Outline

- ✧ 文法变换**
- ✧ 自底向上语法分析思想
- ✧ 句柄*
- ✧ 移进-归约分析
- ✧ **LR分析***

文法变换

✧ 文法变换：消除左递归、提取左公因子

- LL(1)文法不含左递归和左公因子
- 许多文法在消除左递归和提取左公因子后可以变换为LL(1)文法
- 但不含左递归和左公因子的文法不一定都是LL(1)文法

文法变换：消除左递归

✧ 左递归消除规则

– 消除直接左递归

对形如

$$P \rightarrow P\alpha \mid \beta$$

的产生式，其中α非ε， α、β不以 P 开头，

可改写为：

$$P \rightarrow \beta Q$$

$$Q \rightarrow \alpha Q \mid \varepsilon$$

其中Q为新增加的非终结符

文法变换：消除左递归

✧ 左递归消除规则

– 消除直接左递归

对更一般的形如

$$P \rightarrow P\alpha_1 | P\alpha_2 | \ldots | P\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$$

的一组产生式，其中α_i（$1 \le i \le m$）不为ε，β_j（$1 \le j \le n$）不以 P 打头，

可改写为：

$$P \rightarrow \beta_1 Q | \beta_2 Q | \ldots | \beta_n Q$$

$$Q \rightarrow \alpha_1 Q | \alpha_2 Q | \ldots | \alpha_m Q | \varepsilon$$

其中Q为新增加的非终结符

文法变换：消除左递归

✧ 左递归消除举例

原文法 G[E]:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid a$$

消除左递归后的文法 G'[E]:

(1) $E \rightarrow TE'$ (2) $E' \rightarrow + TE'$ (3) $E' \rightarrow \varepsilon$

(4) $T \rightarrow FT'$ (5) $T' \rightarrow * FT'$ (6) $T' \rightarrow \varepsilon$

(7) $F \rightarrow (E)$ (8) $F \rightarrow a$

文法变换：消除左递归

✧ 左递归消除规则

– 消除一般左递归

对无回路($A \overset{+}{\Rightarrow} A$)、无ε-产生式的文法，通过下列步骤可消除一般左递归（包括直接和间接左递归）：

（1）以某种顺序将文法非终结符排列$A_1, A_2, \ldots, A_n$

（2）for i = 1 , …, n do {

for j=1, …, i-1 do {

用$A_j \to \alpha_1 \mid \alpha_2 \ldots \mid \alpha_k$替换掉形如 $A_i \to A_j \gamma$ 的规则的右部中的首个A_j，得到 $A_i \to \alpha_1 \gamma \mid \alpha_2 \gamma \ldots \mid \alpha_k \gamma$，其中 $A_j \to \alpha_1 \mid \alpha_2 \ldots \mid \alpha_k$ 是关于A_j 的全部产生式;

}

消除A_i规则的直接左递归;

}

（3）化简由（2）得到的文法

文法变换：消除左递归

✧ 左递归消除举例

原文法 G[S]: $S \to PQ \mid a$

$P \to QS \mid b$

$Q \to SP \mid c$

非终结符排序为 S、P、Q，按照消除一般左递归的方法，进行如下变换：

$Q \to SP \mid c$

$\Longrightarrow Q \to PQP \mid aP \mid c$

$\Longrightarrow Q \to QSQP \mid bQP \mid aP \mid c$

$\Longrightarrow Q \to bQPR \mid aPR \mid cR$

$R \to SQPR \mid \varepsilon$

结果：

$S \to PQ \mid a$

$P \to QS \mid b$

$Q \to bQPR \mid aPR \mid cR$

$R \to SQPR \mid \varepsilon$

文法变换：提取左公因子

✧ 提取左公因子规则

– 对形如

$$P \rightarrow \alpha\beta \mid \alpha\gamma$$

的一对产生式，可用如下三个产生式替换：

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

其中Q为新增加的未出现过的非终结符

文法变换：提取左公因子

✧ 提取左公因子规则

– 一般含有左公因子的产生式形如

$$P \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$$

其中，每个 γ 不以α开头.

