编译原理
3. 语法分析-自底向上

rainoftime.github.io
浙江大学
计算机科学与技术学院

课程内容

1. Introduction
2. Lexical Analysis
3. **Parsing**
4. Abstract Syntax
5. Semantic Analysis
6. Activation Record
7. Translating into Intermediate Code
8. Basic Blocks and Traces
9. Instruction Selection
10. Liveness Analysis
11. Register Allocation
13. Garbage Collection
14. Object-oriented Languages
18. Loop Optimizations

回顾: SLR(1)的问题/局限

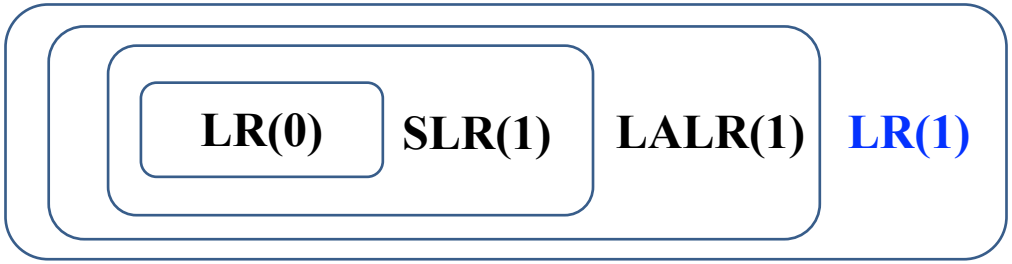
- **SLR技术解决冲突的方法**
  - 按照$A \to \alpha$进行归约的条件是： 下一个输入符号x可以在某个句型中跟在A之后，也就是$x \in Follow(A)$

- **LR分析过程是最右推导的逆**
  - 每一步都应该是最右句型
  - 如果βAx不是任何最右句型的前缀，那么即使x在某个句型中跟在A之后，仍不应该按$A \to \alpha$归约
  - (利用Follow集可看作"可能不够精确的近似")

进行归约的条件更加严格才能进一步降低冲突的可能

本讲内容

1. LR(1) 分析
2. LALR(1) 分析
3. 语法分析器的生成器
4. 错误恢复
5. 语法分析小结



1. LR(1)分析

LR(1)项

- LR(1)项中包含更多信息来消除一些归约动作
  - 实际的做法相当于“分裂”一些LR(0)状态，精确指明何时应该归约

- LR(1)项的形式 $A \rightarrow \alpha \cdot \beta, a$
  - a称为**向前看符号**，可以是终结符号或者$
  - $A \rightarrow \alpha \cdot \beta, a$ 表示 sequence α is on top of the (symbol) stack, and at the head of the input is a string derivable from βa.

LR(1) Parsing: Closure

LR(1)

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

LR(0)

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

- 处理ε-transactions(也就是添加$X \rightarrow \bullet\ \gamma$) 时，记录 w
  - $w \in$ First(βz) : 把$A \rightarrow \alpha \bullet X\beta$, z的信息“传递”到$X \rightarrow \bullet\ \gamma$, w
- 起始状态: LR(1)项 $S' \rightarrow \bullet\ S$ \$, ? 的闭包
  - ”?” 是什么无关紧要，因为\$不会被移进

LR(1) Parsing: Goto

- 和LR(0)项集的Goto算法基本相同

LR(1)

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

LR(0)

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

移入动作不考虑a(a传递到下一状态)

LR(1)的Reduce Action

LR(1)

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

SLR(1)

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

Add z in items and reduce actions

- The action (I, z, A → α) indicates that in state I, on lookahead symbol z, the parser will reduce by rule A → α.
  - The lookahead token of LR(1) is more accurate than that of SLR.
  - Some tokens in Follow(A) may not always be valid in the states with $A \rightarrow \alpha$.

如果要根据A → αβ •, a 归约，下一个输入符号**必须**是a

例: LR(1) Parsing

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

S' -> . S $?

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

例: LR(1) Parsing

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

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

S' -> . S $	?
S -> . V = E	$
S -> . E	$

例: LR(1) Parsing

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

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

S' -> . S $	?
S -> . V = E	$
S -> . E	$
V -> . x	=
V -> . * E	=

例: LR(1) Parsing

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

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

S' -> . S $	?
S -> . V = E	$
S -> . E	$
V -> . x	=
V -> . * E	=
E -> . V	$

例: LR(1) Parsing

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

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

S' -> . S $	?
S -> . V = E	$
S -> . E	$
V -> . x	=
V -> . * E	=
E -> . V	$
V -> . x	$
V -> . * E	$

例: LR(1) Parsing

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

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

S' -> . S $	?
S -> . V = E	$
S -> . E	$
V -> . x	=
V -> . * E	=
E -> . V	$
V -> . x	$
V -> . * E	$

例: LR(1) Parsing

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

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

S' -> . S $	?
S -> . V = E	$
S -> . E	$
V -> . x	=
V -> . * E	=
E -> . V	$
V -> . x	$
V -> . * E	$

S' -> . S $	?
S -> . V = E	$
S -> . E	$
V -> . x	$, =
V -> . * E	$, =
E -> . V	$

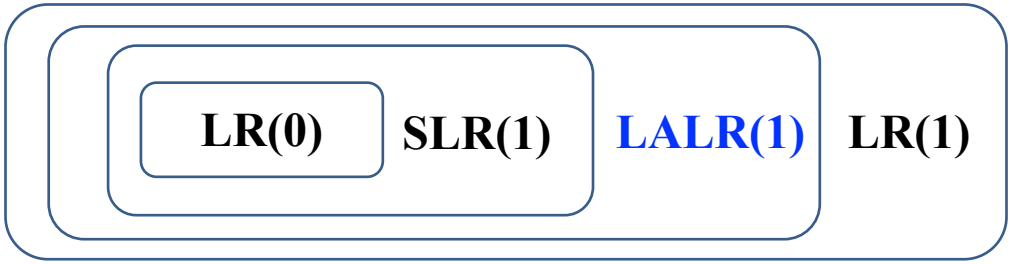
LR(1) Parsing的局限

- LR(1) parsing tables can be very large, with many states.

