

# Compilerconstructie

najaar 2014

<http://www.liacs.nl/home/rvvliet/coco/>

**Rudy van Vliet**

kamer 124 Snellius, tel. 071-527 5777  
rvvliet(at)liacs(dot)nl

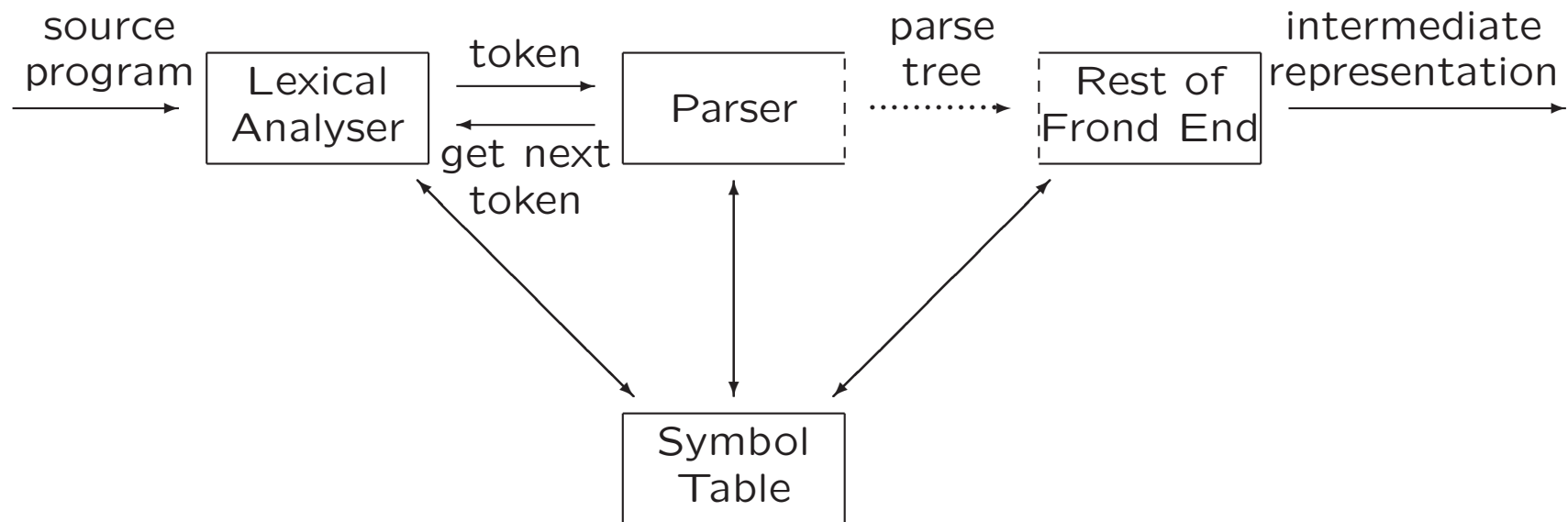
college 4, dinsdag 23 september 2014

+ werkcollege

Syntax Analysis (2)

## 4.1.1 The Role of the Parser

(from college 3)



- Obtain string of tokens
- Verify that string can be generated by the grammar
- Report and recover from syntax errors

# Parsing

(from college 3)

Finding parse tree for given string

- Universal (any CFG)
  - Cocke-Younger-Kasami
  - Earley
- Top-down (CFG with restrictions)
  - Predictive parsing
  - LL (Left-to-right, Leftmost derivation) methods
  - LL(1): LL parser, needs only one token to look ahead
- Bottom-up (CFG with restrictions)

Last week: top-down parsing

Today: bottom-up parsing

## 4.5 Bottom-Up Parsing

LR methods

Left-to-right scanning of input, Rightmost derivation (in reverse)

- Shift-reduce parsing
- Reduce string  $w$  to start symbol
- – Simple LR = SLR(1) = SLR
- Canonical LR = canonical LR(1) = LR
- Look-ahead LR = LALR

## Bottom-Up Parsing (Example)

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \mathbf{id}$$

Construct parse tree for **id \* id** bottom-up. . .