提取左公共因子，产生式改写成：

$$P \to \alpha Q \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$$

$$Q \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

文法变换：提取左公因子

✧ 提取左公因子举例

– 对文法 **G(S):**

$S \rightarrow \underline{if}\ C\ t\ S \mid \underline{if}\ C\ t\ S\ e\ S \mid d$

$C \rightarrow b$

提取左公因子后，可改写为文法**G'(S)**：

$S \rightarrow \underline{if}\ C\ t\ S\ A \mid d$

$A \rightarrow e\ S \mid \varepsilon$

$C \rightarrow b$

✧ **举例：**许多文法在消除左递归和提取左公因子后可以变换为LL(1)文法

可验证如下文法 **G[E]**是**LL(1)**文法:

(1) E → TE’ (2) E’ → + TE’

(3) E’ → ε (4) T → FT’

(5) T’ → * FT’ (6) T’ → ε

(7) F → (E) (8) F →a

文法变换

✧ 不含左递归和左公因子的文法一定是LL(1)文法吗？

不一定！

S → if C t S
  | if C t S e S
  | d
C → b

	First集	Follow集
S	{ if, d }	{ #,e }
A	{ e, ε }	{ #,e }
C	{ b }	{ t }

提取左公因子后：

S → if C t S A | d
A → e S | ε
C → b

M[A,e] = {A→ e S,
A→ ε }

预测分析中的错误处理

✧ 表驱动LL(1)分析中的错误处理

– 错误报告（error reporting）

- 栈顶的终结符与当前输入符不匹配
- 非终结符*A*于栈顶，面临的输入符为a，但分析表*M*的*M*[*A*,*a*]为空

✧ 预测分析中的错误处理

– 简单的应急恢复（*panic-mode* error recovery）
跳过输入串中的一些符号直至遇到同步符号（*synchronizing* token）为止

同步符号的选择：

- 把 Follow(A) 中的所有符号作为A的同步符号, 跳过输入串中的一些符号直至遇到这些“同步符号”，把A从栈中弹出，可使分析继续
- 把First(A)中的符号加到A的同步符号集，当First(A)中的符号在输入中出现时，可根据A恢复分析

ANTLR (Review)

- Language tool providing a framework to construct
  - recognizers
  - interpreters
  - compilers
  - translators
- Terence Parr, Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. PLDI 2011.

LL(*): The Foundation of the ANTLR Parser Generator

Terence Parr
University of San Francisco
parrt@cs.usfca.edu

Kathleen Fisher*
Tufts University
kfisher@eecs.tufts.edu

Abstract

Despite the power of Parser Expression Grammars (PEGs) and GLR, parsing is not a solved problem. Adding nondeterminism (parser speculation) to traditional *LL* and *LR* parsers can lead to unexpected parse-time behavior and introduces practical issues with error handling, single-step debugging, and side-effecting embedded grammar actions. This paper introduces the *LL(*)* parsing strategy and an associated grammar analysis algorithm that constructs *LL(*)* parsing decisions from ANTLR grammars. At parse-time, decisions gracefully throttle up from conventional fixed $k \geq 1$ lookahead to arbitrary lookahead and, finally, fail over to backtracking depending on the complexity of the parsing decision and the input symbols. *LL(*)* parsing strength reaches into the context-sensitive languages, in some cases beyond what GLR and PEGs can express. By statically removing as

trast, modern computers are so fast that programmer efficiency is now more important. In response to this development, researchers have developed more powerful, but more costly, nondeterministic parsing strategies following both the "bottom-up" approach (*LR*-style parsing) and the "top-down" approach (*LL*-style parsing).

In the "bottom-up" world, *Generalized LR* (GLR) [19] parsers parse in linear to cubic time, depending on how closely the grammar conforms to classic *LR*. GLR essentially "forks" new subparsers to pursue all possible actions emanating from nondeterministic *LR* states, terminating any subparsers that lead to invalid parses. The result is a parse forest with all possible interpretations of the input. *Elkhound* [12] is a very efficient GLR implementation that achieves *yacc*-like parsing speeds when grammars are *LALR*(1). Programmers unfamiliar with *LALR* parsing theory, though,

自底向上分析思想

✧ 语法分析

– 核心问题：句型分析

对任意上下文无关文法$G = (V, T, P, S)$和任意$w \in T^*$，是否有$w \in L(G)$？若成立，则给出分析树或（最右）推导步骤；否则，进行报错处理。