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

	x	*	=	$	*S*	*E*	*V*
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

(a) LR(1)

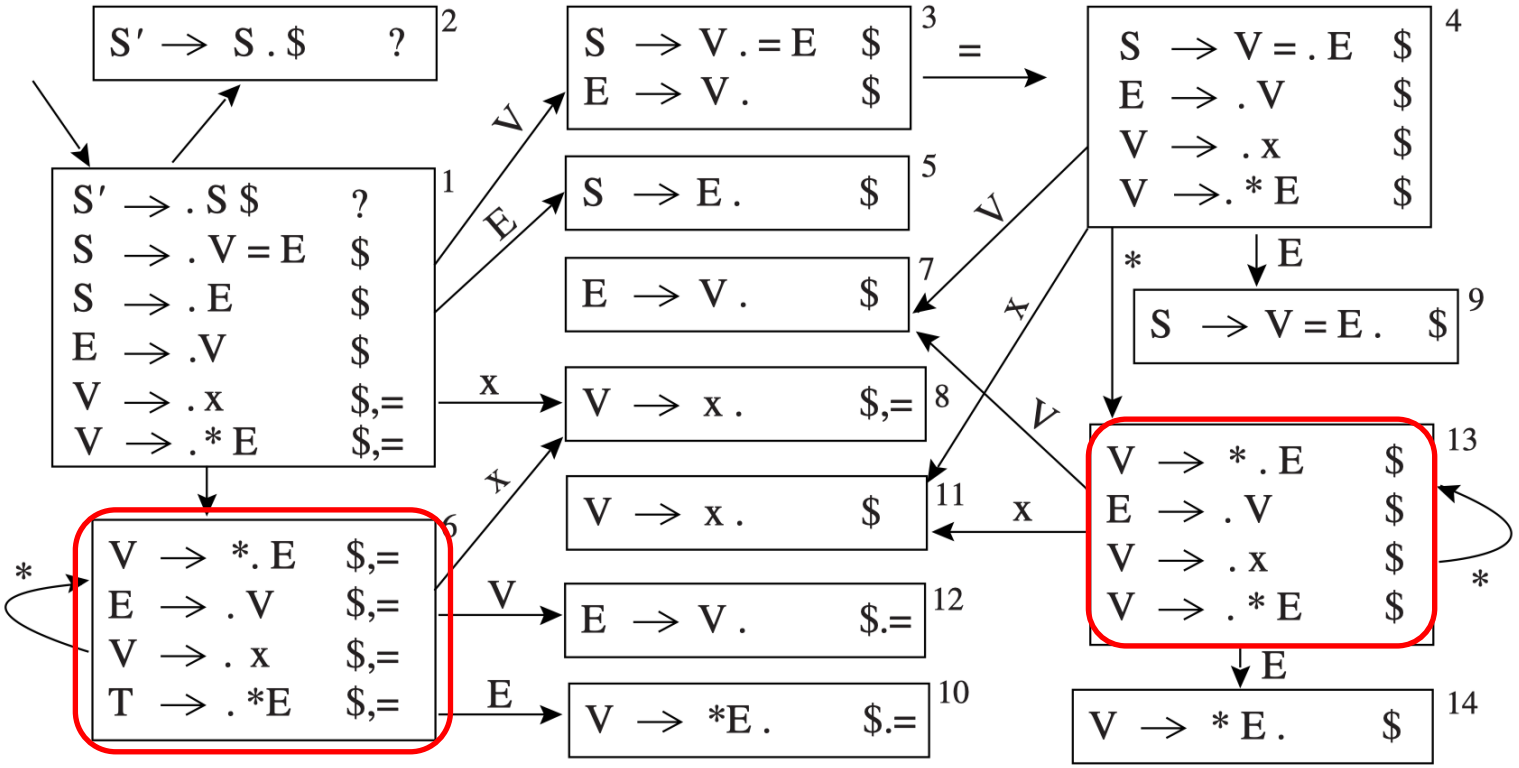


5. LALR(1)

Key Observation behind LALR

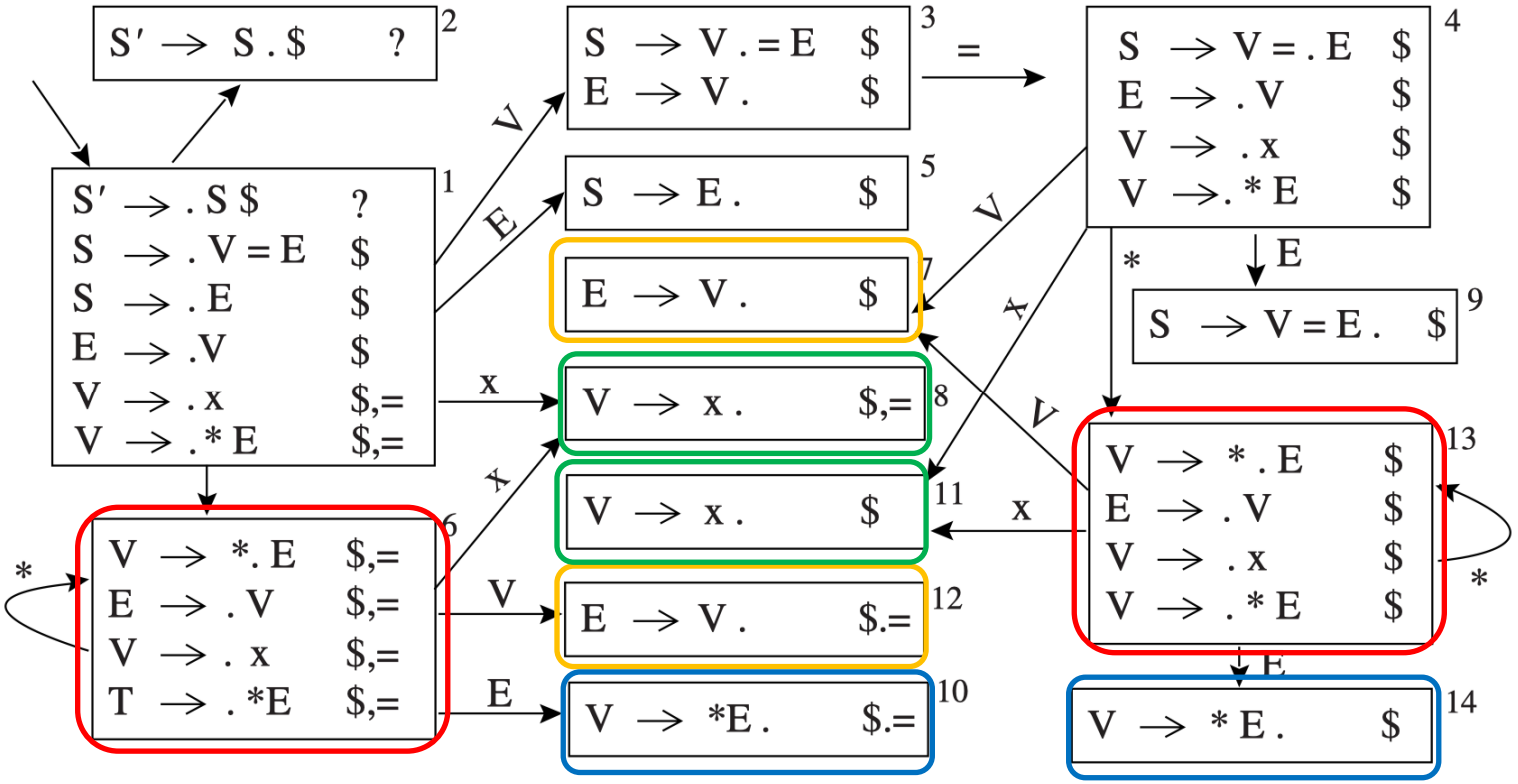
- Some states in LR(1) are only different on the lookahead symbols

