CH2 词法分析

2.1 实现方式

1. 先由自然语言规则通过人工描述将其变成 REX
2. 把 REX 使用 Thompson 算法以自动化的形式转化为 NFA
3. 通过子集构造法（也是自动化的形式）转成 DFA
4. 通过 Hopcroft 最小化算法简化 DFA
5. 组合不同 token 类型的 DFA，再通过一个二维数组判断状态转换（state [0]是判定为不接收的状态），一个一维 finality 数组实现终止状态类型判断实现 DFA 的自动执行来（不是很重要）生成词法分析器的代码

2.2 正则表达式

优先级

闭包>连接>选择

简写方式

- `[abcd]` 代表 `(a|b|c|d)`
- `[b-g]` 代表 `[bcdefg]`
- `[b-gM-Qkr]` 代表 `[bcdefgMNOPQkr]`
- `ϵ` 可以被省略
- `M?` 代表一个或没有
- `M+` 代表至少一个
- `.` 代表任意一个character（除newline）
- `""` 代表引用，只能一个一个字符

  精准匹配引用里面的内容

REX命名

通过$d_i \to r_i$的方式给r_i命名成d_i，

并且之后可以将d_i视作一个character一样

在REX中使用，比如：

```
digit -> [0-9]
number -> digit+
```

歧义处理

如果所有语句都只对应一个语法树，那么这个文法没有歧义

1. longest match 最长匹配

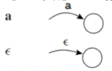
- 实现方式：通过 last final 序列（细节见词法分析 2.3.1）

2. rule priority 规则优先 - 根据规则和规则之间的优先级关系，越前面越高

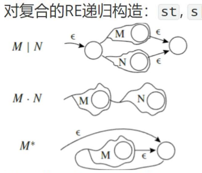
2.3 REX 转 NFA

1. 单个 NFA 的构造

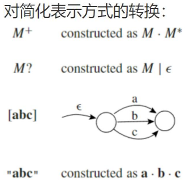
对基本的RE直接构造：ϵ, a



对复合的RE递归构造：`st`, `s|t`, `s*`



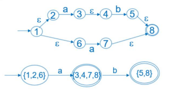
对简化表示方式的转换：



2.4 NFA 转 DFA

1.首先列出所有状态的闭包；
2.将初始状态的闭包作为新的初始状态；
3.计算在每个新状态下在各个字符上的转移的闭包作为新的状态，转移自然成为新的转移；
4.包含原接受状态的所有新状态都是接受状态。

```
states[0] ← {};    states[1] ← closure({s1})
p ← 1;    j ← 0
while j ≤ p
  foreach c ∈ Σ
    e ← DFAedge(states[j], c)
    if e = states[i] for some i ≤ p
      then trans[j, c] ← i
      else p ← p + 1
           states[p] ← e
           trans[j, c] ← p
  j ← j + 1
```



states

j	States
0	{}
1	{1, 2, 6}
2	{3, 4, 7, 8}
3	{5, 8}

trans

	a	b
0	0	0
1	2	0
2	0	3
3	0	0

p	j	c	DFAedge(states[j], c)
1	0	a	{}
1	0	b	{}
1	1	a	{3, 4, 7, 8}
2	1	b	{}
2	2	a	{}
2	2	b	{5, 8}
3	3	a	{}
3	3	b	{}
3	4		

2.5 DFA 最小化

1. 划分等价类

1. **初始划分：接受状态组和非接受状态组 $P = \{S - F, F\}$**
2. **利用"可区分"的概念，反复分裂划分中的组 G**

for (P中的每个元素/集合G) {
  细分G，使得G中的s、t仍然在同一组中 iff
    对任意输入a，s、t都到达P中的同一组；
  P_{new} = 将P中的G替换为细分得到的小组;
}

集合G的每个状态读入同一字符后，都落入相同的某个集合，那么就不用细分S

3. **直到不可再分裂（如果$P_{new} == P$，令$P_{final} = P$，算法完成；否则$P = P_{new}$，转步骤2）**

2. 重建 DFA：从划分得到的等价类中选取代表，并重建 DFA

第二章例题：自然语言到正则语言的转换

a. Strings over the alphabet $\{a, b, c\}$ where the first a precedes the first b.

