词法分析的错误恢复

错误恢复策略

- 最简单的错误恢复策略："恐慌模式 (*panic mode*)" 恢复
  - 从剩余的输入中不断删除字符，直到词法分析器能够在剩余输入的开头发现一个正确的字符为止

预测分析中的错误恢复

例

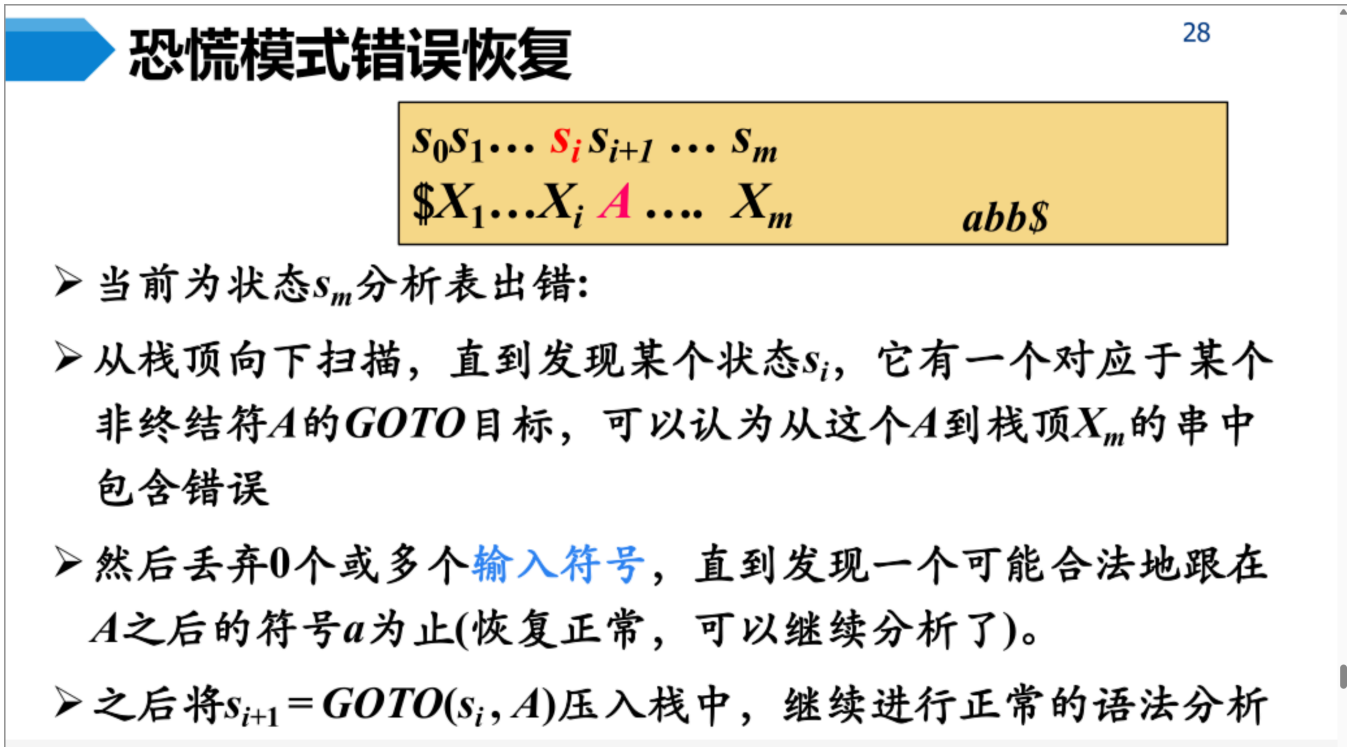
非终结符	输入符号					
	id	+	*	(	)	$
E	*E*→*TE'*			*E*→*TE'*	synch	synch
E'		*E'*→+*TE'*			*E'*→ε	*E'*→ε
T	*T*→*FT'*	synch		*T*→*FT'*	synch	synch
T'		*T'*→ε	*T'*→**FT'*		*T'*→ε	*T'*→ε
F	*F*→id	synch	synch	*F*→(*E*)	synch	synch