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



LALR(1) Parsing in a Nutshell

- LALR(1) parsing: the parsing table is made by merging any two states whose items are identical except for lookahead sets in the LR(1) parsing table.



The Core of a Set of LR(1) Items

- Definition: The core of a set of LR items is the set of first components
  - Without the lookahead terminals

- Example: the core of

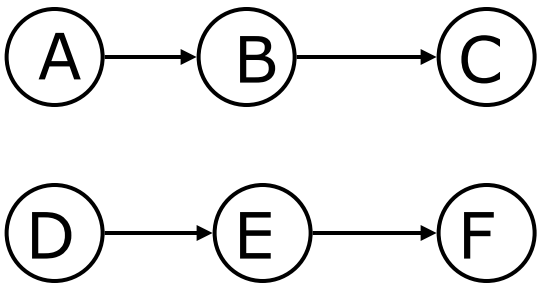
  $\{ X \rightarrow \alpha \bullet \beta, b, Y \rightarrow \gamma \bullet \delta, d \}$

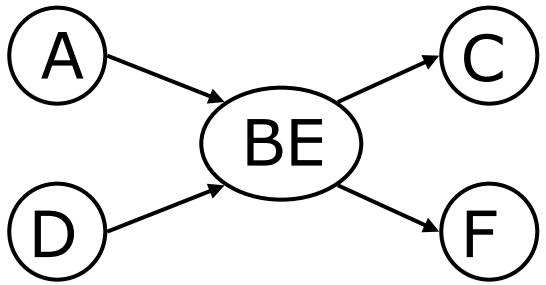
  is

  $\{ X \rightarrow \alpha \bullet \beta, \ Y \rightarrow \gamma \bullet \delta \}$

Building LALR(1) Parsing DFA

- Repeat until all states have distinct core
  - Choose two distinct states with same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from predecessors to new state; new state points to all the previous successors





例: 从LR(1) 到LALR(1)

	x	*	=	$	*S*	*E*	*V*
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

(a) LR(1)

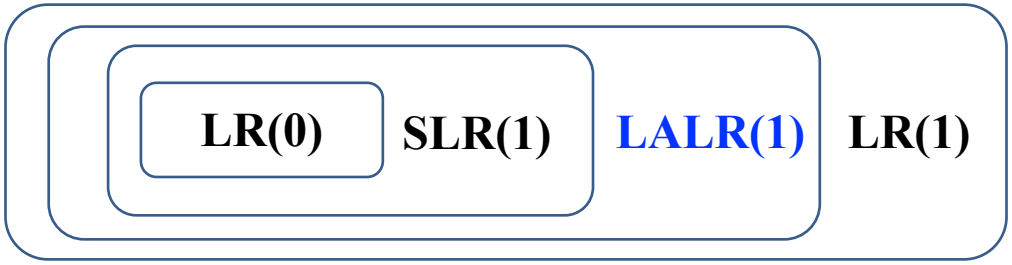
	x	*	=	$	*S*	*E*	*V*
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s8	s6				g9	g7
5				r2			
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

(b) LALR(1)

- LALR(1) parsing table requires less memory to represent
- Typically 10 times fewer LALR(1) states than LR(1)

LALR(1) vs. LR(1) Parsing

- **LR(1)**
  - 把期望的向前看符号也加入项中成为**LR(1)项**
  - 向前看符号(串)的长度即为LR(k)中的k
  - 充分利用向前看符号，但是状态很多
- **LALR(1)**
  - 介于SLR(1)和LR(1)之间，且分析表和SLR一样大
  - LALR已经可以处理大部分的程序设计语言