b. Strings over the alphabet $\{a, b, c\}$ with an even number of a's.

c. Binary numbers that are multiples of four.

d. Binary numbers that are greater than 101001.

e. Strings over the alphabet $\{a, b, c\}$ that don't contain the contiguous substring *baa*.

f. The language of nonnegative integer constants in C, where numbers beginning with 0 are *octal* constants and other numbers are *decimal* constants.

g. Binary numbers n such that there exists an integer solution of $a^n + b^n = c^n$.

a. c*a(a|c)*b(a|b|c)*

b. (b|c)*((b|c)*a(b|c)*a(b|c)*)*

c. 1(0|1)*00

d. (1(0|1)*(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)) | (10101(0|1)) | (1011(0|1)(0|1)) | (11(0|1)(0|1)(0|1)(0|1))

e. (a|c)*((b|(ba)) (c(a|c)*)*)*

f. (0[1-7][0-7]*) | ([1-9][0-9]*) | 00 | 0

g. 1|10

CH3 语法分析

3.1 定义

EOF 的解决：添加一个新的开始符号 S'和一个新的规则 S'->S$

推导和规约：

文法
① $E \to E + E$
② $E \to E * E$
③ $E \to (E)$
④ $E \to \mathrm{id}$

输入
id + (id + id)

最左推导 / 最右归约

推导过程
$E \Rightarrow_{lm} E + E$
$\Rightarrow_{lm} \mathrm{id} + E$
$\Rightarrow_{lm} \mathrm{id} + (E)$
$\Rightarrow_{lm} \mathrm{id} + (E + E)$
$\Rightarrow_{lm} \mathrm{id} + (\mathrm{id} + E)$
$\Rightarrow_{lm} \mathrm{id} + (\mathrm{id} + \mathrm{id})$

句型、句子和语言：

- 句型：起始符号推导出来的任意串，既可以包含终结符，又可以包含非终结符，也可能是空串
- 句子：仅含终结符的句型
- 语言：由文法推导出的所有句子构成的集合

编程语言的文法

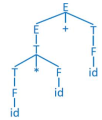
编程语言的文法由于为了语法分析的高效而添加了一定的处理和限制，是任意 CFG 的子集。

消除二义性

"给定 CFG 是否无二义性"是不可判定问题，不存在通用的算法。

通常用于解决这个问题的方式是分层：

- **运算优先级** - 根据算符不同的优先级，引入新的非终结符；越接近开始符号 S 的文法符号优先级越低
- **运算结合性** - 左结合：而全是加法时，左边的加离根节点最远



$E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$ ⇒ $E \to E + T \mid T$, $T \to T * F \mid F$, $F \to (E) \mid \mathrm{id}$

消除左递归

$A \to A\alpha \mid \beta$，其中$\alpha \neq \varepsilon$，α，β不以A开头

$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

把左递归转成了右递归

提左公因子

对形如 $P \to \alpha\beta \mid \alpha\gamma$ 的一对产生式，用如下产生式替换

$P \to \alpha Q$
$Q \to \beta \mid \gamma$

Q为新增加的未出现过的非终结符

3.2 Top-Down - LL(1)

采用最左推导（每次优先替换最左边的非终结符），代表文法是 LL(1)文法。

3.2.1 First、Follow 和 Nullable

Nullable 集 - 一个文法对应单个集合，定义是集合内的每一个元素都可以生成空串

定义：

- Base case：$X \to \epsilon$
- Inductive case：$X \to Y1 \ldots Yn$，

  if $Y1 \ldots Yn$是n个非终结符且都可以生成空串

算法：根据定义递归生成寻找，结束条件是集合不再增长。

First 集 - 一个符号（终结符和非终结符）对应单个集合，定义是可以由这个符号推导得到的串的首终结符号的集合

Base case:

- If X is a terminal: First (X) = {X}

Inductive case:

- If $X \to Y_1 Y_2 \ldots Y_n$
  - First (X) ∪= First(Y_1) 等价于First (X) ⊇ First(Y_1)
  - If Y_1 ∈Nullable, First (X) ∪= First(Y_2)
  - If Y_1,Y_2 ∈Nullable, First (X) ∪= First(Y_3)
  - ……

上述规则似乎是关于非终结符的，但是 First 是关于文法符号串（如产生式右部）的，规则如 inductive case。

Follow 集 - 一个非终结符对应单个集合，定义是从起始符号出发，可能在推导过程中跟在这个符号右边的终结符号的集合

Base case:

- Follow (A) = {}

Inductive case:

- If B → s1 A s2 for any s1 and s2
  1. Follow(A) ∪= First(s2)
  2. If s2 is Nullable, Follow(A) ∪= Follow(B)

对于一些特殊产生式的 follow 集合推论：

Production	Constraints
$T' \to T\$$	$\{\$\} \subseteq FOLLOW(T)$
$T \to R$	$FOLLOW(T) \subseteq FOLLOW(R)$
$T \to aTc$	$\{c\} \subseteq FOLLOW(T)$
$R \to \varepsilon$	
$R \to RbR$	$\{b\} \subseteq FOLLOW(R)$

LL(1)算法的预测分析表

- **行A**：对应一个**非终结符**
- **列a**：对应某个**终结符或输入结束符$**
- **项M(A,a)**：针对非终结符为A，当下一个输入Token为a时，可选的产生式

对文法G的每个产生式$X \to \gamma$

- if t ∈ First(γ): enter ($X \to \gamma$) in row X, col t
- if γ is Nullable and t ∈ Follow(X): enter ($X \to \gamma$) in row X, col t

如果表格内有冲突，则代表其不是 LL(1)文法，我们需要通过消除左递归和提左公因子将其转换为 LL(1)文法。

而保证产生式唯一性的条件如下（对任何两个产生式$A \to \alpha|\beta$）：

1. **First(α) ∩ First(β) = ∅**

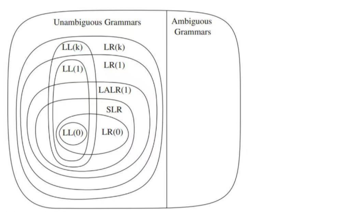
α和β推导不出以同一个单词为首的串

2. **若$\beta \Rightarrow^* \varepsilon$，那么$\alpha \not\Rightarrow^* \varepsilon$，且First(α) ∩ Follow(A) = ∅**

α和β不能同时推出ε; First(α)不应在Follow(A) 中

3.3 Bottom-Up - LR(0)、SLR(1)、LR(1)和 LALR

对应的是最左规约，也就是逆向的最右推导



关注三条线：

- $LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1) \subset LR(k)$
- $LL(0) \subset LL(1) \subset LL(k)$
- $\forall k, LL(k) \subset LR(k)$，但是$LL(1) \not\subset LALR(1)$

3.3.1 LR(0)

状态/项

产生式加一个 • 记录当前识别进度，当 • 在产生式最后的时候是可以使用这个产生式规约的状态。

- 起始状态为$S' \to \cdot S\$$
- 终止状态为$S' \to S \cdot \$$

项集闭包

两个循环，先遍历闭包中每一个 item，再遍历每一个以 item 的 • 后面的 symbol 为起始项的转换关系；直至不再增长（不动点思想）。

Closure(I) =
  repeat
    for any item $A \to \alpha{\bullet}X\beta$ in I
      for any production $X \to \gamma$
        $I \leftarrow I \cup \{X \to \bullet\gamma\}$
  until I does not change.
  return I

转换关系

goto 函数最开始是空集，对于闭包里面每一个项，如果·后面是 X，则将其加入 goto 的结果，并且计算其 closure。

Goto(I, X) =
  set J to the empty set
  for any item $A \to \alpha \bullet X\beta$ in I
    add $A \to \alpha X \bullet \beta$ to J
  return Closure(J)

整体 DFA 构造算法

对每一个状态里面的每一项计算 goto 的结果并加入 DFA 直至不再增长。

Initialize T to {**Closure**({$S' \to \bullet$ S$})}
Initialize E to empty.
repeat
  for each state I in T
    for each item $A \to \alpha \bullet X\beta$ in I
      let J be **Goto**(I, X)
      $T \leftarrow T \cup \{J\}$
      E ← E ∪ {$I \xrightarrow{X} J$}
until E and T did not change