# Bottom-Up Parsing (Example)

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

Reducing a sentence

$$\begin{array}{r} \mathbf{id} * \mathbf{id} \\ \hline F * \mathbf{id} \\ \hline T * \mathbf{id} \\ \hline T * F \\ \hline T \\ \hline E \end{array}$$

Bottom-up parsing corresponds to rightmost derivation

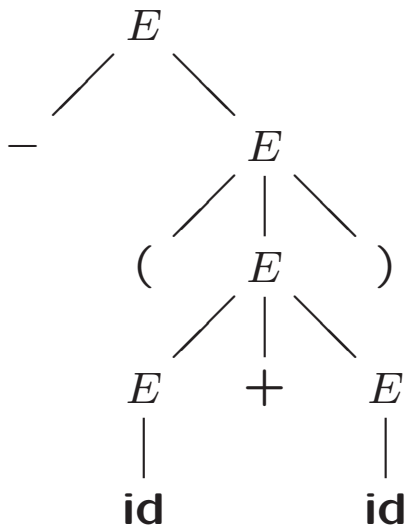
$$\begin{array}{r} E \xRightarrow{rm} T \\ \xRightarrow{rm} T * F \\ \xRightarrow{rm} T * \mathbf{id} \\ \xRightarrow{rm} F * \mathbf{id} \\ \xRightarrow{rm} \mathbf{id} * \mathbf{id} \end{array}$$

## 4.2.4 Parse Trees and Derivations

(from college 3)

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

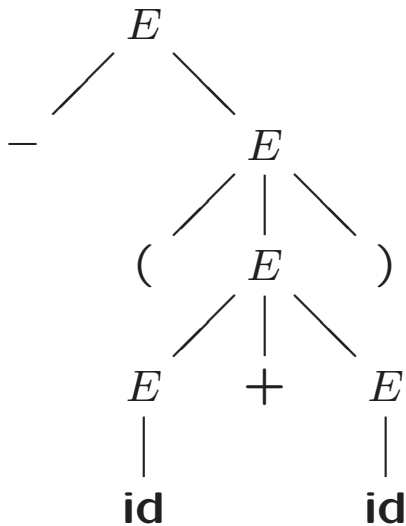
$$E \xRightarrow{lm} -E \xRightarrow{lm} -(E) \xRightarrow{lm} -(E + E) \xRightarrow{lm} -(\mathbf{id} + E) \xRightarrow{lm} -(\mathbf{id} + \mathbf{id})$$



Many-to-one relationship between derivations and parse trees...

# Parse Trees and Derivations

$$E \xRightarrow{lm} -E \xRightarrow{lm} -(E) \xRightarrow{lm} -(E + E) \xRightarrow{lm} -(\mathbf{id} + E) \xRightarrow{lm} -(\mathbf{id} + \mathbf{id})$$



- |                      |   |                                 |
|----------------------|---|---------------------------------|
| Leftmost derivation  | ≈ | WLR construction tree           |
|                      | ≈ | top-down parsing                |
| Rightmost derivation | ≈ | WRL construction tree           |
| Bottom-up parsing    | ≈ | LRW construction tree           |
|                      | ≈ | rightmost derivation in reverse |



## 4.5.2 Handle Pruning

**Handle:** substring that matches body of production, whose reduction represents one step along reverse of rightmost derivation

$$\begin{aligned}
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow (E) \mid \mathbf{id}
 \end{aligned}$$

Reducing a sentence

$$\begin{array}{l}
 \mathbf{id} * \mathbf{id} \\
 \underline{F} * \mathbf{id} \\
 T * \underline{\mathbf{id}} \\
 \underline{T * F} \\
 \underline{T} \\
 E
 \end{array}$$

Bottom-up parsing corresponds to rightmost derivation

$$\begin{array}{l}
 E \xRightarrow{rm} T \\
 \xRightarrow{rm} T * F \\
 \xRightarrow{rm} T * \mathbf{id} \\
 \xRightarrow{rm} F * \mathbf{id} \\
 \xRightarrow{rm} \mathbf{id} * \mathbf{id}
 \end{array}$$

Handles / not a handle...

