Language & Syntax Description

Alphabet & String

Alphabet

非空 symbol 集，使用大写希腊字母如 Σ, V 表示

Symbol(Character)

alphabet 中的元素，Language 中最小的元素，使用 $a, b, c, \cdots$ 表示

String

symbol 组成的有限序列，null-string 记作 ϵ，使用 $\alpha, \beta, \gamma, \cdots$ 表示

Sentence

使用某种构建规则形成的 string 序列，使用 $A, B, C, \cdots$ 表示

Language

sentence 的集合

String 集的操作

Concatenate(Product) 操作

若有 $A = \{\alpha_1, \alpha_2, \cdots\}, B = \{\beta_1, \beta_2, \cdots\}$，则 (Cartesiane)Product AB 定义为 $AB = \{\alpha\beta \mid \alpha \in A \wedge \beta \in B\}$

- string 集自己 product 称为 power
- $A^0 = \{\epsilon\}$
- A^n 是一个所有 string 长度都为 n 的集合

Closure 和 positive closure

- **Closure**

  $A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$

- Positive closure

  $A^+ = A^1 \cup A^2 \cup \cdots = A^* - \{\epsilon\}$

  language 就是 positive closure 的子集

Grammar & Language

基础概念

Grammar

grammar 是 syntax elements 形式化的 production 规则

- syntax elements 包括 sentence 和 word
- production 规则的形式为 左边→右边

Non-terminal symbol

是出现在规则左边的符号，放在 `<>` 内，non-terminal symbol 的集合表示为 V_N

Terminal symbol

不可分解的 string，包括单 character 的 string，使用 V_T 表示，是 setence 的基础元素

Start symbol

一种特殊的 Non-terminal symbol，又名 identified symbol

Production

描述 string 关系的规则集，形式为 $A \to \alpha$，即 A 具有 α 的形式

- 例如 $< Sentence > \to < Subject >< Predicate >$

Derivation

从 start symbol 开始的过程，根据 production 规则，通过用右边替换左边来 derive 一个 sentence

- Leftmost(Rightmost) derivation 每次只使用一次 production 规则，用右边替换最左侧或最右侧的 terminal symbol。也称 canonical derivation

Reduction

reduction 为 derivation 的逆，通过使用左边替换右边，由给定的 sentence 推导出 start symbol

- Leftmost(Rightmost) reduction 也称 canonical reduction

Sentential form、Sentence & Language

- **Sentential form**

  从 start symbol 每次 derivation 出来的 string α，记作 $S \overset{*}{\to} \alpha, \alpha \in (V_N \cup V_T)^*$

- **Sentence**

  只包含 terminal symbol 的 sentential form

- **Language**

  由 S 的一个或多个 derivation produce 出来的 sentence 集或 string 集，记作 $L(G) : L(G) = \left\{ \alpha \mid S \overset{+}{\to} \alpha \wedge \alpha \in V_T^* \right\}$

Grammar 规则的递归定义

non-terminal symbol 的定义中包括了另一个 non-terminal symbol

Grammar 规则的扩展符号

使用 BNF(Backus Naur Form) 符号

- `()` 提取公因子。如 $U \to ax \mid ay \mid az$ 记作 $U \to a(x \mid y \mid z)$
- `{}` 重复。如 $< Identifier > \to < Letter > \{< Letter > | < Digit >\}_0^5$
- `[]` 可选。如 $< Integer > \to [+ \mid -] < Digit > \{Digit\}$

Meta-language symbol

描述 grammar symbol 之间关系的 symbol。如 $\to, |$

Formal definition

Grammar 定义

G 定义为四元组 (V_N, V_T, P, S)

Catalog of grammars

- **0-type grammar**(Phrase grammar or grammar without limitation)

  P 中的形如 $\alpha \to \beta, \alpha \in V^+, \beta \in V^*$ 的 production，α 至少有一个 non-terminal symbol

  - 识别 0-type grammar 的叫做图灵机
  - 0-type grammar 是 production 限制最少的 grammar
  - 其他 grammar 类型由 0-type grammar 得出
- **1-type grammar**(context-sensitive grammar or length-added grammar)

  P 中形如 $\alpha \to \beta$ 的 production，若 $S \to \varepsilon$，则 S 不会出现在右侧；其余情况下，$|\beta| \geq |\alpha|$

  即除了 $S \to \varepsilon$ 均有形式 $\alpha A \beta \to \alpha \gamma \beta, \alpha, \beta \in V^*, A \in V_N, \gamma \in V^+$