不断增加新状态直到不动点

语法分析表

意义：

以状态为行，以Action的终结符和GOTO的非终结符为列，表项内容为操作：

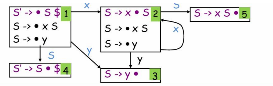
- Action表项：
  - shift+goto下一状态：例如 s2 就是 shift并且去到第二个状态
  - 当状态**存在**以·为结尾的项时（有一个项满足就整体都选择规约），选择index为k的产生式进行reduce
- GOTO 表项：
  - 列是每次规约所用产生式左边的nonterminal A，表示每次规约之后去到的下一个状态是什么

通过 DFA 构造：

1. s 和 g 的区别在于状态转移决定，如果是非终结符就是 g，终结符就是 s，然后当前状态对应行，转移读入的符号是列，去到的状态是 g/s 后面跟的数字；
2. r 就是如果这个状态包括·在最后的产生式，那么使用这个产生式规约，**这一整行的** Action 都加上 r+产生式对应的序号这一项。

例子：

0: S' -> S$
1: S -> x S
2: S -> y



	Action			GOTO
State	x	y	$	S
1	s2	s3		g4
2	s2	s3		g5
3	r2	r2	r2	
4			accept	
5	r1	r1	r1	

Shift: edge labeled with terminal t and from si to sn:
- **T[i, t] = sn** (shift n)

Reduce: Item in a state **i** with dot at the end
- **T[i, each terminal] = rk** (reduce k)
- k is the **index of this production**

Accept: for each state i containing S' -> S . $:
- **T[i, $] = accept**

Goto: edge labeled with non-terminal X and from si to sn:
- **T[i, X] = gn** (goto n)

符号栈与状态栈：

统一状态下先操作符号栈再操作状态栈：

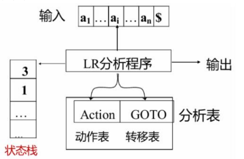
符号栈变化：

- **Shift** - push下一个输入的终结符
- **Reduce** -
  - Top of stack should match RHS of rule (e.g., **X -> A B**)
  - pop the RHS from the top of stack (e.g., **pop B A**)
  - push the LHS onto the stack (e.g., **push X**)
- **Error**
- **Accept** - shift $ 并且可以把栈里面的剩余元素规约成起始符号

状态栈变化：

令s是栈顶状态, a是w$的第一个符号;
while (1) { // 一开始s为Parsing DFA的状态1
  if (Action[s, a] = "shift s' ") {
    将s' 压入栈内；
    让a为下一个输入符号；   // 移进
  } **else if** (Action[s, a] = "reduce A→β ") {
    从栈顶弹出|β|个状态；
    令s'是当前栈顶状态, 把GOTO[s', A]入栈；
  } **else if** (Action[s, a] = "Accept") break;
  else error( ); // 后面会讲;
}

例子：



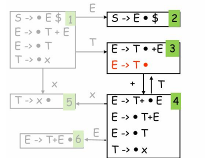
Stack (states)	Stack (symbols)	Input	Action
1		x x y $	shift 2
1 2	x	x y $	shift 2
1 2 2	x x	y $	shift 3
1 2 2 3	x x y	$	reduce 2 (S -> y)
1 2 2 5	x x S	$	reduce 1 (S -> x S)
1 2 5	x S	$	reduce 1 (S -> x S)
1 4	S	$	accept

3.3.2 SLR(1)

在 LR(0)的基础上在规约条件中增加了 next token 是否在产生式左边的 follow 集合内的判断：

0: S -> E$
1: E -> T+E
2: E -> T
3: T -> x

- What are the valid next tokens for r2?
  - **Idea**: we can choose the reduce action only if the next input token **t ∈ Follow(E) ={$}**
  - **Only T[3, $] can be r2!**



	Action			GOTO
State	x	y	$	S
1	s2	s3		g4
2	s2	s3		g5
3	~~r2~~	~~r2~~	r2	
4			accept	
5	r1	r1	r1	