– 三种实现途径

通用分析（Cocke-Younger-Kasami算法）
自顶向下（*top-down*）分析
自底向上（*bottom-up*）分析

自底向上分析思想

✧ 自底向上分析的一般过程

- 从所要分析的终结符串开始进行归约；
- 每一步归约是在当前串中找到与某个产生式的右部相匹配的子串，然后将该子串用这一产生式的左部非终结符进行替换；如果找不到这样的子串，则回退到上一步归约前的状态，选择不同的子串或不同的产生式重试；
- 重复上一步骤，直到归约至文法开始符号；
- 如果不存在任何一个这样的归约，则表明该终结符串存在语法错误

自底向上分析思想

✧ 自底向上分析中的非确定性

- 在每一步归约中，匹配哪一个位置上的子串以及选择哪一个产生式都可能是非确定的
- 这些非确定性导致分析过程会有很高的复杂性

aaab	（ $A \to \varepsilon$, $B \to b$ ）
$\Leftarrow$ aaaAb	（ $A \to aA$, $A \to \varepsilon$, $B \to b$ ）
$\Leftarrow$ aaAb	（ $A \to aA$, $A \to \varepsilon$, $B \to b$ ）
$\Leftarrow$ aAb	（$B \to b$, $A \to aA$, $A \to \varepsilon$ ）
$\Leftarrow$ aAB	（ $A \to aA$, $A \to \varepsilon$, $S \to AB$ ）
$\Leftarrow$ AB	（ $S \to AB$, $A \to \varepsilon$ ）
$\Leftarrow$ S	

✧ 自底向上分析中的确定化

- 最左归约（目标，唯一性/确定化）
- 等价于最右推导（保证了唯一性）
- 分析栈 + 输入串（实现手段：移进-归约分析）

文法 G[S]:

(1) $S \to E$
(2) $E \to E + T$
(3) $E \to T$
(4) $T \to T * F$
(5) $T \to F$
(6) $F \to (E)$
(7) $F \to v$
(8) $F \to d$

可归约串(句柄)

$$S \underset{rm}{\Rightarrow} \underline{E}_{(1)} \underset{rm}{\Rightarrow} \underline{E+T}_{(2)} \underset{rm}{\Rightarrow} \underline{E+T*F}_{(4)} \underset{rm}{\Rightarrow} \underline{E+T*d}_{(8)} \underset{rm}{\Rightarrow} \underline{E+F*d}_{(5)} \underset{rm}{\Rightarrow} \underline{E+v*d}_{(7)} \underset{rm}{\Rightarrow} \underline{T}+v*d_{(3)} \underset{rm}{\Rightarrow} \underline{F}+v*d_{(5)} \underset{rm}{\Rightarrow} \underline{v}+v*d_{(7)}$$

句柄

✧ 改进的方法

– 选择“可归约串”进行归约

在实用的自底向上分析中，总是选择某个“可归约串”进行归约，可大大减少回溯

– 对于文法 $G = (V_N, V_T, P, S)$，以及

$\alpha, \gamma \in (V_N \cup V_T)^*$，$\beta \in V_T^*$

若 $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A \beta$ 且 $A \rightarrow \gamma$，则称

γ 是右句型αγβ 相对于非终结符 A 的句柄

对于一个句型而言，“可归约串” 即句柄

句柄

✧ 举例：句柄

– 对于右边的文法G(S),

文法 G（S）：

（1）S → AB
（2）A → aA
（3）A → ε
（4）B → b
（5）B → bB

aaab 的句柄： ε

aaAb 的句柄： aA

$$S \underset{rm}{\Rightarrow} \underline{AB} \underset{rm}{\Rightarrow} A\underline{b} \underset{rm}{\Rightarrow} \underline{aAb} \underset{rm}{\Rightarrow} aa\underline{Ab} \underset{rm}{\Rightarrow} aa\underline{aAb} \underset{rm}{\Rightarrow} aaab$$

ε

句柄

✧ 举例：句柄一定唯一吗？

– 对于右边的文法G(S),

aaab 的句柄：ε

右句型aaAb的句柄：

aA，aaA

文法 G（S）：

（1）S → AB
（2）A → aA
（3）A → aaA
（4）A → ε
（5）B → b
（6）B → bB

不唯一的原因：
G（S）是二义文法，右句型的最右推导有多个

自底向上分析思想

- 自底向上分析的实现技术
  - 移进–归约（*shift-reduce*）分析技术

    LR分析和算符优先分析
    均采用移进–归约分析技术

- 句柄的作用
  - 作为LR分析中的“可归约串”