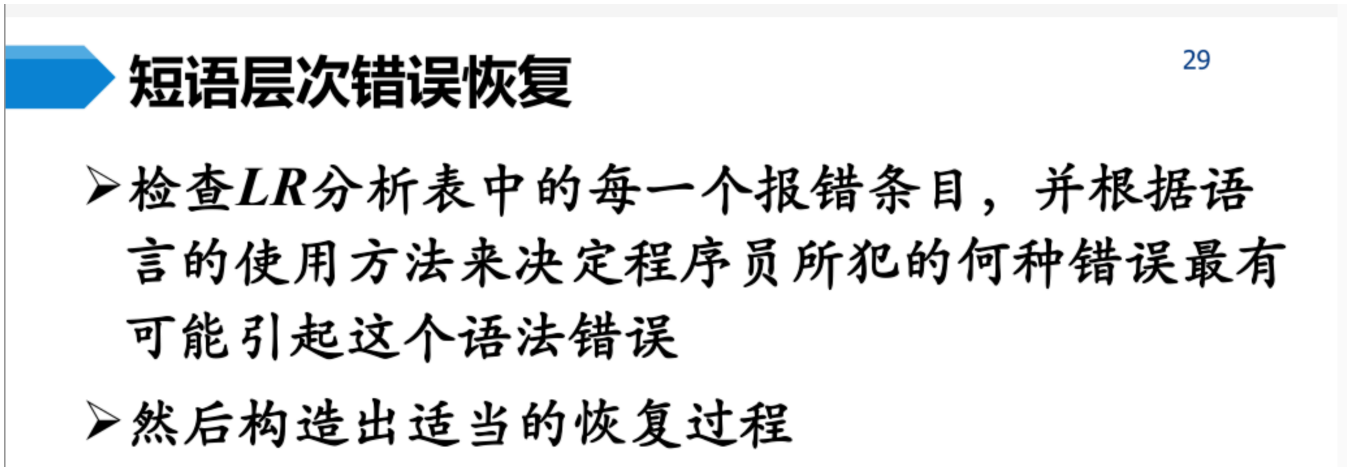
X	FOLLOW(*X*)
E	$)
E'	$)
T	+) $
T'	+) $
F	* +) $

*Synch*表示根据相应非终结符的*FOLLOW*集得到的同步词法单元

- 分析表的使用方法
  - 如果M[*A*,*a*]是空，表示检测到错误，根据恐慌模式，忽略输入符号*a*
  - 如果M[*A*,*a*]是*synch*，则报错，弹出栈顶的非终结符*A*，试图继续分析后面的语法成分
  - 如果栈顶的终结符和输入符号不匹配，则弹出栈顶的终结符

LR分析中的错误恢复





3. 文法定义的语言

形式化定义：

设文法G = (V_t, V_n, S, P)，其中：

- V_t：终结符集（终端符号）
- V_n：非终结符集（非终端符号）
- S：开始符号（S ∈ V_n）
- P：产生式集合

文法G定义的语言L(G)：

```text
L(G) = { w ∈ Vt* | S ⇒+ w }
```

即：由开始符号S经过一步或多步推导，能得到的**所有终结符串**的集合。

语言 L 是某个字母表 Σ 上所有有限长度字符串构成的集合 Σ* 的一个子集。

形式化表示为：

$$L \subseteq \Sigma^*$$

其中：

- **Σ（字母表）**：一个有限的、非空的符号集合
- **Σ*（克林闭包）**：字母表 Σ 上所有可能的有限长度字符串（包括空串 ε）的集合
- **L（语言）**：满足某种特定条件的字符串集合

11. 语法分析方法的分类、核心任务

语法分析概述：

语法分析（解析）是编译器的第二阶段，验证token序列是否符合源语言的语法规则。

核心任务：

1. **语法正确性检查**：验证输入是否属于文法定义的语言
2. **语法结构构建**：生成语法树或推导序列
3. **错误检测与恢复**：发现并报告语法错误，尝试恢复
4. **指导后续分析**：为语义分析、代码生成提供结构信息

