对应树的后序遍历 (Post-order)。

控制栈 (Control Stack):

原理: 过程调用和返回具有后进先出 (LIFO) 的特性, 因此使用栈来管理活动记录。

状态: 栈中的记录对应于当前未结束 (Live) 的活动路径 (即活动树中从根到当前活动节点的路径)。

2.2 活动记录 (Activation Record / Stack Frame)

活动记录是栈上分配的一块连续存储区, 用于保存单次过程调用的信息。其典型布局如下:

- 实参 (Actual Parameters): 调用者传递给被调用者的参数。
- 返回值 (Returned Values): 被调用者返回给调用者的数据。
- 控制键 (Control Link): 指向调用者的活动记录的指针 (通常指向调用者的动态链基址)。用于恢复活动指针 SP。
- 访问键 (Access Link): 指向静态父过程 (定义该过程的过程) 的活动记录的指针。用于访问非局部变量 (实现静态作用域)。
- 保存的机器状态 (Saved Machine Status): 包括返回地址 (Return Address) 和寄存器内容 (程序计数器 PC 等)。
- 局部数据 (Local Data): 过程内部定义的局部变量。
- 临时变量 (Temporaries): 表达式求值过程中产生的临时值 (如 $x+y*z$ 中的中间结果)。

布局原则: 变长数据 (如动态数组) 通常放在记录的末尾。调用者和被调用者之间传递的数据 (参数、返回值) 放在被调用者记录的开头 (紧跟调用者) 以便双方访问。

2.3 调用序列与返回序列 (Calling & Return Sequences)

这是实现过程调用的代码片段, 由调用者和被调用者共同完成。

(1) 调用序列 (Calling Sequence)

目标: 在栈上分配活动记录并初始化。

步骤 (典型): 调用者: 计算实参。调用者: 将返回地址和旧的栈顶指针 (top_sp) 存入被调用者的活动记录。调用者: 增加 top_sp, 使其指向新记录的开始。被调用者: 保存寄存器值和其他状态信息。被调用者: 初始化局部数据并开始执行。

(2) 返回序列 (Return Sequence)

目标: 恢复机器状态, 返回结果, 释放栈空间。

步骤 (典型): 被调用者: 将返回值放置在紧跟实参的位置。被调用者: 利用保存的机器状态恢复 top_sp 和寄存器。被调用者: 跳转到保存的返回地址。

3. 堆管理 (Heap Management)

3.1 内存管理器 (Memory Manager)

功能:

分配 (Allocation): 响应请求, 在堆中找到一块足够大的连续空闲空间。如果空间不足, 向操作系统申请更多内存。

回收 (Deallocation): 将不再使用的内存归还到空闲空间池, 以便重用 (通常不直接归还给 OS)。

目标: 空间效率: 最小化碎片。程序效率: 利用局部性原理提高运行速度。低开销: 分配和回收操作要快。

3.2 程序局部性原理 (Program Locality)

这是利用内存层次结构 (寄存器 > 缓存 > 内存 > 磁盘) 进行优化的基础。时间局部性 (Temporal Locality): 如果一个存储位置被访问, 它很可能在不久的将来再次被访问 (例如: 循环变量)。

空间局部性 (Spatial Locality): 如果一个存储位置被访问, 其附近的存储位置很可能在不久的将来被访问 (例如: 数组遍历)。

3.3 碎片化 (Fragmentation)

定义: 堆被分割成许多小的、不连续的空闲块 (Holes), 导致虽然总空闲空间足够, 但无法满足较大的内存请求。

解决: 必须将相邻的空闲块合并 (Coalescing) 成更大的块。

3.4 放置策略算法 (Placement Strategies)

当有内存请求时, 如何从空闲列表中选择合适的块?