# Handle Pruning

- Formally, if  $S \xRightarrow{*}_{rm} \alpha Aw \xRightarrow{rm} \alpha\beta w$ , then  $A \rightarrow \beta$  is handle of  $\alpha\beta w$
- Handle pruning to obtain rightmost derivation in reverse
  - $w$  is string of terminals
  - $S = \gamma_0 \xRightarrow{rm} \gamma_1 \xRightarrow{rm} \dots \xRightarrow{rm} \gamma_{n-1} \xRightarrow{rm} \gamma_n = w$
  - Locate handle  $\beta_n$  in  $\gamma_n$  and replace  $\beta_n$  ( $A \rightarrow \beta_n$ ) to obtain right-sentential form  $\gamma_{n-1}$
  - Repeat until we produce right-sentential form consisting of only  $S$
- Problems
  - How to locate substring to be reduced?
  - How to determine what production to choose?

## 4.5.3 Shift-Reduce Parsing

Cf. bottom-up PDA from FI2

Use stack to hold symbols corresponding to part of input already read

- Initially,

Stack	Input
\$	$w$ \$

- Repeat
  - Shift zero or more input symbols onto stack
  - Reduce a detected handle **on top of stack**

until error or

Stack	Input
$\$S$	\$

# Shift-Reduce Parsing

Cf. bottom-up PDA from FI2

Use stack to hold symbols corresponding to part of input already read

Possible actions shift-reduce parser:

- **Shift**      shift next symbol onto stack
- **Reduce**    replace handle on top of stack by nonterminal
- **Accept**    announce successful completion of parsing
- **Error**      detect syntax error and call error recovery routine

# Shift-Reduce Parsing (Example)

	Stack	Input	Action
$E \rightarrow E + T \mid T$	\$	<b>id<sub>1</sub></b> * <b>id<sub>2</sub></b> \$	shift
$T \rightarrow T * F \mid F$	\$ <b>id<sub>1</sub></b>	* <b>id<sub>2</sub></b> \$	reduce by $F \rightarrow \mathbf{id}$
$F \rightarrow (E) \mid \mathbf{id}$	\$ <b>F</b>	* <b>id<sub>2</sub></b> \$	reduce by $T \rightarrow F$
	\$ <b>T</b>	* <b>id<sub>2</sub></b> \$	shift
	\$ <b>T*</b>	<b>id<sub>2</sub></b> \$	shift
	\$ <b>T * id<sub>2</sub></b>	\$	reduce by $F \rightarrow \mathbf{id}$
	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
	\$ <b>E</b>	\$	accept

Problems remain

- How to determine when to reduce
- How to determine what production to choose?

# Shift-Reduce Parsing (Example)

	Stack	Input	Action
$E \rightarrow E + T \mid T$	\$	<b>id<sub>1</sub></b> * <b>id<sub>2</sub></b> \$	shift
$T \rightarrow T * F \mid F$	\$ <b>id<sub>1</sub></b>	* <b>id<sub>2</sub></b> \$	reduce by $F \rightarrow \mathbf{id}$
$F \rightarrow (E) \mid \mathbf{id}$	\$ <b>F</b>	* <b>id<sub>2</sub></b> \$	reduce by $T \rightarrow F$
	\$ <b>T</b>	* <b>id<sub>2</sub></b> \$	shift
	\$ <b>T*</b>	<b>id<sub>2</sub></b> \$	shift
	\$ <b>T * id<sub>2</sub></b>	\$	reduce by $F \rightarrow \mathbf{id}$
	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
	\$ <b>E</b>	\$	accept

Problems remain

- How to determine when to reduce
- How to determine what production to choose?