3.3.3 LR(1)

状态/项

定义LR(1)的项为$A \to \alpha \cdot \beta, a$，其中$a$是lookahead的next token（终结符）。起始状态是$S' \to \cdot S\$, ?$，?代表任意终结符。

项集闭包

我们在添加闭包项的时候，应该对item的next token加以判断（产生式和之前是同理的），同时我们要注意每一次增长的时候即使已有这个产生式也要重新计算，因为next token可能不一样。

转换关系

基本相同，next token直接一模一样复制过去就可以。

Closure(I) =   LR(1)
  repeat
    for any item ($A \to \alpha \bullet X\beta$, z) in I
      for any production $X \to \gamma$
        for any $w \in First(\beta z)$
          $I \leftarrow I \cup \{(X \to \bullet\gamma, w)\}$
  until I does not change
  return I

Goto(I, X) =   LR(1)
  $J \leftarrow \{\}$
  for any item ($A \to \alpha \bullet X\beta$, z) in I
    add ($A \to \alpha X \bullet \beta$, z) to J
  return Closure(J)

语法分析表

其他一样，而规约的时候只需要填入next token对应的列即可。

- 如果要根据$A \to \alpha\beta$,a规约，下一个输入符号必须是a

$R \leftarrow \{\}$ LR(1)
for each state I in T
  for each item ($A \to \alpha \bullet$, z) in I
    $R \leftarrow R \cup \{(I, z, A \to \alpha)\}$

Add z in items and reduce actions

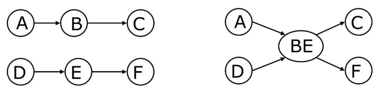
$R \leftarrow \{\}$ SLR(1)
for each state I in T
  for each item $A \to \alpha \bullet$ in I
    for each token X in Follow(A)
      $R \leftarrow R \cup \{(I, X, A \to \alpha)\}$

3.3.4 LALR

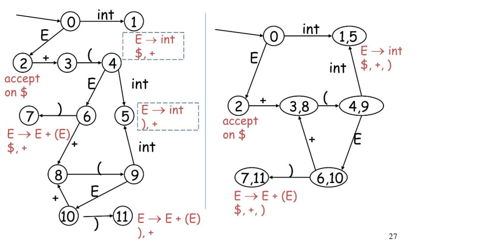
核心思想：合并LR(1)中只有lookahead symbols不同的状态。

合并流程：core（状态中去掉lookahead的部分）保留，lookahead取并集。

合并完成之后同步改变转换edge：

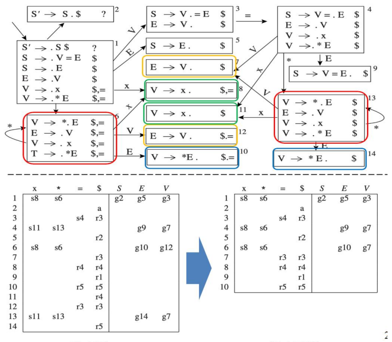


例子：



两个方法的语法分析表例子：

0 S' -> S $
1 S -> V = E
2 S -> E
3 E -> V
4 V -> x
5 V -> * E



(a) LR(1) (b) LALR(1)

CH5 语义分析 - 符号表

5.1 定义

- **Binding** - 维护一个`<Name/Symbol, Meaning/Attribute>`的映射，使用`|->`表示映射

Name/Symbol	Meaning/Attribute
type identifier	type (e.g., int, string)
variable identifier	type, value, access info, …
function identifier	args.&reult type, …

- **Environment** - Binding的集合
  - 环境更新的例子：

```
1 function f(a:int, b:int, c:int) =
2 (print_int(a+c);
3  let var j := a+b
4      var a := "hello"  /* Shadows parameter 'a' */
5  in print(a); print_int(j)
6  end;
7  print_int(b)
8 )
```