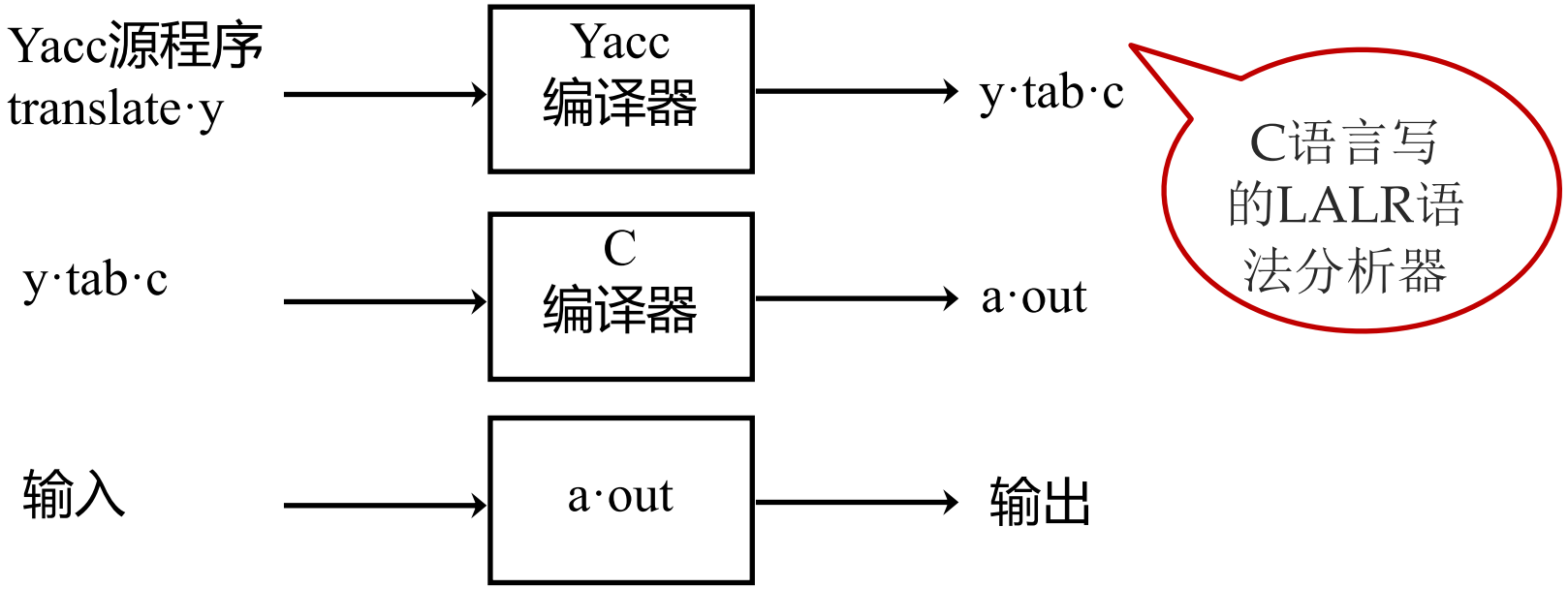


3. 语法分析器的生成器

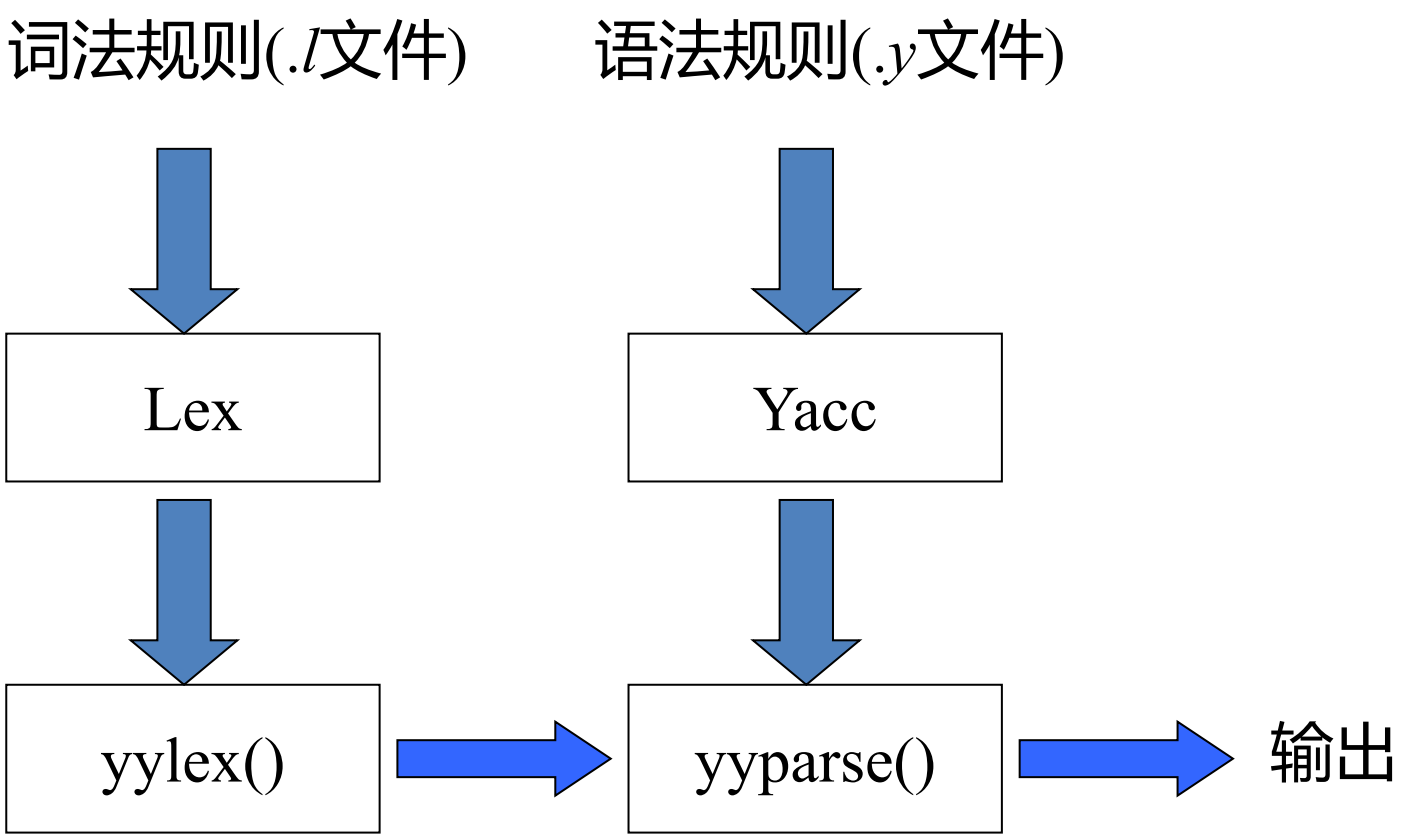
语法分析器的生成器Yacc

- **Yacc：yet another compiler-compiler**
  - 基于LALR(1), 用BNF(Backus Naur Form)形式书写
- **Yacc 的 GNU 版叫做 Bison**

yacc <options> <filename ending with ·y>



Lex和Yacc的联系



YACC源程序的结构

- **声明**
  - 放置C声明和对词法单元的声明
- **翻译规则**
  - 指明产生式及相关的语义动作
- **辅助性C语言例程**
  - 被直接拷贝到生成的C语言源程序中
  - 可在语义动作中调用
  - 包括yylex()，这个函数返回词法单元，可以由Lex生成

```
声明
%%
翻译规则
%%
辅助性C语言例程
```

例: Yacc使用

exp → *exp addop term* | *term*

addop → + | -

term → *term mulop factor* | *factor*

mulop → *

factor → *(exp)* | **number**

```
%{
#include <stdio.h>
#include <ctype.h>
int yylex(void);
int yyerror (char * s);
%}
%token NUMBER
%%
command: exp {printf("%d\n", $1);};
exp: exp '+' term {$$ = $1 + $3;}
   | exp '-' term {$$ = $1 - $3;}
   | term {$$ = $1}
;
term: term '*' factor {$$ = $1 * $3;}
    | factor {$$ = $1;}
;
factor: NUMBER {$$ = $1;}
      | '(' exp ')' {$$ = $2;}
;
```

Rules:

- Rule {Action Code}
- Action code will be executed after the parse performs a reduce action using the rule.