  - 自动机称为 Linear Bound (LBA)
  - 替换 non-terminal symbol 时需要考虑上下文。non-terminal symbol 不能用 ε 代替，除非是 produce ε 的 start symbol
- **2-type grammar**(Context-free grammar)

  P 中形如 $A \to \beta$ 的 production，$A \in V_N, \beta \in V^*$

  - 左侧是 non-terminal symbol，右侧可以是 V_N, V_T, ε
  - 自动机称为 Pushdown Automation(PDA)
- **3-type grammar**(Regular grammar, right-linear grammar or left-linear grammar)

  P 中形如 $A \to \alpha B, A \to \alpha$ 或 $A \to B\alpha, A \to \alpha$ 的 production，$A, B \in V_N, \alpha \in V_T^*$

  - 3-type grammar 的 production 为 right-linear productions 或者 left-linear productions。如果 3-type grammar 的所有 production 均为 right(left)-linear productions，则称其为 right(left)-linear grammar
  - 自动机称为 finite state automation
  - 2-type grammar 即 self-embedded grammar($S \to aSb$) + regular grammar

Hierarchy	别名	Production 形式	自动机名字
0-type	Grammar without limitation	$\alpha \to \beta, \alpha \in V^+$	Turing Machine
1-type	Context-sensitive grammar	$\alpha A \beta \to \alpha \gamma \beta, A \in V_N$	Linear Bound Automation
2-type	Context-free grammar	$A \to \beta, A \in V_N$	Pushdown automation
3-type	Regular grammar	$A \to \alpha B, A \to \alpha, A, B \in V_N, \alpha \in V_T^*$	Finite automation

i-type language

记作 $L(G)$：$L(G) = \left\{\omega \mid \omega \in V_T^* \wedge S \xrightarrow{+} \omega\right\}$

- 例如对于 $G_1 = (\{S\}, \{a, b\}, P, S)$，$P$ 包含 $S \to aS, S \to a, S \to b$，则 $L(G_1) = \left\{a^i(a \mid b) \mid i \geq 0\right\}$
- 例如对于 $G_2 = (\{S\}, \{a, b\}, P, S)$，$P$ 包含 $S \to aSb, S \to ab$，则 $L(G_2) = \{a^n b^n \mid n \geq 1\}$

Grammar construction and simplification

Constructing a grammar from a language by experiences

对称方法

针对具有对称性的 language

- 找到对称轴
- 找出对称特性

逐步求精法

- 从左至右
- 从上至下

等效法

针对要求数目相等的 language

电路状态转换图法

针对数目奇偶性的 language

- 全零作为结束态

有限自动机法

层次法

Grammar Simplification

- 删除 $P \to P$
- 删除不能用于 derivation 的 production
- 删除不能 derive 到 terminal string 的 production
- 重新组织剩余 production

Construct a context-free grammar without ε-production

作用：减少推导步骤

production 条件

- 若 P 中有 production $S \to \varepsilon$，则 S 不应当出现在任何 production 的右侧，S 为 start symbol
- P 中没有其他 ε-production

构造算法

$G = (V_N, V_T, P, S) \Longrightarrow G' = (V_N', V_T', P', S')$

- 找出所有能经过若干步 derive 到 ε 的 non-terminal symbol，把他们放入集合 V_0
- 若 V_0 中的一个 symbol 出现在一个 production 的右侧，则不断把这个 symbol 替换为 ε 和它本身，并把产生的 production 放入 P'
- 否则，若该 symbol 与 ε-production 无关，则把与其相关的 production 放入 P'
- 若 P 中存在 $S \to \varepsilon$，则改变为 $S' \to \varepsilon \mid S$ 并放入 P'，S' 为 G' 的 start symbol，$V_N' = V_N \cup \{S'\}$

Ambiguity of a grammar

- 若一个 sentence 有两个及以上的 syntex 树，则为 ambiguous
- 若一个 language 包含 ambiguous 的 sentence，则它也是 ambiguous 的