## 4.5.4 Conflicts During Shift-Reduce Parsing

Sometimes stack contents and next input symbol are not sufficient to determine shift / (which) reduce

- Shift/reduce conflicts and reduce/reduce conflicts
- Caused by
  - Ambiguity of grammar
  - Limitation of the LR parsing method used (even when grammar is unambiguous)

# Shift/Reduce Conflict (Example)

“Dangling-else” -grammar

$stmt \rightarrow$  **if** *expr* **then** *stmt*  
          | **if** *expr* **then** *stmt* **else** *stmt*  
          | **other**

Stack	Input	Action
\$ ...	...\$	...
\$ ... <b>if</b> <i>expr</i> <b>then</b> <b>if</b> <i>expr</i> <b>then</b> <i>stmt</i>	<b>else</b> ... \$	shift or reduce?

Resolve in favour of shift,  
so **else** matches closest unmatched **then**



# Reduce/Reduce Conflict (Example)

$stmt \rightarrow id (parameter\_list) \mid expr := expr$   
 $parameter\_list \rightarrow parameter\_list, parameter \mid parameter$   
 $parameter \rightarrow id$   
 $expr \rightarrow id (expr\_list) \mid id$   
 $expr\_list \rightarrow expr\_list, expr \mid expr$

Statement beginning with  $p(i, j)$  would appear as token stream **id (id, id )**

Stack	Input	Action
\$ ...	... \$	...
\$ ... <b>id (id</b>	<b>, id )</b> ... \$	reduce by $parameter \rightarrow id$ or by $expr \rightarrow id$ ?

# Reduce/Reduce Conflict (Example)

Possible solution

$$\begin{aligned}
 stmt &\rightarrow \mathbf{procid} (parameter\_list) \mid expr := expr \\
 parameter\_list &\rightarrow parameter\_list, parameter \mid parameter \\
 parameter &\rightarrow \mathbf{id} \\
 expr &\rightarrow \mathbf{id} (expr\_list) \mid \mathbf{id} \\
 expr\_list &\rightarrow expr\_list, expr \mid expr
 \end{aligned}$$

Requires more sophisticated lexical analyser

Stack	Input	Action
\$ ...	... \$	...
\$ ... <b>procid</b> ( <b>id</b>   , <b>id</b> ) ... \$		reduce by <i>parameter</i> → <b>id</b>

or

Stack	Input	Action
\$ ...	... \$	...
\$ ... <b>id</b> ( <b>id</b>   , <b>id</b> ) ... \$		reduce by <i>expr</i> → <b>id</b>