语义动作: `$$ = $1 + $3`

- $$表示和产生式头相关的属性值，$*i*表示产生式体中第*i*个文法符号的属性值

Yacc文件格式中的几个问题

- **消除二义性**: 为算符指定优先级与结合律

```
%left  '-'  '+'
%left  '*'  '/'
%right UMINUS      /* negation--unary minus */
%right  '^'        /* exponentiation        */
```

- **冲突解决**
  - 归约/归约冲突：选择Yacc说明中先出现的产生式

    $A \to \alpha\,.\quad B \to \alpha\,.$
  - 移进/归约冲突：移近优先

I_4: $S \to iS \cdot eS$

$S \to iS \cdot$

更通用的方法：改写文法以消除冲突
(例如, 消除二义性的同时也可能减少了冲突)

4. 错误恢复

Error Recovery

- Motivation: developers would like to have all the errors in her program reported, not just the first error.

- Error recovery techniques:
  - **Local error recovery**

    Adjust the parse stack and the input at the point where the error was detected in a way that will allow parsing to resume.
  - **Global error repair**

    Find the smallest set of insertions and deletions that would turn the source string into a syntactically correct string

Local Error Recovery

- One local recovery mechanism in Yacc
  - Use a special *error* symbol to control the recovery process

$exp \rightarrow ID$

$exp \rightarrow exp + exp$

$exp \rightarrow (exps)$

$exps \rightarrow exp$

$exps \rightarrow exps ; exp$

$exp \rightarrow (error)$

$exps \rightarrow error ; exp$

- If a syntax error is encountered in the middle of an expression, the parser should skip to the next semicolon or right parenthesis and resume parsing.

these are called synchronizing tokens

Local Error Recovery

***exp* → (*error*)**
exps* → *error* ; *exp

error is considered a terminal symbol

- **When the LR parser reaches an error state, it takes the following actions:**

1. Pop the stack (if necessary) until a state is reached in which the action for the *error* token is *shift*.
2. Shift the *error* token.
3. Discard input symbols (if necessary) until a lookahead is reached that has a nonerror action in the current state.
4. Resume normal parsing.

例: Local Error Recovery

exp → *ID*

exp → *exp* + *exp*

exp → (*exps*)

exps → *exp*

exps → *exps* ; *exp*

exp → (*error*)

exps → *error* ; *exp*

input tokens: (ID++)$

Stack (symbol)	Input	Action
	(ID++)$	shift
(	ID++)$	shift
(ID	++)$	reduce
(exp	++)$	shift

例: Local Error Recovery

exp → ID

exp → exp + exp

exp → (exps)

exps → exp

exps → exps ; exp

exp → (error)

exps → error ; exp

input tokens: (ID++)$

Stack (symbol)	Input	Action
	(ID++)$	shift
(	ID++)$	shift
(ID	++)$	reduce
(exp	++)$	shift
(exp+	+)$	error

例: Local Error Recovery

exp → *ID*
exp → *exp* + *exp*
exp → (*exps*)
exps → *exp*
exps → *exps* ; *exp*
exp → (*error*)
exps → *error* ; *exp*

input tokens: (ID++)$

Stack (symbol)	Input	Action
	(ID++)$	shift
(	ID++)$	shift
(ID	++)$	reduce
(exp	++)$	shift
(exp+	+)$	error
(exp	error+)$	pop

- Pop the stack (if necessary) until a state is reached in which the action for the *error* token is *shift*.

例: Local Error Recovery

exp → *ID*
exp → *exp* + *exp*
exp → (*exps*)
exps → *exp*
exps → *exps* ; *exp*
exp → (*error*)
exps → *error* ; *exp*

input tokens: (ID++)$

Stack (symbol)	Input	Action
	(ID++)$	shift
(	ID++)$	shift
(ID	++)$	reduce
(exp	++)$	shift
(exp+	+)$	error
(exp	error+)$	pop
(	error+)$	shift

- Pop the stack (if necessary) until a state is reached in which the action for the *error* token is *shift*.

例: Local Error Recovery

exp → *ID*

exp → *exp* + *exp*

exp → (*exps*)

exps → *exp*

exps → *exps* ; *exp*

exp → (*error*)

exps → *error* ; *exp*

input tokens: (ID++)$

Stack (symbol)	Input	Action
	(ID++)$	shift
(	ID++)$	shift
(ID	++)$	reduce
(exp	++)$	shift
(exp+	+)$	error
(exp	error+)$	pop
(	error+)$	shift
(error	+)$	discard input

Discard input symbols (if necessary) until a lookahead is reached that has a nonerror action in the current state.

例: Local Error Recovery

exp → *ID*

exp → *exp* + *exp*

exp → (*exps*)

exps → *exp*

exps → *exps* ; *exp*

exp → (*error*)

exps → *error* ; *exp*

input tokens: (ID++)$

Stack (symbol)	Input	Action
	(ID++)$	shift
(	ID++)$	shift
(ID	++)$	reduce
(exp	++)$	shift
(exp+	+)$	error
(exp	error+)$	pop
(	error+)$	shift
(error	+)$	discard input
(error	)$	shift

Discard input symbols (if necessary) until a lookahead is reached that has a nonerror action in the current state.

例: Local Error Recovery

exp → *ID*

exp → *exp* + *exp*

exp → (*exps*)

exps → *exp*

exps → *exps* ; *exp*

exp → (*error*)

exps → *error* ; *exp*

input tokens: (ID++)$

Stack (symbol)	Input	Action
	(ID++)$	shift
(	ID++)$	shift
(ID	++)$	reduce
(exp	++)$	shift
(exp+	+)$	error
(exp	error+)$	pop
(	error+)$	shift
(error	+)$	discard input
(error	)$	shift
(error)	$	reduce
exp	$	

小结: Local Error Recovery

- 使用错误产生式来完成语法错误恢复
  - 错误产生式$A \rightarrow \text{error } \alpha$
  - 例如：$stmt \rightarrow \text{error ;}$
- 定义哪些非终结符号有错误恢复动作
  - 比如：表达式、语句、块、函数定义等非终结符号
- 当语法分析器遇到错误时
  - 不断弹出栈中状态，直到栈顶状态包含项$A \rightarrow \cdot \text{error } \alpha$
  - 分析器将error移入栈中
  - 如果α为空，分析器直接执行归约，并调用相关的语义动作；否则跳过一些符号，找到可以归约为α的串为止

Global Error Recovery

- **Global error repair :** finds the smallest set of insertions and deletions that would turn the source string into a syntactically correct string, *even if the insertions and deletions are not at a point where an LL or LR parser would first report an error*.