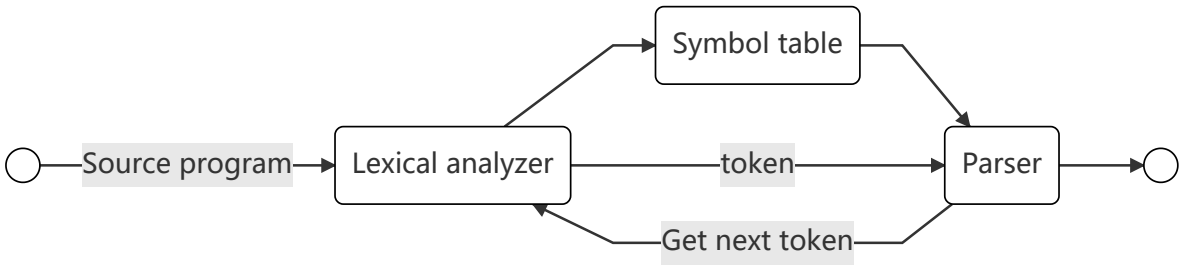
Lexical Analysis

The role of lexical analyzer

主要任务

- 读取输入的字符
- 产生给 parser 用作 syntax analysis 的 token 序列
- 作为 parser 的辅助

lexical analyzer 和 parser 的交互



Processes in lexical analyzers

- 扫描

  预处理：删除注释和空白，替换宏函数
- 将 compiler 的错误信息与源程序关联
- lexical analysis

Terms of the lexical analyzer

- token

  格式为 (Lexeme, Type, Inner_code)

  type 种类有：keyword,operator,identifiermcinstant,literal string,punctuation symbol
- lexeme

  源程序中实际的单词

Specification of Tokens

Regular Expression & Regular language

每个 regular expression r 都意味着一个 language $L(r)$

The rule of regular expression over alphabet Σ

- ε 属于 $\{\varepsilon\}$
- 若 a 为 Σ 中的 symbol，则其属于 $\{a\}$
- 若 α 和 β 为 regular expression，则 $\alpha \mid \beta, \alpha\beta, \alpha^*, \beta^*$ 也为 regular expression

Recognition of Tokens

Task of recognition of token in a lexical analyzer

- 将每个 lexeme 单独处理
- 输出为 <id,表指针>

Transition Diagram(Stylized flowchart)

- 每个圈代表一个 state，圈内有一个该状态的 id
- 圈到圈之间的箭头代表读入的字符
- 结束字符使用双层圈表示
- 带有星号（*）的表示读取下一个字符时，需要回退一个字符

A generalized transition diagram

分为 Deterministic 和 non-deterministic FA

non-deterministic 表示在某一 state 时，同一个输入可能有不同的路径

Finite automata

DFA

- DFA 格式为 $M(S, \Sigma, move, s_0, F)$
  - S: state 的集合
  - Σ: 输入的 symbol alphabet
  - $move$: 转换函数，$move : S \times \Sigma \to S$，$move(s, a) = s'$
  - s_0: 开始 state，$s_0 \in S$
  - F: 一组互不相同的 accepting states，$F \subseteq S$
- 没有 ε 转换

  离开 s 且标有 a 的边最多仅有一条

  DFA 接收一个 string 当且仅当转换图中存在这样一条路径

NFA

- NFA 格式为 $M(S, \Sigma, move, s_0, F)$
  - $move$: 转换函数，$move : S \times \Sigma \to S$，$move(s, a) = 2^S, 2^S \subseteq S$
- 离开 s 且标有 a 的边可以不止一条

  DFA 是 NFA 的特例

Conversion of an NFA into a DFA

可以防止 ambiguity

- ε-closure(T)

  T 中的 NFA state s 只经过 ε-transition 就可以得到的 NFA states 的集合
- Subset Construction algorithm
  - 输入：NFA $N = (S, \Sigma, move, S_0, Z)$

    输出：DFA $D = (Q, \Sigma, \delta, I_0, F)$
  - 算法
    - $I_0 =$ε-closure(S_0),$I_0 \in Q$
    - 对所有的 $I_i \in Q$，$I_t =$ε-closure($move(I_i, a)$)，即 I_i 经过一步（ε 不计入步数）可以到达的所有 state，一个 I_i 可能会有多个 I_t。若 $I_t \notin Q$，则把 I_t 放入 Q
    - 重复上一步，直到没有新 state 可以被放入 Q
    - $F = \{I \mid I \in Q, I \cap Z \neq \varnothing\}$
  - DFA 速度更快，但也更大

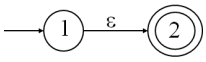
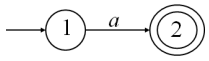
Minimizing the number of States of a DFA

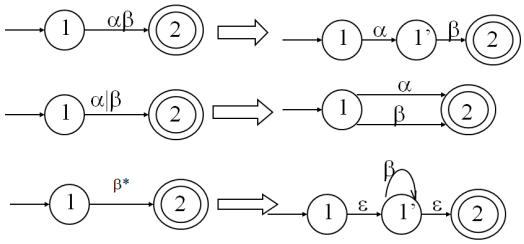
- 构建初始分割集合 Π_0，该集合包含两组 state：accepting states 和 non-accepting states。