Line	Code	Environment Change	Current Environment
1	function f(a:int, b:int, c:int)	Add function parameters	σ1 = σ0 + {a !→ int, b !→ int, c !→ int}
2	print_int(a+c);	Use environment σ1	σ1
3	let var j := a+b	Add variable j	σ2 = σ1 + {j !→ int}
4	var a := "hello"	Add new variable a (shadows parameter)	σ3 = σ2 + {a !→ string}
5	print(a); print_int(j)	Use environment σ3	σ3
6	end;	Exit let scope	σ1
7	print_int(b)	Use environment σ1	σ1
8	)	Exit function scope	σ0

  - 如果有重名的情况，会采取覆盖操作
  - 退出对应scope之后，相应的环境会被丢弃
- **Symbol Table** - 环境的集合

工程上，java 的符号表是这样实现的（个人理解是基于 class 和 package 等嵌套实现）：

```
package M;
class E {
    static int a = 5;
}
class N {
    static int b = 10;
    static int a = E.a + b;
}
class D {
    static int d = E.a + N.a;
}
```

Java

σ1 = { a ↦ *int* }
σ2 = { *E* ↦ σ1 }
σ3= {b ↦ *int* , a ↦ *int* }
σ4= { *N* ↦ σ3 }
σ5= { d ↦ *int* }
σ6= { *D* ↦ σ5 }
σ7 = σ2 +σ4+ σ6

In Java, forward reference is allowed. so *E*, *N*, and *D* are all compiled in the environment σ7. The result of the analysis is {M ↦ σ7}.

5.2 实现方式

需要实现的函数（interface）：

- insert - 插入一个符号并且实现覆盖关系
- lookup - 在符号表中查询当前的符号
- enter - 进入一个scope
- exit - 退出一个scope

实现方式分类：

- 命令式 - 有了新的就看不到老的，但是退出scope的时候还能回得去
- 函数式 - 每次发生变化的时候老的都还保留着，$O(1)$恢复到任意新老状态

命令式 - Imperative Style

实现方式：一个存储所有有效 bindings 的哈希表 ＋ 一个记录哪些符号在哪些作用域的栈（利用作用域 marker）

- 进入新作用域 - 往栈里面推一个marker
- 添加一个符号 - 同时往哈希表里面添加、在栈里面记录
- 退出作用域 - 在哈希表中删除属于最后一个作用域的符号（pop栈直到marker）

命令式实现方式的代码实现：

```python
from dataclasses import dataclass
from typing import TypeVar, Generic, Optional
T = TypeVar("T")

@dataclass
class ScopeMarker:
    """标记作用域的开始"""
    pass

class SymbolTable(Generic[T]):
    def __init__(self):
        # self.stack: 用于跟踪所有绑定和作用域的顺序的列表. 包含两种元素:
        # 1. 字符串标识符 - 表示在当前作用域中添加的变量
        # 2. ScopeMarker对象 - 表示新作用域的开始
        # 元素按添加顺序排列, 最新的在列表末尾
        self.stack = []
        # self.bindings: 存储所有变量绑定的字典
        # 键: 变量标识符(字符串). 值: 该变量所有绑定值的列表, 按作用域从外到内(旧到新)排序
        # 例如: {"x": [1, 2, 3]} 表示x在最外层作用域绑定为1, 中间为2, 当前作用域绑定为3
        self.bindings = {}
```

Stack存标识符和ScopeMarker

正好实现了最内层作用域的变量覆盖外层同名变量的规则

```python
    def add(self, identifier: str, value: T):
        """添加新的绑定到当前作用域"""
        self.stack.append(identifier)
        self.bindings[identifier] = self.bindings.get(identifier, []) + [value]
```

1. 将标识符添加到栈中，记录添加顺序
2. 在bindings中为该标识符添加新的绑定值

```python
    def find(self, identifier: str) -> Optional[T]:
        """查找给定标识符的最新绑定"""
        if identifier in self.bindings and self.bindings[identifier]:
            return self.bindings[identifier][-1]
        return None
```

如果标识符存在且有绑定值，返回最后一个（最新的）绑定

```python
def begin_scope(self):
    """开始一个新的作用域"""
    self.stack.append(ScopeMarker())
```