- **Burke-Fisher error repair :** single-token insertion, deletion, or replacement at every point that occurs no earlier than *K* tokens before the point where the parser reported the error.
  - ✓ The LL(*k*) or LR(*k*) (or LALR, etc.) grammar is not modified at all (no *error* productions)
  - ✓ Nor are the parsing tables modified.

5. 语法分析小结

SLR和LR(1)分析对比

		SLR	LR(1)
动作	移进	$A \to \alpha \cdot a\beta \in I_i$ $Goto(I_i, a) = I_j$ $Action[i, a] = sj$	$A \to \alpha \cdot a\beta, b \in I_i$ $Goto(I_i, a) = I_j$ $Action[i, a] = sj$
	归约	$A \to \alpha \cdot \in I_i$, $a \in Follow(A)$ $Action[i, a] = rj$	$A \to \alpha \cdot, a \in I_i$ $Action[i, a] = rj$

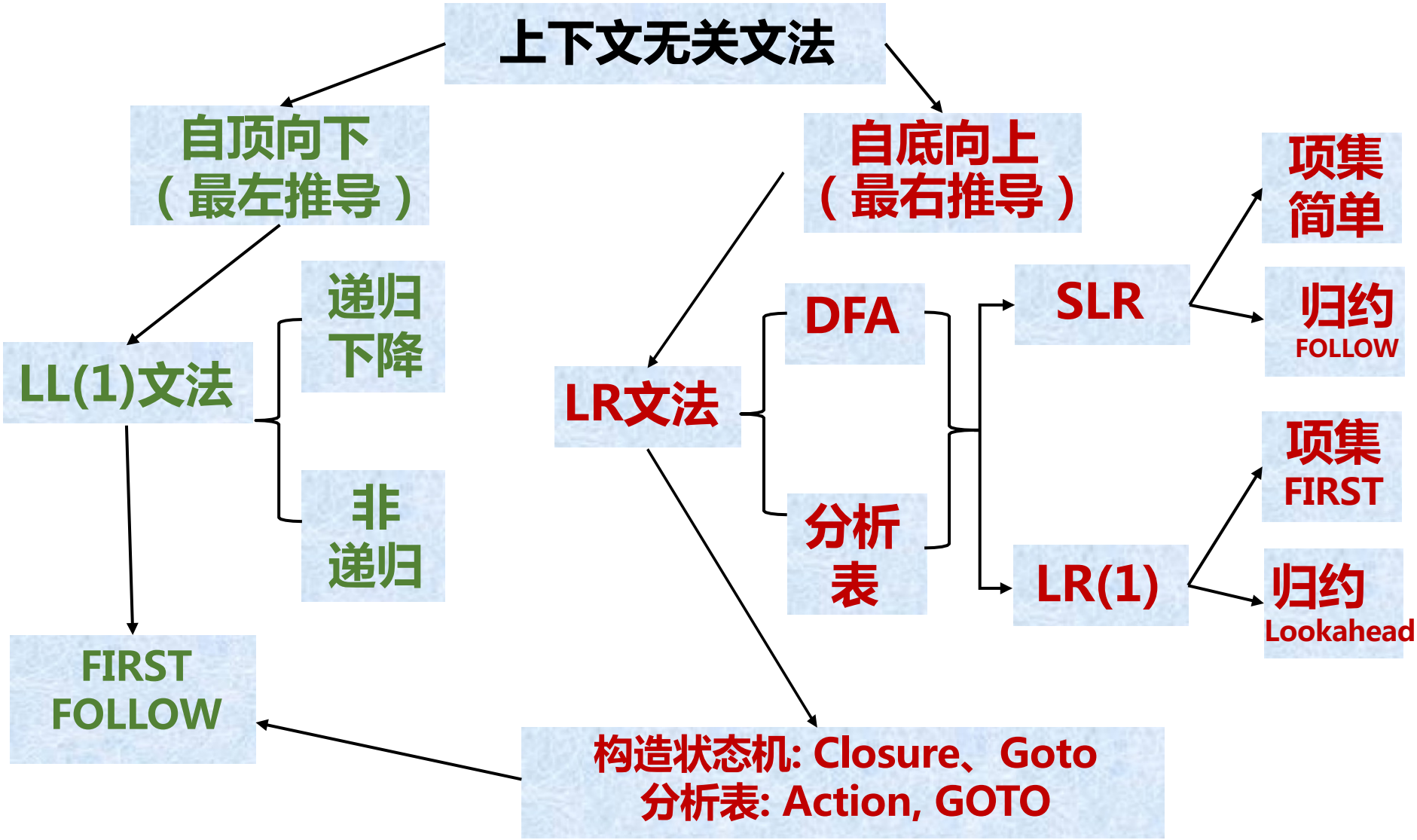
LL(1)和LR(1)分析对比

	LR(1)方 法	LL(1)方 法
建立分析树	自底而上	自顶而下
归约or推导	规范归约(最右推导的逆)	最左推导
分析表	状态×文法符号，大	非终结符×终结符，小
分析栈	状态栈，信息更多	文法符号栈

- **LL(1):** 向前看下一个输入根据First, Follow确定使用哪条产生式推导； $A \to \alpha_1 | \alpha_2 | \alpha_3$ ，
- **LR(1):** 在识别出整个α后，再往前看1个符号，然后确定使用哪条产生式归约。