  $\Pi_0 = \{I_0^1, I_0^2\}$
- 对于 Π_{i-1} 中的每组 state，对其进行划分。该组中的任意两个 state 会被划分到新集合的同一组中，当且仅当对于任意输入，这两个 state 的转换结果到达同一组。将所得的划分结果记录为 Π_i。重复执行直到 $\Pi_i = \Pi_{i-1}$
- 在 Π_i 的每组中选取一个代表，它们构成了新的 state 集合
- 若存在 dead state（即非 accepting 且对于所有输入，只能到达它自己）则删了它

  删除所有由 start state 不能到达的 state

Regular expression to an NFA

- 将正则表达式 r 拆分为子表达式
- 对于 r 中的每个 symbol
  - ε: 
  - a: 
- 将它们组合起来



A FA to Regular expression

- 前后分别加一个 start state 和 accepting state

- 将 Regular expression to an NFA 的方法倒过来使用

Regular Grammar to an NFA

Right-linear grammar to FA

输入 $G = (V_N, V_T, P, S)$

输出 $M = (Q, \Sigma, move, q_0, Z)$

- G 中的每个 non-terminal symbol 都作为一个 state

  增加新 state T 作为 accepting state
- $Q = V_N \cup \{T\}$

  $\Sigma = V_T$

  $q_0 = S$

  若存在 $S \to \varepsilon$，则 $Z = \{S, T\}$；否则 $Z = \{T\}$

- 对于 $A_1 \to aA_2$，构建 $(A_1, a) = A_2$

  对于 $A_1 \to a$，构建 $(A_1, a) = T$

  对于 Σ 中的所有 a，$(T, a) = \varnothing$

FA to Right-linear grammar

输入 $M = (S, \Sigma, f, s_0, Z)$

输出 $G = (V_N, V_T, P, s_0)$

- 若 $s_0 \notin Z$

  对于 $f(A_i, a) = A_j$，构建 $A_i \to aA_j$

  若 $A_j \in Z$，构建 $A_i \to a \mid aA_j$
- 若 $s_0 \in Z$

  对于 $f(s_0, \varepsilon) = s_0$，构建 $s_0' \to \varepsilon \mid s_0$，$s_0'$ 作为新的 accepting state

Design of a lexical analyzer generator

Syntax Analysis

The role of the parser

- 从 lexical analyzer 处获得 tokens 的 string
- 验证 string 可以有相关编程语言的 grammar 产生
- 报告 syntax 错误

Top-down parsing

Recursive descent

从上至下构建 parse tree。当出现不匹配后，程序返回最近的 non-terminal

- left-recursive grammar 会导致它进入死循环
- 具有 ambiguity 的 grammar 会导致它回溯
- left factor 也会导致回溯

Elimination of left recursion

- left recursion 的形式

  直接递归 $P \to P\alpha \mid \beta$

  间接递归 $P \to Aa, A \to Pb$
- 算法

  消除直接递归

  - $P \to P\alpha \mid \beta \Longrightarrow P \to \beta\alpha^* \Longrightarrow P \to \beta P', P' \to \alpha P' \mid \varepsilon$

  消除间接递归

- 设 G 中的 non-terminals 为 $P_1, P_2, \cdots, P_n$
- 对任意 P_i 和 $k < i$，将 $P_i \to P_k\gamma$ 替换为 $P_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_n\gamma, P_k \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_n$

  再将 $P_i \to P_i\alpha_1 \mid \cdots \mid P_i\alpha_m \mid \beta_1 \mid \cdots \mid \beta_n$ 替换为
  $P_i \to \beta_1 P_i' \mid \cdots \mid \beta_n P_i', P_i' \to \alpha_1 P_i' \mid \cdots \mid \alpha_m P_i' \mid \varepsilon$
- 化简 grammar

Eliminating ambiguity of a grammar

Left factoring

$A \to \delta\beta_1 \mid \delta\beta_2 \mid \cdots \mid \delta\beta_n$ 替换为 $A \to \delta A', A' \to \beta_1, \beta_2, \cdots, \beta_n$

Non-recursive predictive parsing

M 为 parsing 表，X 为栈顶的 symbol，a 为当前输入的字符，$ 为输入终止符

- 若 $X = a = \$$，parser 中止并返回成功
- 若 $X = a \neq \$$，弹出 X，将输入指针指向下一个输入 symbol
- 若 X 为 non-terminal，程序查询 $M[X, a]$