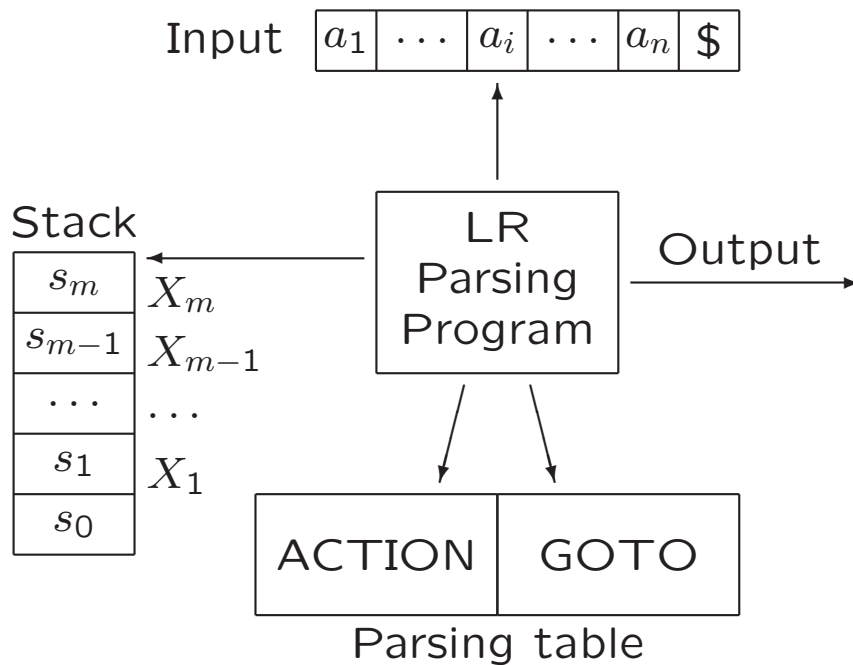
## 4.6 Introduction to LR Parsing

- Bottom-up parsing for large class of CFGs
- LR( $k$ )
  - Left-to-right scanning of input
  - Rightmost derivation in reverse
  - $k$  symbols of look-ahead
- Maintains **states** representing **'item sets'**, which are used to construct **parsing table**, which guides **shift/reduce decisions**

## 4.6.1 Why LR Parsers?

- LR parser pros:
  - Covers all programming language constructs
  - Most general non-backtracking shift-reduce parsing
  - Allows efficient implementation
  - Detects syntactic errors as soon as possible (in left-to-right scanning)
  - Can parse more grammars than  $LL(k)$  parsers
- LR parser con: too much work to be constructed by hand, but: LR parser generators available

## 4.6.2 Items and the LR(0) Automaton



# Simple LR Parsing

States are sets of LR(0) items

Production  $A \rightarrow XYZ$  yields four items:

$$A \rightarrow \cdot XYZ$$

$$A \rightarrow X \cdot YZ$$

$$A \rightarrow XY \cdot Z$$

$$A \rightarrow XYZ \cdot$$

Item indicates how much of production we have seen in input

LR(0) items are combined in sets

Canonical LR(0) collection is specific collection of item sets

These item sets are the states in LR(0) automaton,  
a DFA that is used for making parsing decisions

# Closure of Item Sets

- – Consider  $A \rightarrow \alpha \cdot B \beta$ 
  - We expect to see substring derivable from  $B\beta$ , with prefix derivable from  $B$ , by applying  $B$ -production
  - Hence, add  $B \rightarrow \cdot \gamma$  for all  $B \rightarrow \gamma$
- Let  $I$  be item set
  1. Add every item in  $I$  to  $\text{CLOSURE}(I)$
  2. Repeat
    - If  $A \rightarrow \alpha \cdot B \beta$  is in  $\text{CLOSURE}(I)$  and  $B \rightarrow \gamma$  is production, then add  $B \rightarrow \cdot \gamma$  to  $\text{CLOSURE}(I)$until no more new items are added

# Closure of Item Sets (Example)

Augmented grammar

$$\begin{aligned}E' &\rightarrow E \\E &\rightarrow E + T \mid T \\T &\rightarrow T * F \mid F \\F &\rightarrow (E) \mid \mathbf{id}\end{aligned}$$

If  $I = \{[E' \rightarrow \cdot E]\}$ , then  $\text{CLOSURE}(I) = \dots$



# Closure of Item Sets (Example)

Augmented grammar

$$\begin{aligned}E' &\rightarrow E \\E &\rightarrow E + T \mid T \\T &\rightarrow T * F \mid F \\F &\rightarrow (E) \mid \mathbf{id}\end{aligned}$$

If  $I = \{[E' \rightarrow \cdot E]\}$ , then  $\text{CLOSURE}(I) = I_0$ :

$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot \mathbf{id}$

# Function GOTO

- Let  $I$  be set of items, and  $X$  be grammar symbol
- $\text{GOTO}(I, X)$ : items you can get by moving  $\cdot$  over  $X$  in items from  $I$  (and then taking closure)

Example:

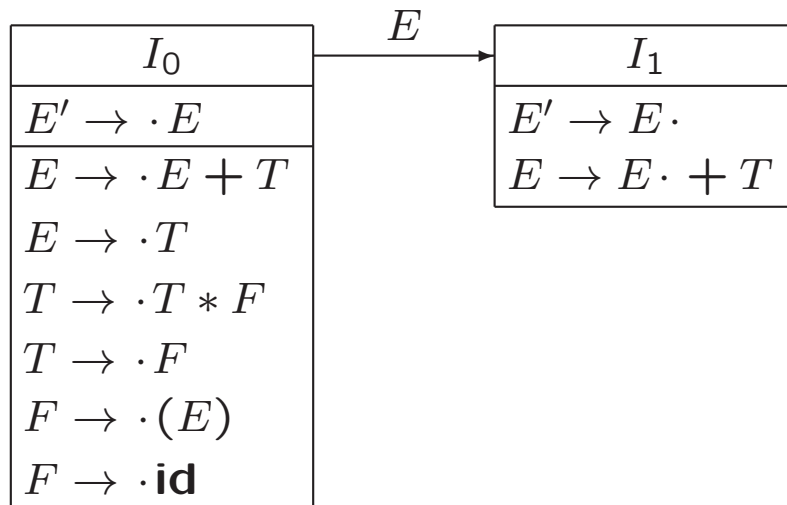
$I_0$
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot \mathbf{id}$

$\text{GOTO}(I_0, E) = \dots$

# Function GOTO

- Let  $I$  be set of items, and  $X$  be grammar symbol
- $\text{GOTO}(I, X)$ : items you can get by moving  $\cdot$  over  $X$  in items from  $I$  (and then taking closure)

Example:



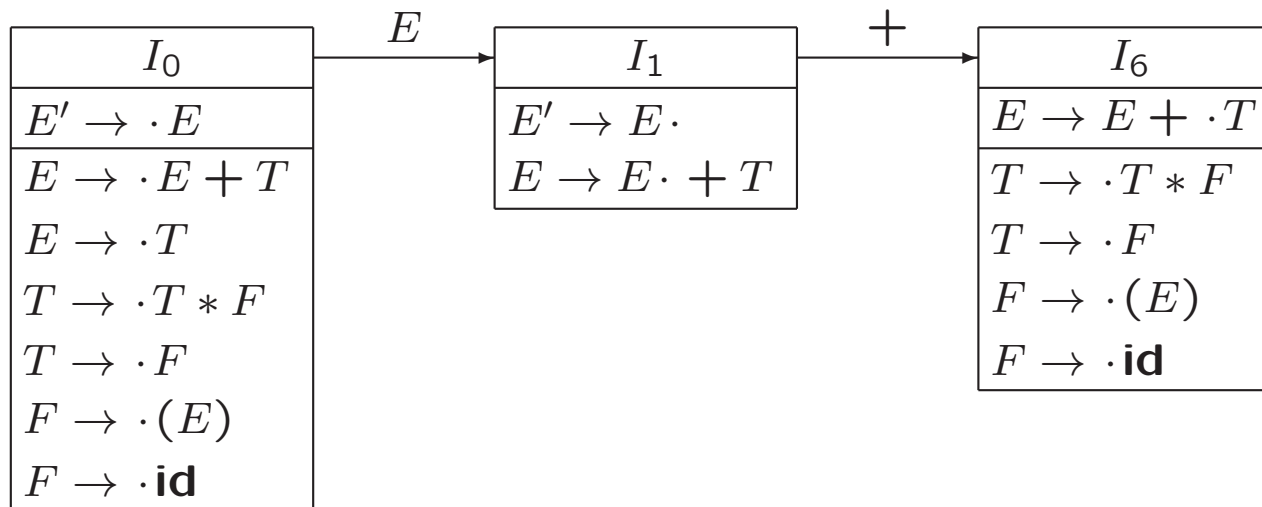
$$\text{GOTO}(I_0, E) = I_1$$

$$\text{GOTO}(I_1, +) = \dots$$

# Function GOTO

- Let  $I$  be set of items, and  $X$  be grammar symbol
- $\text{GOTO}(I, X)$ : items you can get by moving  $\cdot$  over  $X$  in items from  $I$  (and then taking closure)