    LR分析是**bottom-up parsing**的重点内容

自底向上分析思想

✧ 与自顶向下技术相比

– 功能较强大

原因在于推导和归约过程有如下差别：推导时仅观察可推导出的输入串的一部分，而归约时可归约的输入串整体已全部出现

– 构造较复杂

不适合手工构造

但存在很好的自动构造技术

（如Yacc工具采用的LALR分析技术）

✧ 基本原理

- 确定的自底向上语法分析：最左归约

– 借助于一个确定的下推自动机来实现，包括一个下推栈（分析栈）和一个有限状态控制引擎

下推自动机根据当前状态、下推栈当前状态、剩余输入单词序列来唯一确定如下动作之一，然后进入新状态：

- *Reduce*: 依确定的产生式序列对位于栈顶的可归约串进行归约
- *Shift*: 从输入序列移进一个单词
- *Error*: 发现语法错误，进行错误处理/恢复
- *Accept*：分析成功

移进–归约分析

✧ 移进–归约分析过程的一个例子

文法 G[S]:

(1) $S \rightarrow E$
(2) $E \rightarrow E + T$
(3) $E \rightarrow T$
(4) $T \rightarrow T * F$
(5) $T \rightarrow F$
(6) $F \rightarrow (E)$
(7) $F \rightarrow v$
(8) $F \rightarrow d$

待分析输入串:
v*+*v* * *d

步骤	分析栈 (top →)	余留输入串	动作
(0)		***v*** + ***v*** * ***d*** #	*Shift*
(1)	***v***	+ ***v*** * ***d*** #	*Reduce(7)*
(2)	***F***	+ ***v*** * ***d*** #	*Reduce(5)*
(3)	***T***	+ ***v*** * ***d*** #	*Reduce(3)*
(4)	***E***	+ ***v*** * ***d*** #	*Shift*
(5)	***E*** +	***v*** * ***d*** #	*Shift*
(6)	***E*** + ***v***	* ***d*** #	*Reduce(7)*
(7)	***E*** + ***F***	* ***d*** #	*Reduce(5)*
(8)	***E*** + ***T***	* ***d*** #	*Shift*
(9)	***E*** + ***T*** *	***d*** #	*Shift*
(10)	***E*** + ***T*** * ***d***	#	*Reduce(8)*
(11)	***E*** + ***T*** * ***F***	#	*Reduce(4)*
(12)	***E*** + ***T***	#	*Reduce(2)*
(13)	***E***	#	*Reduce(1)*
(14)	***S***	#	*Accept*

移进—归约分析

✧ 成功分析的结果

- 对应一个最右推导

  将分析栈中的符号串和余留输入串并置，逆向观察每一分析步骤，从步骤（14）至步骤（1）对应一个最右推导

- 格局

  栈内容.余留输入串

- 句柄

  栈顶形成；有效识别

步骤	分析栈	余留输入串	动作
(0)		v + v * d #	.v+v *d
(1)	v	+ v * d #	⇐ v.+v *d
(2)	F	+ v * d #	⇐ F.+v *d
(3)	T	+ v * d #	⇐ T.+v *d
(4)	E	+ v * d #	⇐ E.+ v*d
(5)	E +	v * d #	⇐ E+.v *d
(6)	E + v	* d #	⇐ E+v .*d
(7)	E + F	* d #	⇐ E+F .*d
(8)	E + T	* d #	⇐ E+T .*d
(9)	E + T *	d #	⇐ E+T *.d
(10)	E + T * d	#	⇐ E+T *d.
(11)	E + T * F	#	⇐ E+T *F.
(12)	E + T	#	⇐ E+T.
(13)	E	#	⇐ E.
(14)	S	#	⇐ S.