1. 向栈中添加一个作用域标记
2. 标记将作为当前作用域的边界

```python
def end_scope(self):
    """结束当前作用域, 移除该作用域中的所有绑定

    工作原理是从栈顶开始弹出元素, 直到遇到一个作用域标记(ScopeMarker)
    - 对于每个弹出的标识符, 移除其最新的绑定
    - 最后移除作用域标记本身
    """
    # 处理当前作用域中的所有标识符
    while len(self.stack) > 0 and not isinstance(self.stack[-1], ScopeMarker):
        # 移除该标识符最新(最内层作用域)的绑定
        # bindings[identifier]是一个列表, [-1]索引指向最后添加的值
        identifier = self.stack.pop()
        self.bindings[identifier].pop()

        # 如果该标识符没有剩余绑定, 从字典中删除它
        if not self.bindings[identifier]:
            self.bindings.pop(identifier)

    # 处理完所有标识符后, 如果栈不为空, 弹出作用域标记本身
    # 这表示当前作用域现已完全关闭
    if self.stack:
        self.stack.pop()
```

1. 从栈顶开始，弹出所有标识符，直到遇到作用域标记
2. 对每个弹出的标识符，移除其最新的绑定
3. 如果某个标识符没有剩余绑定，将其从bindings字典中移除
4. 最后弹出作用域标记本身

例子：

```
# 创建符号表实例
st = SymbolTable[int]()

# 步骤1: 全局作用域添加变量
st.add("x", 10)
st.add("y", 20)

# 步骤2: 进入第一层嵌套作用域
st.begin_scope()
st.add("z", 30)
# 在新作用域中重新定义x
st.add("x", 100)

# 步骤3: 退出嵌套作用域
st.end_scope()
```

栈 (Stack)

索引	元素
0	x
1	y
2	ScopeMarker
3	z
4	x

绑定 (Binding)

标识符	值列表 [从外层到内层]
x	[10, 100]
y	[20]
z	[30]

在当前作用域中重新定义变量 x = 100。注意 'x' 再次被压入栈中，而在 bindings 中，'x' 对应的值列表增加了一个新值 100。表示在当前作用域中，x 的值是 100。

```
# 创建符号表实例
st = SymbolTable[int]()

# 步骤1: 全局作用域添加变量
st.add("x", 10)
st.add("y", 20)

# 步骤2: 进入第一层嵌套作用域
st.begin_scope()
st.add("z", 30)
# 在新作用域中重新定义x
st.add("x", 100)

# 步骤3: 退出嵌套作用域
st.end_scope()
```

栈 (Stack)

索引	元素
0	x
1	y

绑定 (Binding)

标识符	值列表 [从外层到内层]
x	[10]
y	[20]

使用 end_scope() 结束嵌套作用域。栈中元素会被弹出直到遇到 ScopeMarker。首先弹出 'x'，移除其值 100；然后弹出 'z'，完全移除；最后弹出 ScopeMarker 本身。

最终状态：嵌套作用域已关闭，只剩下全局作用域中的变量 x = 10 和 y = 20。

函数式 - Functional Style

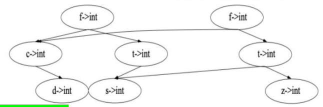
基本思路：

当我们新进入一个作用域的时候，我们会新建一个专属于这个作用域的表格（而不是编辑旧的表格），而退出这个作用域的时候直接删除这个作用域的表格即可，方便快速回退。

实现方式：

通常使用平衡二叉树 BST，排序方式直接采用字符串的大小比较，节点内容是标识符到 binding 的映射。

- 查找就是标准的BST搜索算法，复杂度$O(\log n)$
- 插入是使用了**路径复制**技术："复制从根节点到被插入节点的父节点的所有节点"
  比如插入了`z->int`，就是先创建z这个节点，再复制所有的z的祖先：



函数式与命令式比较：

特性	命令式	函数式
状态管理	可变状态，直接修改数据	不可变状态，创建新实例
数据结构	标准字典/哈希表	持久化数据结构
内存使用	通常较低	较高（由于持久化数据结构）
并发性	需要显式同步	天然线程安全
调试	可能更困难（状态变化）	更容易（不可变状态）