Construction of a predictive parser

- FIRST

  $\alpha \in V^*, \mathrm{FIRST}(\alpha) = \{a \mid \alpha \to a\cdots, a \in V_T\}$

  若 $\alpha \xrightarrow{+} \varepsilon$，则 ε 也在 $\mathrm{FIRST}(\alpha)$ 中
- FOLLOW

  $\mathrm{FOLLOW}(A) = \{a \mid S \to \cdots Aa\cdots, a \in V_T\}$

  若 $S \to \cdots A$，则 $\$ \in \mathrm{FOLLOW}(A)$
- $\mathrm{FIRST}(\alpha)$ 计算

  X 为任意 symbol 或 string
  - 若 X 为 terminal symbol，则 $\mathrm{FIRST}(X) = \{X\}$
  - 若有 $X \to \varepsilon$，则将 ε 加入 $\mathrm{FIRST}(X)$
  - 若 X 为 non-terminal 或 string，且 $X \to Y_1Y_2\cdots Y_k, Y_j \in (V_N \cup V_T)$

    若 $Y_1, \cdots, Y_{i-1}, i < k$ 中均有 ε symbol，则向 $\mathrm{FIRST}(X)$ 中加入 $\mathrm{FIRST}(Y_i) - \{\varepsilon\}$

    若 $Y_1, \cdots, Y_k$ 中均有 ε symbol，则向 $\mathrm{FIRST}(X)$ 中加入 ε
- $\mathrm{FOLLOW}(A)$ 计算
  - S 为 start symbol，则将 $ 放入 $\mathrm{FOLLOW}(S)$
  - 若 G 中有 $B \to \alpha A\beta$，则将 $(\mathrm{FIRST}(\beta) - \varepsilon)$ 放入 $\mathrm{FOLLOW}(A)$
  - 若有 $B \to \alpha A$ 或 $B \to \alpha A\beta$ 且 $\mathrm{FIRST}(\beta)$ 包含 ε，则将 $\mathrm{FOLLOW}(B)$ 放入 $\mathrm{FOLLOW}(A)$
- 构建 predictive parsing 表

  输入 grammar G

  输出 parsing 表 M
  - 对每个 production $A \to \alpha$
    - 对 $\mathrm{FIRST}(\alpha)$ 中的每个 terminal a，将 $A \to \alpha$ 添加到 $M[A, a]$ 中
    - 若 ε 在 $\mathrm{FIRST}(\alpha)$ 中，对 $\mathrm{FOLLOW}(A)$ 中的每个 terminal b，将 $A \to \alpha$ 添加到 $M[A, b]$ 中

若 ε 在 $\mathrm{FIRST}(\alpha)$ 中且 \$ 在 $\mathrm{FOLLOW}(A)$ 中，将 $A \to \alpha$ 添加到 $M[A, \$]$ 中

- 将 M 所有空条目定义为 error

LL(1) Grammars

没有多重定义 entries 的 parsing 表的 grammar 称为 LL(1)

- LL(1) 没有 ambiguous
- G 为 LL(1) 当且仅当任意 $A \to \alpha \mid \beta$ 为两个不同的 productions

Transform a grammar to LL(1) Grammar

- 消除所有 left recursion
- left factoring

Bottom-up parsing

LR Parsers

LR parser

LR(k)：L - 从左至右扫描，R - 逆向构建 rightmost serivation，k - 向前查看输入 symbols 的数量

LR parsing model

LR 算法由输入、输出、栈、parsing table 组成

- 输入输出：以 \$ 结尾的字符串
- 栈：栈底为 \$
- parsing table：由 action part 和 goto part 两部分组成

Parsing table

- Action

  $\mathrm{action}[S, a]$：S 为栈顶的状态，a 为当前输入的 symbol

  - shift：将下一个输入的 symbol 压入栈顶，符号为 S_j，表明 shift state j
  - reduce：栈顶存储着 handle 的右端，parser 定位到 handle 的左端并使用 noon-terminal symbol 替换这个区间，符号为 r_j，表明 reduce production j
  - accept：parser 表明编译成功
  - error： 发生了 syntax error 并调用 error recovery routine
- Action conflict
  - shift / reduce conflict：无法找到要 shift 或 reduce 的位置
  - reduce / reduce conflict：无法决定执行哪个 reduction

  ambiguous grammar 会导致 conflicts 且不可能成为 LR
- Goto

  把一个 state 和 grammar symbol 作为 arguments 并产生一个 state

Algorithm

(状态栈, 已 reduce 的符号栈, 待分析的输入串)