语法分析方法分类：

```text
语法分析方法
├── 自顶向下分析 (Top-Down)
│   ├── 递归下降分析 (Recursive Descent)
│   ├── 预测分析 (Predictive Parsing)
│   └── LL分析 (LL(1), LL(k))
│
└── 自底向上分析 (Bottom-Up)
    ├── 算符优先分析 (Operator Precedence)
    └── LR分析
        ├── LR(0)
        ├── SLR(1)
        ├── LALR(1)
        └── LR(1)
```

12. 下推自动机的作用

基本定义：

下推自动机（PDA）是有限自动机（FA）的扩展，增加了一个**栈**作为辅助存储器。

形式定义：PDA = (Q, Σ, Γ, δ, q_0, Z_0, F)

- Q：有限状态集
- Σ：输入字母表
- Γ：栈字母表
- δ：转移函数 Q × (Σ ∪ {ε}) × Γ → 2^(Q × Γ*)
- q_0：初始状态（q_0 ∈ Q）
- Z_0：初始栈符号（Z_0 ∈ Γ）
- F：接受状态集（F ⊆ Q）

PDA的核心作用：

1. 识别上下文无关语言

- **定理**：一个语言是上下文无关的，当且仅当它能被某个PDA接受
- PDA能接受所有CFG生成的语言
- 示例：L = {$a^n b^n$ | n ≥ 0}

2. 实现语法分析

- **自顶向下分析**：用栈保存待匹配的非终结符
- **自底向上分析**：用栈保存已识别的句型前缀

自顶向下PDA模拟：

```
text
```

复制 下载

```
初始：栈=[S$]  # S为开始符号
规则：
1. 栈顶为非终结符A：用产生式右部替换
2. 栈顶为终结符a：与输入匹配，弹出
3. 栈顶为$：检查输入是否结束
```

自底向上PDA模拟（移进-归约）：

```
text
```

复制 下载

```
初始：栈=[$]  # 仅栈底标志
规则：
1. 移进：将输入符号压栈
2. 归约：当栈顶形成句柄β，用A替换（A → β）
3. 接受：栈为[S$]且输入结束
```

13. 动态存储分配的分类

存储分配概述：

程序运行时，数据对象的存储空间分配策略。

分类：

```text
动态存储分配
├── 栈式分配（Stack Allocation）
│   └── 用于局部变量、函数调用
│
└── 堆式分配（Heap Allocation）
    ├── 显式管理（malloc/free，new/delete）
    └── 隐式管理（垃圾回收）
```

详细说明：

1. 栈式分配

原理：后进先出（LIFO）的栈结构
用途：管理函数调用和局部变量

活动记录（AR）结构：

14. 符号表的作用

符号表概述：

符号表是编译器用于记录源程序中标识符（变量、函数、类型等）信息的数据结构。

核心作用：

1. 信息存储

记录每个标识符的属性：

- **名称**：标识符字符串
- **类型**：int、float、array、struct等
- **种类**：变量、常量、函数、类型名
- **作用域**：声明所在作用域
- **存储信息**：地址、偏移量、寄存器
- **维度信息**：数组上下界、结构体成员
- **其他**：行号、是否初始化、访问权限等

2. 快速查找

3. 作用域管理

处理嵌套作用域规则：

- **静态作用域**（词法作用域）：按程序文本嵌套
- **动态作用域**：按调用顺序（较少使用）
- 支持作用域的进入和退出

4. 类型检查支持

- 验证表达式类型兼容性
- 检查函数参数匹配
- 验证赋值相容性

DFA与NFA的详细区别：

特性	DFA	NFA
确定性	确定（唯一下一状态）	非确定（多个可能）
ε转移	不允许	允许
转移函数	δ: Q×Σ → Q	δ: Q×(Σ∪{ε}) → 2^Q
状态数	可能较多	通常较少
实现	简单高效（表驱动）	需回溯或并行模拟
设计难度	较难（需考虑所有情况）	较易（自然）
接受条件	最终状态 ∈ F	存在路径到接受状态