- 最佳适配 (Best-Fit): 搜索整个空闲列表, 选择大小最接近且足够的块。
 - 优点: 保留了大块空闲区, 空间利用率较高。
 - 缺点: 搜索速度慢, 容易产生极小的碎片。
- 首次适配 (First-Fit): 搜索空闲列表, 选择第一个足够大的块。
 - 优点: 分配速度快。
 - 缺点: 容易在列表头部产生碎片, 可能会分割大块空间。
- 分箱策略 (Binning): 将空闲块按大小分类 (如 16 字节箱, 24 字节箱...)
 - 优点: 查找极快 (直接去对应大小的箱找)。
 - 缺点: 需要管理多个列表。

Doug Lea 的分配器 (dmalloc): GCC 编译器的内存管理器。使用分箱策略: 对于小块 (如 < 512 字节), 使用精确大小的箱 (8 字节对齐)。

对于大块, 使用按大小排序的列表。

荒野块 (Wilderness Chunk): 内存最高处的巨大空闲块, 可向 OS 申请扩展。

4. 补充知识 (Supplement)

4.1 访问键 (Access Link) 的工作原理

静态作用域 (Static Scope): 在支持嵌套过程定义的语言 (如 Pascal, Ada) 中, 内部过程可以访问外部过程定义的变量。

实现: 每个活动记录包含一个 Access Link, 指向其直接外层过程的活动记录。

查找: 访问非局部变量时, 沿着 Access Link 链向上查找, 直到找到定义该变量的层级, 嵌套深度差决定了需要跳转的步数。

4.2 显示表 (Display Table)

替代方案: 为了加速非局部变量的访问 (避免长链查找), 可以使用一个全局数组 display。Display[i] 指向嵌套深度为 i 的当前活动过程的活动记录。访问只需一次数组索引。

4.3 垃圾回收 (Garbage Collection)

概念: 自动回收堆中不再被引用的对象。

基本技术: 引用计数 (Reference Counting): 记录每个对象被引用的次数。归零时间回收 (缺点: 无法处理循环引用)。标记-清除 (Mark-and-Sweep): 从根集 (根、全局变量) 出发, 标记所有可达对象, 然后清除未标记对象。复制 (Copying): 将存活对象复制到新区域, 原区域整体回收 (解决碎片问题)。

4.4 代码生成器概览 (Overview)

1.1 输入与输出

输入: 中间表示 (IR) + 符号表。

输出: 目标程序 (Target Program)。可以是绝对机器语言、可重定位机器语言或汇编语言。

1.2 主要任务 (Primary Tasks)

指令选择 (Instruction Selection): 选择适当的目标机器指令来实现 IR 的操作。这取决于目标机器的指令集架构 (RISC/CISC)。

寄存器分配与指派 (Register Allocation and Assignment):

分配 (Allocation): 决定哪些变量的值应该驻留在寄存器中。

指派 (Assignment): 决定每个值具体驻留在哪个物理寄存器中。

指令排序 (Instruction Ordering): 安排指令的执行顺序以提高效率 (如利用指令流水线)。

2. 目标语言与寻址模式 (Target Language & Addressing)

2.1 简单目标机器模型

假设为一个三地址机器, 具有加载 (Load)、存储 (Store)、计算 (Computation)、跳转 (Jump) 等操作。

指令形式: OP dst, src1, src2。

2.2 寻址模式 (Addressing Modes)

代码生成器需要将 IR 中的名字转换为目标代码中的地址。

变量名 (Variable name): 直接引用分配给变量的内存位置 (直接寻址)。

索引寻址 (Indexed): a(r) 表示地址为 l-value(a) + contents(r)。用于访问数组。

间接寻址 (Indirect): constant(r): LD R1, 100(R2) $\rightarrow R1 = \text{contents}(100 + \text{contents}(R2))$ 。*: LD R1, *R2 $\rightarrow R1 = \text{contents}(\text{contents}(\text{contents}(R2)))$ (多级间接)。

立即数 (Immediate): #constant, 如 LD R1, #100 将常数 100 加载到寄存器。

3. 存储分配策略 (Storage Allocation)

如何为过程调用和返回生成代码, 以及如何管理变量的运行时地址。

3.1 静态分配 (Static Allocation)

原理: 所有数据对象的地址在编译时确定。

实现:

call callee: 将返回地址保存到被调用者活动记录的固定位置 (通常是开头)。跳转到被调用者的代码区。

```
ST callee.staticArea, #here + 20 // 保存返回地址
BR callee.codeArea // 跳转
```

return: 跳转到保存的返回地址。

```
BR *callee.staticArea
```

局限: 不支持递归调用, 因为每次调用都会覆盖固定的返回地址存储位置。

3.2 栈式分配 (Stack Allocation)