- 若 $\text{action}[S_m, a_i] = \text{shift}S'$，则执行 shift，将 $S' = \text{goto}[S_m, a_i]$ 和 a_i 入栈，a_{i+1} 成为当前 symbol

  $(S_0S_1\cdots S_m, \$X_1X_2\cdots X_m, a_ia_{i+1}\cdots a_n\$) \longrightarrow (S_0S_1\cdots S_mS', \$X_1X_2\cdots X_ma_i, a_{i+1}\cdots a_n\$)$
- 若 $\text{action}[S_m, a_i] = \text{reduce}A \to \alpha$，则执行 move。若 α 长度为 γ，则从栈中删除 γ 个 state，使得栈顶为 $S_{m-\gamma}$。将 $S' = \text{goto}\,[S_{m-\gamma}, A]$ 和 non-termina A 入栈。当前 symbol 不变

  $(S_0S_1\cdots S_m, \$X_1X_2\cdots X_m, a_ia_{i+1}\cdots a_n\$) \longrightarrow (S_0S_1\cdots S_{m-\gamma}S', \$X_1X_2\cdots X_{m-\gamma}A, a_ia_{i+1}\cdots a_n\$)$
- 若 $\text{action}[S_m, a_i] = \text{accept}$，则 parsing 完成

  $(S_0S_1\cdots S_m, \$X_1X_2\cdots X_m, a_ia_{i+1}\cdots a_n\$) \longrightarrow (S_0S_1, \$E, \$)$，$E$ 为文法开始符号
- 若 $\text{action}[S_m, a_i] = \text{error}$，则 parsing 完成并调用 error recovery routine

LR grammar

- LR grammar 是可以构造 parsing table 的 grammar
- LR 可以比 LL 描述更多的 language

Canonical LR(0)

- LR(0) item

  LR(0) 就是在 production 的右侧加一个点，点的右侧为期望输入的 string，可以理解为符号栈站内外的分界线

  $A \to \varepsilon$ 只能产生一种 item $A \to \cdot$
- 构造 canonical LR(0) collection
  - 若 S 为 start symbol，则令 S' 为新的 start symbol，增加 $S' \to S$
  - 设 I 为 items 的一个集合，构造 $\text{closure}(I)$
    - 将 I 的所有 items 加入 $\text{closure}(I)$
    - 若 $A \to \alpha \cdot B\beta \in \text{closure}(I), B \in V_N$，则对于所有 production $B \to \gamma$，item $B \to \cdot\gamma$ 也属于 $\text{closure}(I)$
  - 执行 goto

    若 I 为 items 的一个集合，X 是一个 symbol，则 $\text{goto}(I, X) = \text{closure}(J)$，其中 J 为 I 中所有形如 $A \to \alpha X \cdot \beta$ 或 $A \to \alpha \cdot X\beta$ 的 items
  - 构造 canonical LR(0) collection
    - 初始化 $C = \text{closure}(\{S' \to \cdot S\})$
    - 对于 C 中的每个 items 集合 I 和每个 symbol X，如果 goto(I,X) 非空且不属于 C，则加入到 C 中。不断重复直到 C 不再扩大

SLR(1) parsing table algorithm

- $C = \{I_0, I_1, \cdots, I_n\}$
- 若 $A \to \alpha \cdot a\beta \in I_k$ 且 $\text{goto}(I_k, a) = I_j$，则 $\text{action}[k, a] = S_j$
- 若 $A \to \alpha\cdot \in I_k$，则对所有 $a \in \text{FOLLOW}(A)$，$\text{action}[k, a] = r_j$，$j$ 为 production $A \to \alpha$ 的编号。其中 A 不为 S'
- 若 $\text{goto}(I_k, A) = I_j, A \in V_N$，则 $\text{goto}[k, A] = j$
- 若 $S' \to S\cdot \in I_k$，则 $\text{action}[k, \$] = \text{accept}$

若构造表时未产生冲突，则为 SLR grammar

LR(1) item

1 指的是 look-ahead of the item 的长度

valid LR(1) item

若存在 production $S' \to \delta A\omega \to \delta\alpha\beta\omega$，其中 $\gamma = \delta\alpha$ 且 $a \in \mathrm{FIRST}(\omega)$ 或 $\omega = \varepsilon, a = \$$，则 $(A \to \alpha \cdot \beta, a)$ 对于 viable prefix γ 是有效的