✧ 成功分析的关键

– 句柄作为“可归约串”

成功的归约对应一个最右推导，因此每一步归约的串必定是一个句柄

对于右边的文法$G[S]$，在当前栈顶出现 $E+T$ 时，没有使用产生式（2）归约，这是因为 $E+T$ 不是句柄

右句型 $E+T*d$ 的唯一一个句柄是d, 于是当d出现在栈顶时使用产生式（8）进行归约

文法 $G[S]$:

(1) $S \rightarrow E$
(2) $E \rightarrow E + T$
(3) $E \rightarrow T$
(4) $T \rightarrow T * F$
(5) $T \rightarrow F$
(6) $F \rightarrow (E)$
(7) $F \rightarrow v$
(8) $F \rightarrow d$

移进–归约分析

✧ 分析过程确定化的关键：解决两类冲突

– 移进–归约（*shift-reduce*）冲突

到达一个不能确定下一步应该移进还是应该归约的状态

例如，有产生式

S → if E then S | if E then S else S

考虑对于如下串进行移进–归约分析

if E then if E then S else S

当 if E then if E then S 出现在分析栈中时，是移进 else，还是归约 if E then S ？

✧ 分析过程确定化的关键：解决两类冲突

– 归约–归约（*reduce-reduce*）冲突

到达这样的状态：有按多个产生式进行归约的选择

例如，有产生式

$$S \rightarrow aA \mid aaA$$

考虑对于串 aaab 进行移进–归约分析

当分析到某一步时，aaA出现在分析栈中（**b** 位于剩余输入区），是用产生式 $S \rightarrow aA$ 归约 aA，还是用产生式 $S \rightarrow aaA$ 归约 aaA？

✧ 表驱动方法

– 借助于移进–归约表

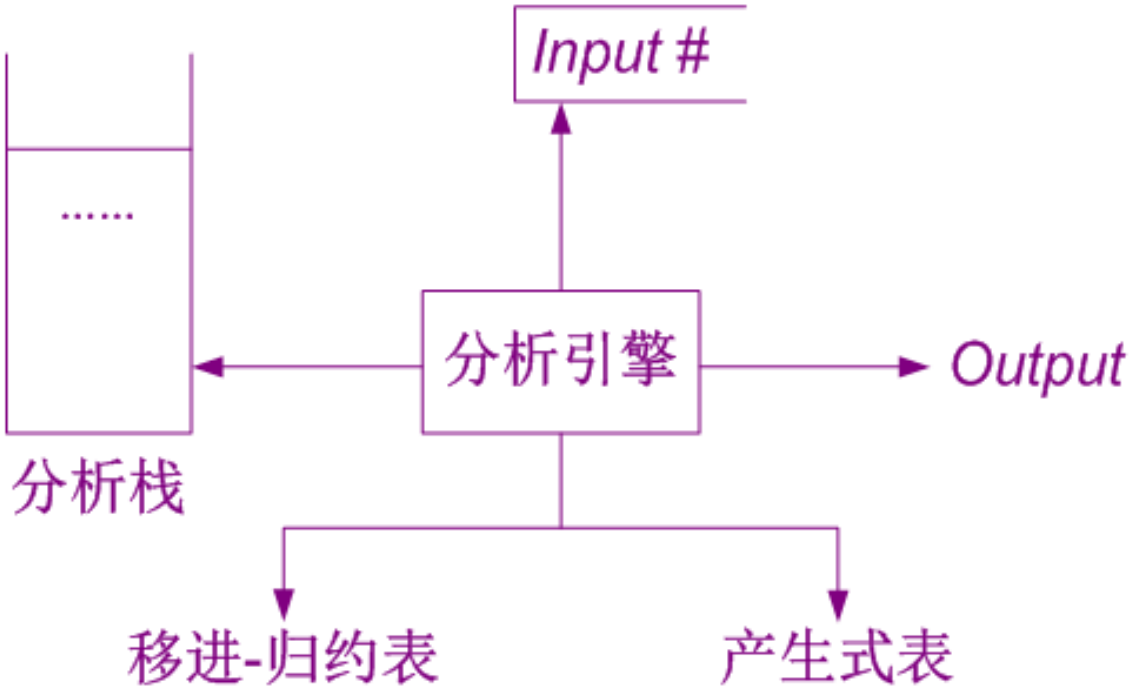
多数移进–归约分析的实现都是基于表驱动方法

下推自动机根据分析栈当前状态*I*内容、剩余输入单词序列，通过查询移进–归约表，确定*Reduce*，*Shift*，*Error*和*Accept* 等动作

移进–归约表可以体现出移进–归约冲突和归约–归约冲突的解决方法

在LR分析中，LR分析表可以起到上述作用

✧ 表驱动移进–归约分析模型



LR分析

- ✧ LR分析基础
- ✧ LR（0）自动机
- ✧ SLR（1）分析
- ✧ LR（1）分析
- ✧ LALR（1）分析
- ✧ LR（K）文法
- ✧ 错误恢复