而在这里我们使用字符串作为键值时可能会遇到性能问题：当键值为字符串时，进行查找操作时需要进行昂贵的字符串比较。

解决方案：使用符号（symbol）数据结构，每个符号对象与一个整数值相关联。所有相同字符串的出现都会映射到同一个符号，而不同的字符串则映射到不同的符号。通过将键值设置为符号，可以在查找时进行便宜的整数比较。

历年卷题目

1.（F）当 k 足够大时，LL(k) 可以表示所有 LR(0) 文法能表示的语言。

2.（F）在命令式符号表更新中，$\{a \mapsto \text{int}\} + \{a \mapsto \text{string}\} = \{a \mapsto \text{int}, a \mapsto \text{string}\}$

在命令式符号表更新时如果更改了原有符号的类型，则更新后原来的符号类型不可见。

3. 默认情况下，Yacc/Bison 在遇到移入-归约冲突时，采取的策略是移入。

4. If a grammar is LR(1), but not LALR(1). There may be shift-reduce conflicts in its parsing table of LALR(1). ----- F

因为如果 LR(1) 没有冲突，那么 LALR 合并之后只可能产生 reduce-reduce 冲突

5. 左递归倾向于左结合（因为左边子树离 root 远）

6. The syntax tree will completely reflect the derivation steps for a string. ----- F

语法树只反映了输入字符串的语法结构，而推导步骤则记录了生成这个结构的具体过程。不同的推导过程可能产生相同的语法树，因此语法树并不会完全反映推导步骤。

7. 3) Give the LR(1) item [A →· αB, a/b], we have ________.

(A) {a,b}⊂Follow(A)　(B) {a,b}⊂Follow(B)

(C) {a,b}⊂First(A)　(D) {a,b}⊂First(B)

个人理解这也是 LR(1)不需要额外算 follow 的原因

8. Yacc 使用的文法是 LALR

9. The output of the scanner is token.

10. In the Top-Down Parsing, the action (　) will never be used.

[A] Shift　[B] Match　[C] Generate　[D] Accept

reduce 也没有

11. A grammar is ambiguous if it has two different derivations or two different parse trees for a sentence. ----- F

同 7，一个语法树可以对应多个推导方式（没有说明最左最右推导的情况下），而歧义的定义是多个语法树或多种最左推导或多种最右推导而不是多种推导过程。

12. LR(1) item [A → α . B γ , a],　follow(B)= {　　　}.

[A] a　[B] γ　[C] { γ , a }　[D] { γa }

这个应该是 B

13. Given a legal string of tokens for a CFG, there must be a unique parsing tree to derivate the string. ----- F

14. 4) Given the grammar rules: A'→A　A→ (A) A | ε and the rightmost derivation A'=>A=> (A)A=>(A)(A)A => (A)(A) => (A)()=>()(). _____ is the viable prefix the sentential form (A)(A).　(B)

[A] A)　[B] (A　[C] ((A)A)A　[D] ()

在处理最后那个 ε，所以前缀是(, (A, (A), (A)(, (A)(A, (A)(A)

15. 对于文法：S->AA, A->bA | a，对于字符串：bbbaa

S.No.	Reverse Rightmost Derivation with Handles	Viable Prefix	Comments
1.	S -> bbbaa	b, bb, bbb, bbba	Here, a is the handle so viable prefix cannot exceed beyond a.
2.	S -> bbbAa	b, bb, bbb, bbbA	Here, bA is the handle so viable prefix cannot exceed beyond bA.
3.	S -> bbAa	b, bb, bbA	Here also, bA is the handle so viable prefix cannot exceed beyond bA.
4.	S -> bAa	b, bA	Here also, bA is the handle so viable prefix cannot exceed beyond bA.
5.	S -> Aa	A, Aa	Here, a is the handle so viable prefix cannot exceed beyond a.
6.	S -> AA	A, AA	Here, AA is the handle so viable prefix cannot exceed beyond AA.

16. 1) Given the CFG: E'→E, E→E + n | n, ________ is(are) the viable prefix(es) of the right sentential form 'n+n'.　(ACD

[A] n　[B] n+　[C] E　[D] E+