$$A \to \alpha \quad B \to \alpha$$

LL(1), SLR和LR(1)对比



Thank you all for your attention

Error Recovery: Global Error Repair

typec → *type type-id = ty*

vardec → *var id := exp*

→ *var id : type-id := exp*

ty → *type-id*

→ *{ tyfields }*

→ *array of type-id*

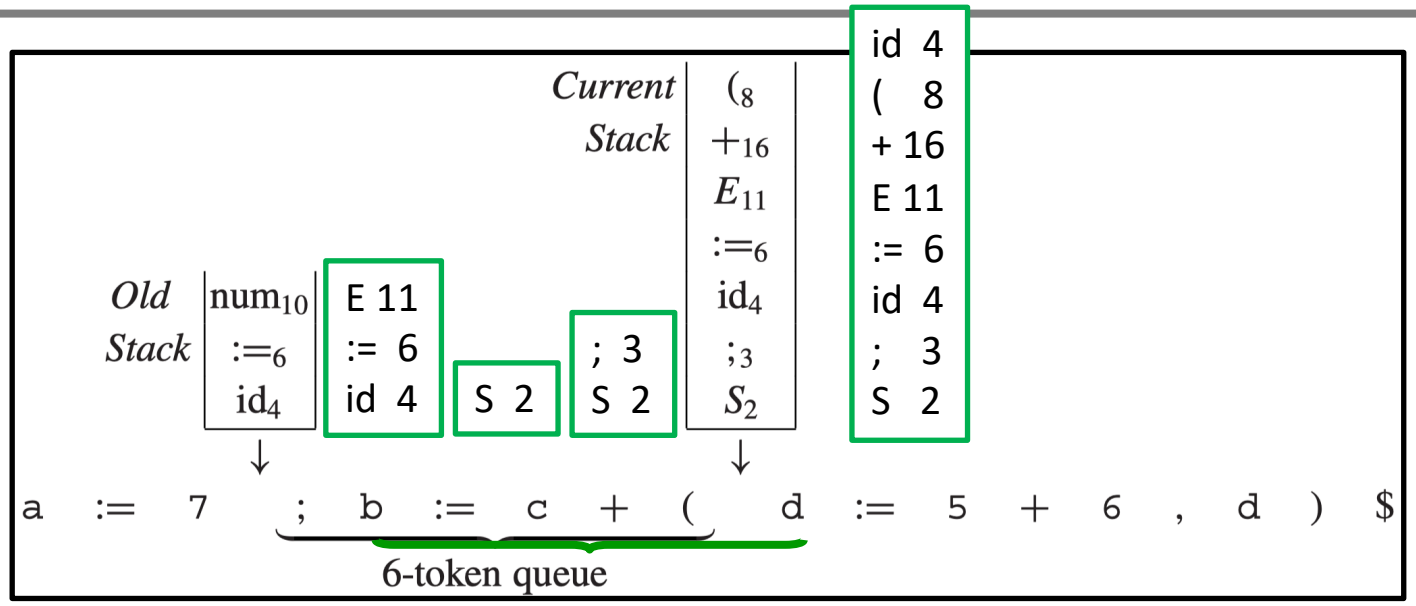
let type a := intArray [10] of 0 in . . .

- How does local error recovery techniques recover this error?
- A local technique will discover a syntax error with := as lookahead symbol:
  - delete the phrase from type to 0
  - or, replace the := by =, and will encounter another syntax error at the [token
- *Global error repair* finds the smallest set of insertions and deletions that would turn the source string into a syntactically correct string, *even if the insertions and deletions are not at a point where an LL or LR parser would first report an error.*

Error Recovery: Burke-Fisher Error Repair

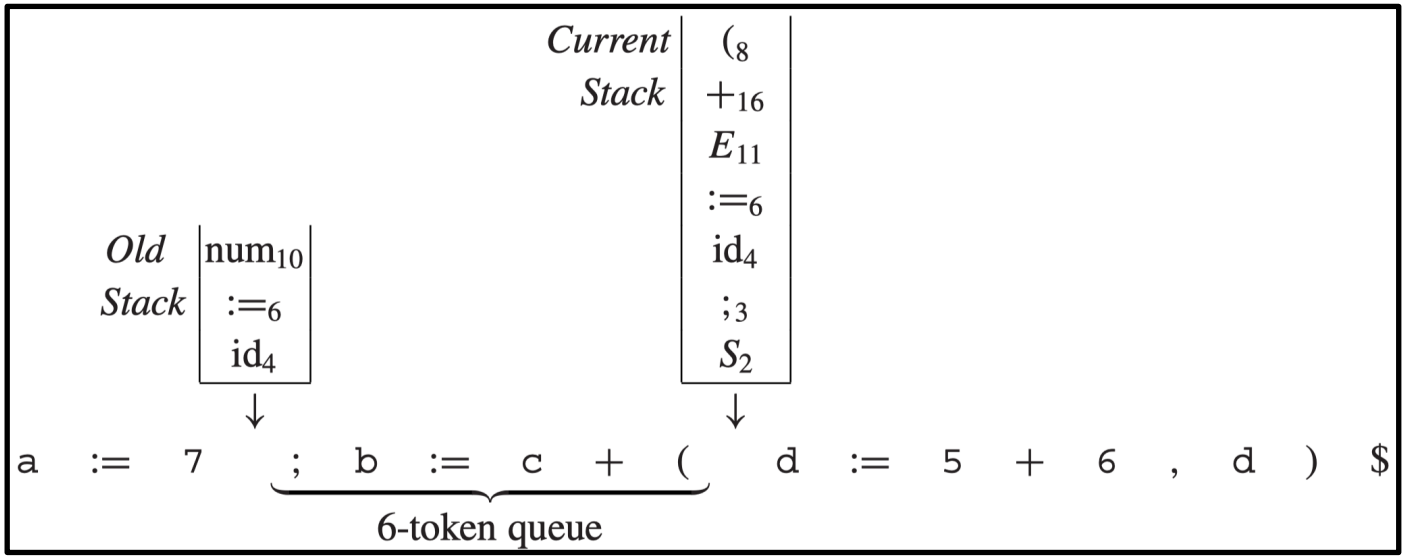
- Burke-Fisher error repair: tries every possible single-token insertion, deletion, or replacement at every point that occurs no earlier than K tokens before the point where the parser reported the error:
  - limited but useful
- How to choose the best error repair?
  - The correction that allows the parser to parse furthest past the original reported error.
  - Generally, if a repair carries the parser R = 4 tokens beyond where it originally got stuck, this is "good enough."
- The parsing engine must be able to back up K tokens and reparse.
- How to implement this technique?

Error Recovery: Burke-Fisher Error Repair



- Maintain two parse stacks: the current stack and the old stack.
- Keep a queue of K tokens
- As each new token is shifted:
  - It is pushed on the current stack and put onto the tail of the queue
  - The head of the queue is removed and shifted onto the old stack
- With each shift onto the old or current stack, the appropriate reduce actions are also performed.

Error Recovery: Burke-Fisher Error Repair



- Suppose a syntax error is detected at the *current* token:
  - For each possible insertion, deletion, or substitution of a token at any position of the queue, the Burke-Fisher error repairer makes that change to within (a copy of) the queue, then attempts to reparse from the old stack.
  - Generally, if three or four tokens past the *current* token can be parsed, this is considered a completely successful repair.

Error Recovery: Global Error Repair

- The advantage of Burke-Fisher Error Repair
  - The grammar is not modified at all (no *error* productions). Only the parsing engine, which interprets the parsing tables, is modified.

Semantic Actions

- Shift and reduce actions are tried repeatly and discarded.
- What if the programmer-specified semantic actions associated with reduce actions have side effects?
  - e.g., nest + 1 and nest -1
- Solution: does not execute any of the semantic actions as reductions are performed on the current stack, but waits until the same reductions are performed (permanently) on the old stack.