原理: 使用栈来动态管理活动记录, 支持递归。

实现: 使用寄存器 SP 指向栈顶活动记录。相对地址用于访问记录内的数据。调用序列 (Calling Sequence): 增加 SP 指向新的活动记录。保存返回地址到栈线上, 跳转到被调用者。

```
ADD SP, SP, #callee.recordSize
ST *SP, #here + 16
BR callee.codeArea
```

返回序列 (Return Sequence): 跳转到栈线上保存的返回地址。恢复 SP (减去记录大小)。

```
BR *0(SP)
SUB SP, SP, #callee.recordSize
```

4. 基本块与流图 (Basic Blocks & Flow Graphs)

这是代码优化的基础结构。

4.1 基本块 (Basic Blocks)

定义: 只有第一个指令是入口, 最后一个指令是出口的连续指令序列。中间没有跳转进入, 也没有跳转离开。

划分算法 (Partition Algorithm)

识别首指令 (Leaders): 中间代码的第一条指令。任何跳转指令 (conditional or unconditional jump) 的目标指令。紧跟在跳转指令之后的那条指令。

构造基本块: 每个首指令及其随后的指令构成一个基本块, 直到遇到下一个首指令或程序结束。

4.2 流图 (Flow Graphs)

节点: 基本块。

边 ($B \rightarrow C$): 表示控制流可能从 B 流向 C 。

存在边的条件: B 的末尾有跳转到 C 开头的指令。 C 在代码序列中紧跟在 B 之后, 且 B 不以无条件跳转结束。

循环 (Loops): 流图中的循环必须包含一个入口节点 (Entry), 该节点支配 (dominates) 循环内的所有其他节点。

6. 简单的代码生成器 (A Simple Code Generator)

6.1 关键数据结构

寄存器描述符 (Register Descriptor): 记录每个寄存器当前存放了哪些变量的值。

地址描述符 (Address Descriptor): 记录每个变量的当前值存放在哪些位置 (寄存器、栈、内存)。

6.2 代码生成算法

对于指令 $x = y + z$: 调用 getReg($x = y + z$) 选择寄存器 R_x, R_y, R_z 。如果 y 不在 R_y 中, 生成 LD R_y, y 。如果 z 不在 R_z 中, 生成 LD R_z, z 。生成 ADD R_x, R_y, R_z 。

更新描述符: R_z 仅包含 x 。 x 的位置仅为 R_x (从内存位置中移除, 因为寄存器值更新)。

6.3 寄存器选择函数 getReg(I)

策略如下:

利用现成: 如果操作数 y 已经在某个寄存器中, 直接使用该寄存器。

使用空闲: 如果没有, 选择一个空闲寄存器。

溢出 (Spill): 如果没有空闲寄存器, 选择一个被占用的寄存器, 生成 ST 指令将其值保存回内存, 然后抢占该寄存器。

评分标准: 选择那个包含“最近将来才会被使用”或“已在内存中有副本”的变量的寄存器进行溢出。

1. 局部优化 (Local Optimization)

范围: 单个基本块内部。

工具: DAG (有向无环图) 是基本块优化的核心数据结构, 用于描绘变量值之间的数据依赖关系。

1.1 DAG 的构造与原理

构造规则:

叶子节点: 为基本块中出现的每个变量的初始值创建节点。
内部节点: 为每条语句 (如 $a = b + c$) 创建节点, 标记为操作符 (如 +), 子节点为操作数的当前节点。

查重 (复用): 在创建新节点前, 检查是否已存在具有相同操作符和相同子节点 (顺序敏感) 的节点。如果存在, 则不创建新节点, 而是直接使用现有节点 (这是消除公共子表达式的键)

变量关联: 将被赋值的变量 (如 a) 附加到该节点上, 表示该变量的当前值。

1.2 常见的局部优化技术

基于 DAG 可以实现以下优化:

消除局部公共子表达式 (Local Common Subexpression Elimination):

原理: 如果一个表达式 $x + y$ 已经被计算过, 且 x 和 y 的值没有改变, 则不需要重新计算。

算法: 在 DAG 构造过程中, 通过检查是否存在相同节点来自动实现。
示例: $a = b + c; \dots; d = b + c$ 。若 b, c 未变, 则 d 直接指向 a 对应的节点。