LR的含义：

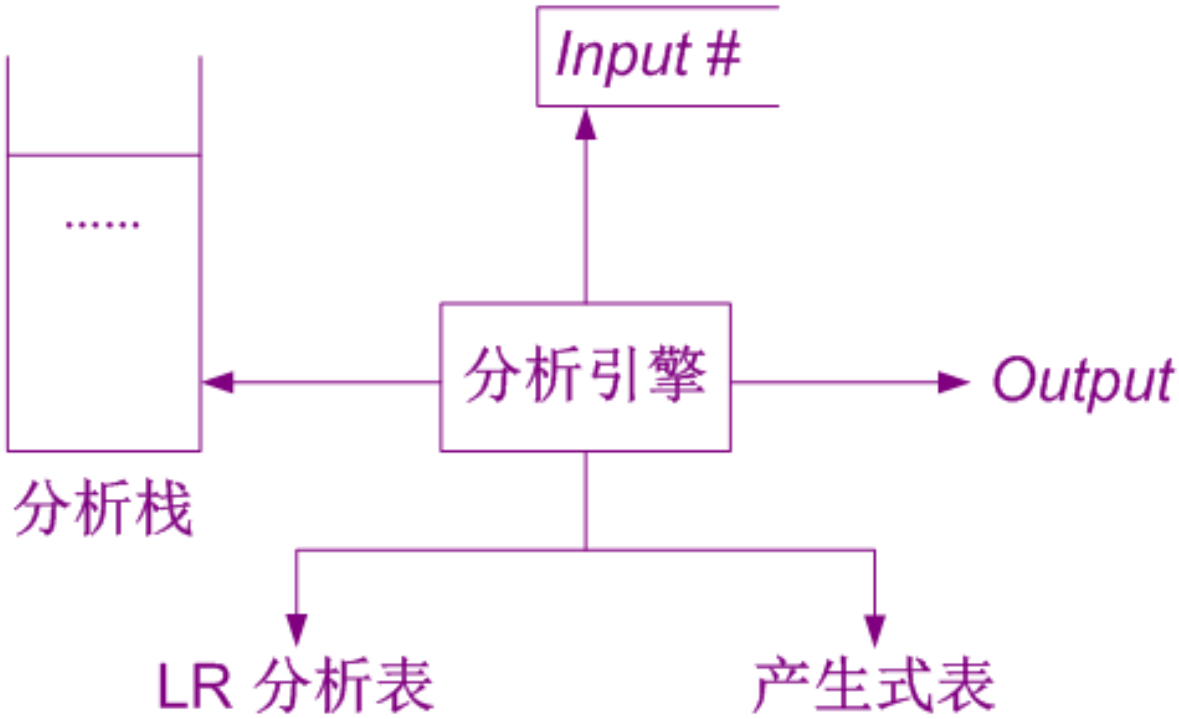
– “L”, 代表从左（Left）向右扫描输入单词序列

– “R”, 代表产生的是最右（Rightmost）推导

LR 分析基础

✧ LR 分析模型

– LR 分析是一种表驱动的移进–归约分析



LR 分析基础

✧ 主要学习四种 LR 分析技术

– LR（0）分析

适用于 LR（0）文法

– SLR（1）分析

适用于 SLR（1）文法

– LR（1）分析

适用于 LR（1）文法

– LALR（1）分析

适用于 LALR（1）文法

"0" – 向前查看 *0* 个符号（见龙书 I）

Simple LR(1)

"1" – 向前查看 *1* 个符号

LookAhead LR(1)

Conclusions

- ✧ 文法变换
- ✧ 错误处理
- ✧ 自底向上语法分析思想
- ✧ 句柄
- ✧ 移进-归约分析
- ✧ **LR分析基础**

推荐教学资料

- § 4.3.3
- § 4.3.4
- § 4.5

Thank you!