Error Recovery: Global Error Repair

Semantic values for insertions

- when tokens such as numbers or identifiers must be inserted, where can their semantic values come from?
- ML-Yacc parser generator has a %value directive

```
%value ID ("bogus")
%value INT (1)
%value STRING ("")
```

Error Recovery: Global Error Error

Programmer-specified substitutions

- Sometimes a particular single-token insertion or substitution is very commonly required and should be tried first. (Provide hints for parser)
- ML-Yacc parser generator has a %change directive

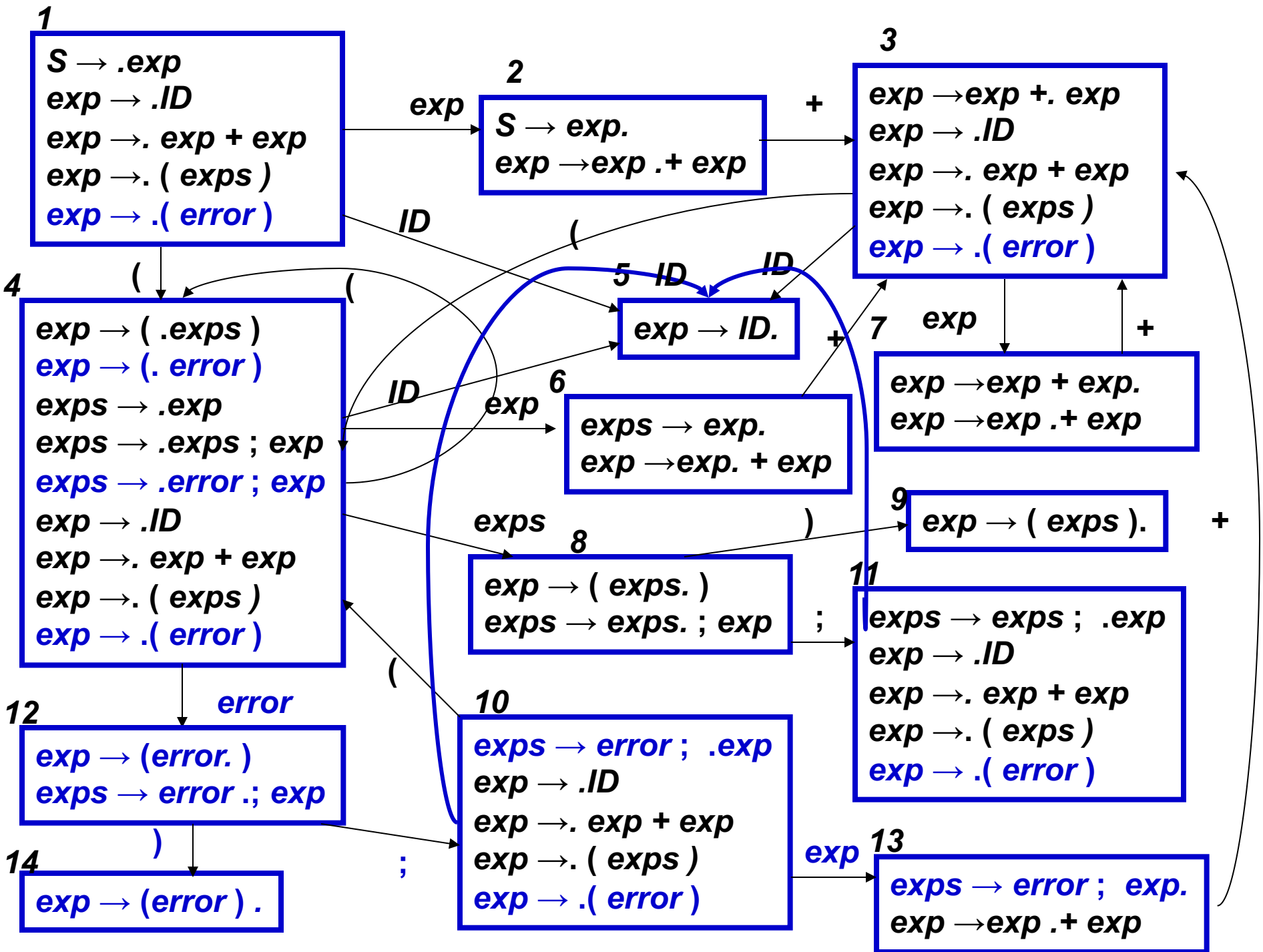
```
%change EQ -> ASSIGN
      | ASSIGN -> EQ
      | SEMICOLON ELSE -> ELSE
      | -> IN INT END
```

- IN INT END -> a scope closer
  - let …. in …. end: the nesting structure in Tiger

S → exp
exp → ID
exp → exp + exp
exp → (exps)
exps → exp
exps → exps ; exp
exp → (error)
exps → error ; exp

input tokens: (a++;b)$

Stack	Input	action
1	(a++;b)$	s4
14, (	a++;b)$	s5
145, (a	++;b)$	r
146, (e	++;b)$	s3
1463, (e+	+;b)$	error
14, (	+;b)$	s12
1412, (error	;b)$	s10
141210, (error;	b)$	s5
1412105, (error; b	)$	r
14121013, (error; e	)$	r
148, (exps	)$	s9
1489, (exps)	$	r
12,e	$	r


1
S → .exp
exp → .ID
exp →. exp + exp
exp →. (exps)
exp → .(error)
exp
2
S → exp.
exp →exp .+ exp
+
3
exp →exp +. exp
exp → .ID
exp →. exp + exp
exp →. (exps)
exp → .(error)
ID
(
4
exp → (.exps)
exp → (. error)
exps → .exp
exps → .exps ; exp
exps → .error ; exp
exp → .ID
exp →. exp + exp
exp →. (exps)
exp → .(error)
5
exp → ID.
ID
ID
6
exps → exp.
exp →exp. + exp
exp
7
exp →exp + exp.
exp →exp .+ exp
exp
+
+
9
exp → (exps).
)
+
exps
8
exp → (exps.)
exps → exps. ; exp
;
11
exps → exps ; .exp
exp → .ID
exp →. exp + exp
exp →. (exps)
exp → .(error)
(
10
exps → error ; .exp
exp → .ID
exp →. exp + exp
exp →. (exps)
exp → .(error)
error
12
exp → (error.)
exps → error .; exp
)
;
14
exp → (error) .
exp
13
exps → error ; exp.
exp →exp .+ exp