Construction of the sets of LR(1) items

- $\mathrm{closure}(I)$
  - I 中所有 items 都属于 $\mathrm{closure}(I)$
  - 若 $(A \to \alpha \cdot B\beta, a)$ 属于 $\mathrm{closure}(I)$，对于每个 production $B \to \gamma$ 和 $\mathrm{FIRST}(\beta a)$ 中的每个 terminal symbol b，若 $(B \to \cdot\gamma, b)$ 不在 $\mathrm{closure}(I)$ 中，则加入进去。重复这一步骤
- $\mathrm{goto}(I, X)$

  $\mathrm{goto}(I, X) = \mathrm{closure}(J)$，其中 J 为 I 中所有形如 $(A \to \alpha X \cdot \beta, a)$ 或 $(A \to \alpha \cdot X\beta, a)$ 的 items
- 构造 the sets of LR(1) items
  - 初始化 $C = \mathrm{closure}(\{(S' \to \cdot S, \$)\})$
  - 对于 C 中的每个 items 集合 I 和每个 symbol X，如果 $\mathrm{goto(I,X)}$ 非空且不属于 C，则加入到 C 中。不断重复直到 C 不再扩大

Construction of the canonical LR parsing table

- $C = \{I_0, I_1, \cdots, I_n\}$
- 若 $(A \to \alpha \cdot a\beta, b) \in I_k$ 且 $\mathrm{goto}(I_k, a) = I_j$，则 $\mathrm{action}[k, a] = S_j$
- 若 $(A \to a\cdot, a) \in I_k$，则对所有 $a \in \mathrm{FOLLOW}(A)$，$\mathrm{action}[k, a] = r_j$，$j$ 为 production $A \to \alpha$ 的编号。其中 A 不为 S'
- 若 $\mathrm{goto}(I_k, A) = I_j, A \in V_N$，则 $\mathrm{goto}[k, A] = j$
- 若 $(S' \to S\cdot, \$) \in I_k$，则 $\mathrm{action}[k, \$] = \mathrm{accept}$

若构造表时未产生冲突，则为 LR(1)

每个 SLR(1) 都是 LR(1)，LR 的 states 比 SLR 多

LALR

合并相同 core 的 state

相同的 core 指的是 production 相同，而搜索 symbol 不同

Using ambiguous grammars

Using precedence and associativity to resolve parsing action conflicts

dangling-else ambiguity

Parser generator Yacc

```
declaration
%%
translation rules
<Left side>: <alt> {semantic action}
%%
supporting C-routines
```

Syntax-directed translation

Syntax-directed definitions

Definitions

- syntax-directed definition：生成 context-free grammar，其中每个 symbol 都有相关的一系列 attributes
- attribute：其值由节点所用的 production 和 semantic rule 决定
  - synthesized attribute：由子节点计算得到
  - inherited attribute：由兄弟和父亲节点计算得到
- dependency graph
- annotated parse tree：计算节点 attribute 的过程

Form of a syntax-directed definition

对于每个 grammar production $A \to \alpha$，其对应着一系列形如 $b = f(c_1, c_2, \cdots, c_k)$ 的 semantic rules

- b 是 A 的 synthesized attribute
- b 是 production 右侧任一 symbol 的 inherited attribute

Synthesized attributes

- S-attributed definition

  仅采用 synthesized attributes 的 SDD
- Annotation for a parse tree for an S-attributed definition

  自底向上分析 attributes 的 semantic rules

Inherited attributes

Bottom-up evaluation of the S-attributed definitions

basic idea

- 通常使用 LR-parser 实现
- 栈中存储已经 parsed 的子树

Evaluating the synthesized attributes

Type checking

Type systems

Type expressions

Type systems

Static and dynamic checking of types

Specification of a simple type checker

A simple language

- source language 的 grammar
  - $P \rightarrow D;E$
  - $D \rightarrow D;D \mid \text{id} : T$
  - $T \rightarrow \text{char} \mid \text{integer} \mid \text{array[num] of } T \mid \text{\^{}}T$
  - $E \rightarrow \text{leteral} \mid \text{num} \mid \text{id} \mid E \text{ or } E \mid E[E] \mid E\text{\^{}}$
- The part of a translation scheme that saves the type of an identifier
  - $P \rightarrow D;E$
  - $D \rightarrow D;D$
  - $D \rightarrow \text{id} : T \{\text{addtype(id.entry,T.type)}\}$
  - $T \rightarrow \text{char} \{\text{T.type=char}\}$
  - $T \rightarrow \text{integer} \{\text{T.type=integer}\}$
  - $T \rightarrow \text{array[num] of T} \{\text{T.type=array}(1..\text{num}, T_1.\text{type}\}$
  - $T \rightarrow \text{\^{}}T_1 \{\text{T.type=pointer}(T_1.\text{type}\}$