死代码消除 (Dead Code Elimination):

定义: 死代码是指计算结果从未被使用的指令。

算法: 从 DAG 中删除没有任何活跃变量 (Live Variables, 即在块出口处仍有用的变量) 附加的根节点 (没有父节点的节点)。重复此过程直到无法删除。

代数恒等式的应用 (Application of Algebraic Laws):

消除计算: 利用代数恒等式简化代码。 $x + 0 = x, x - 0 = x, x \times 1 = x, x / 1 = x$ 。

强度削减 (Reduction in Strength): 用代价更低的操作符替换昂贵的操作符。 $x^2 \Rightarrow x \times x, 2 \times x \Rightarrow x + x, 2 \Rightarrow x \times 0.5$ 。

常量合并 (Constant Folding): 在编译时计算常量表达式的值。 $2 \times 3.14 \Rightarrow 6.28$ 。

2. 窥孔优化 (Peephole Optimization)

概念: 一种简单有效的局部优化技术。通过检查目标代码的一个滑动窗口 (窥孔), 用更短或更快的指令序列替换窗口内的指令序列。

常见变换:

消除冗余加载/存储: LD R0, a; ST a, R0 \Rightarrow LD R0, a (如果后续指令不跳转到 ST)。

消除不可达代码: 删除紧跟在无条件跳转 (goto, return) 之后的未标记指令。

控制流优化: goto L1; ... L1: goto L2 \Rightarrow goto L2。if a < b goto L1; ... L1: goto L2 \Rightarrow if a < b goto L2。

代数简化: 删除 $x = x + 0$ 等无效指令。

3. 全局优化 (Global Optimization)

范围: 整个流图 (跨越多个基本块)。

核心: 数据分析 (Data-Flow Analysis)。

3.1 数据流分析框架 (Data-Flow Analysis Schema)

传递函数 (Transfer Function, f_B): 描述语句或基本块如何改变数据流值。

前向流: $OUT[s] = f_s(IN[s])$ 。

后向流: $IN[s] = f_s(OUT[s])$ 。

控制流约束 (Control-Flow Constraints): 块内: $IN[s_{i+1}] = OUT[s_i]$ 。

块间 (前向): $IN[B] = \bigcup_{P \in pred(B)} OUT[P]$ (对前驱取交汇运算)。

块间 (后向): $OUT[B] = \bigcup_{S \in succ(B)} IN[S]$ (对后继取交汇运算)。

3.2 经典数据流问题详解

(1) 到达定义 (Reaching Definitions)

定义: 定值 d 到达点 p , 如果存在一条从 d 到 p 的路径, 且该路径上 d 没有被“杀死” (即变量被重新定义)。

方向: 前向 (Forward)。

传递函数: $f_B(x) = gen_B \cup (x - kill_B)$ 。 gen_B : 块内生成且未被后续定义覆盖的定值。 $kill_B$: 块内定义覆盖了的程序中其他地方的定值。

交汇运算 (Meet): 并集 (\cup) (只要有一条路径到达即可)。

方程: $OUT[B] = gen_B \cup (OUT[B] - kill_B)$ $IN[B] = \bigcup_{P \in pred(B)} OUT[P]$ 。

边界条件: $OUT[ENTRY] = \emptyset$ 。

应用: 常量传播, 未初始化变量检测。

(2) 活跃变量 (Live Variables)

定义: 变量 x 在点 p 是活跃的, 如果在从 p 开始的某条路径上引用了 x , 且在引用前没有对 x 重新赋值。

方向: 后向 (Backward) (因为后续的使用决定了当前的活跃性)。

传递函数: $IN[B] = use_B \cup (OUT[B] - def_B)$ 。 use_B : 块内使用且使用前未被定义的变量。 def_B : 块内定义且定义前未被使用的变量。

交汇运算 (Meet): 并集 (\cup) (只要有一条路径使用即为活跃)。

方程: $IN[B] = use_B \cup (OUT[B] - def_B)$ $OUT[B] = \bigcup_{S \in succ(B)} IN[S]$ 。

边界条件: $IN[EXIT] = \emptyset$ (假设程序退出后变量不再活跃)。

应用: 死代码消除、寄存器分配。