Type checking of expressions

- $E \rightarrow \text{leteral} \{\text{E.type=char}\}$
- $E \rightarrow \text{num} \{\text{E.type=integer}\}$
- $E \rightarrow \text{id} \{\text{E.type=lookup(id.entry)}\}$
- $E \rightarrow E_1 \bmod E_2$

  $\{E.\text{type=if}(E_1.\text{type==integer}) \;\&\&\; (E_2.\text{type==integer})\text{integer else type\_error}\}$
- $E \rightarrow E_1[E_2]$

  $\{E.\text{type=if}(E_2.\text{type==integer}) \;\&\&\; (E_1.\text{type==array(s,t))t else type\_error}\}$
- $E \rightarrow E_1\text{\^{}}$

$\{E.\text{type}=\text{if}(E_1.\text{type}==\text{pointer}(\text{t}))\ \text{t else type\_error}\}$

Type checking of statements

- $S \rightarrow \text{id}:=E$

  $\{S.\text{type}=\text{if}(\text{id.type}==E.\text{type})\ \text{void else type\_error}\}$
- $S \rightarrow \text{if}\ E\ \text{then}\ S_1$

  $\{S.\text{type}=\text{if}(\text{E.type}==\text{bollean})\ S_1.\text{type else type\_error}\}$
- $S \rightarrow \text{while E do}\ S_1$

  $\{S.\text{type}=\text{if}(\text{E.type}==\text{bollean})\ S_1.\text{type else type\_error}\}$
- $S \rightarrow S_1; S_2$

  $\{S.\text{type}=\text{if}(S_1.\text{type}==\text{void})\ \&\&\ (S_2.\text{type}==\text{void})\text{void else type\_error}\}$

Type checking of functions

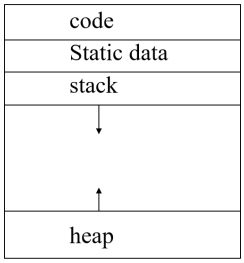
- $T \rightarrow T_1\text{'} \rightarrow \text{'}T_2\ \{T.\text{type}=T_1.\text{type} \rightarrow T_2.\text{type}\}$
- $E \rightarrow E_1(E_2)$

  $\{E.\text{type}=\text{if}(E_2.\text{type}==\text{s})\ \&\&\ (E_1.\text{type}==s \rightarrow t)\text{t else type\_error}\}$

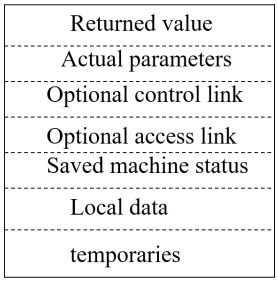
Runtime environments

Storage organization

Subdivision of runtime memory



Activation records

- 定义：某一个执行进程所需要的连续的存储空间
- 

Storage allocation strategies

Storage allocation strategies

- static allocation：存储编译时的所有数据
- stack allocation：以栈的形式管理 run-time storage
- heap allocation：以堆的形式在运行时 allocate 和 de-allocate 存储

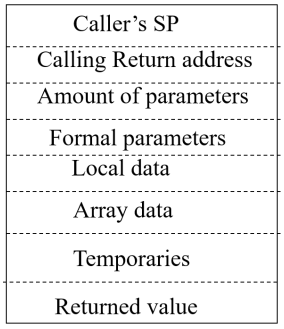
Static allocation

- 数据大小和编译时必须指导的数据
- 禁止出现递归
- 无法动态创建的数据结构

Heap allocation

Stack allocation

- Stack allocation for non-nested procedure



Parameter passing

- call-by-value
- call-by-reference
- copy-restore
- call-by-name

Intermediate code generation

Intermediate languages

Three-address code(TAC)

`x=y op z`

- x 为 names，y 为 constants，z 为 compiler-generated temporaries

Types of TAC

- `x=y op z`
- `x=op y`
- `x=y`
- `goto L`
- `if x relop y goto L`
- `param x`

  `call p,n`

  `return y`
- `x=y[i]`

  `x[i]=y`
- `x=&y`

  `x=*y`

  `*x=y`

Syntax-directed Translation into TAC