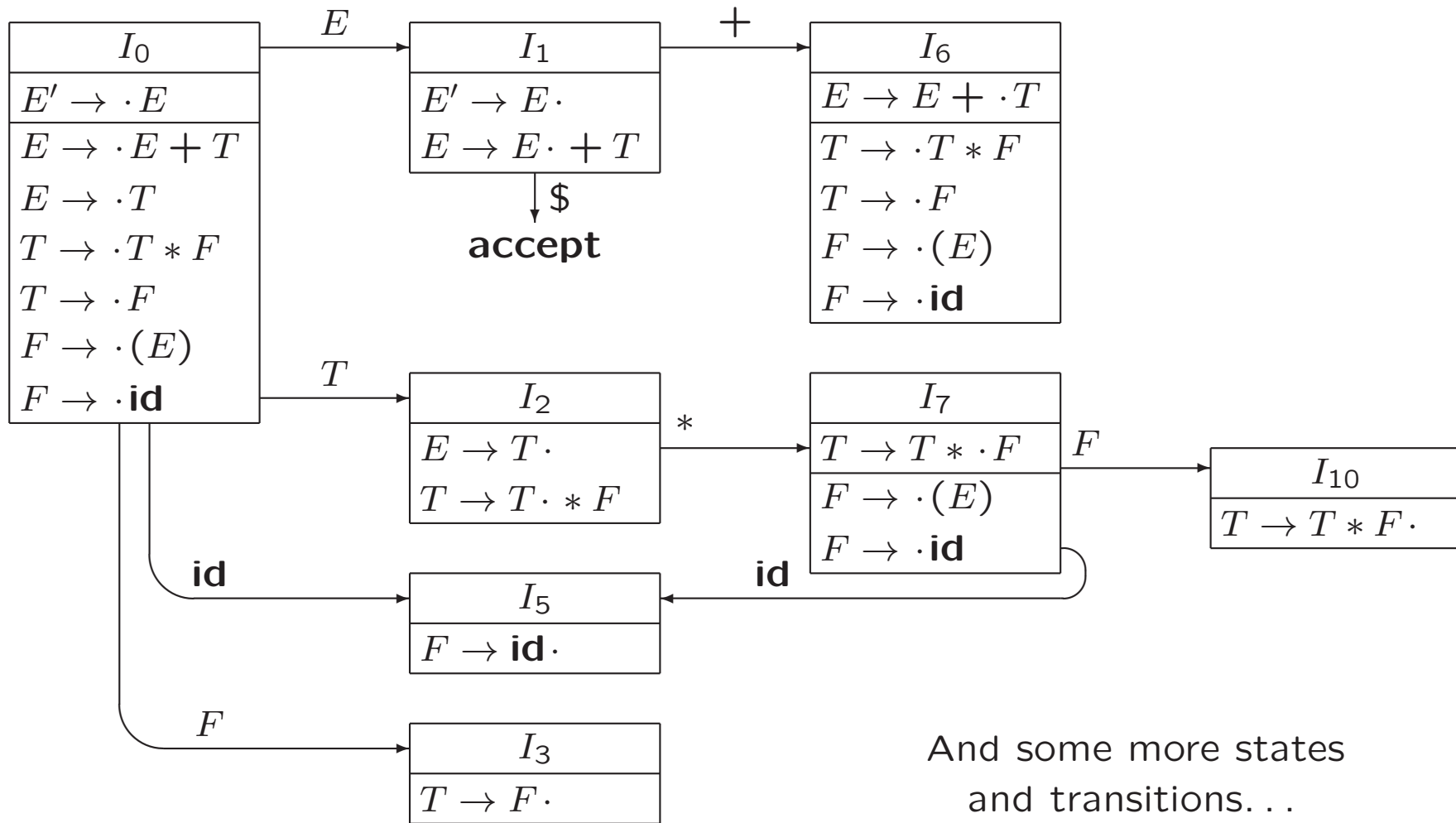
Example:



$$\text{GOTO}(I_0, E) = I_1$$

$$\text{GOTO}(I_1, +) = I_6$$

# LR(0) Automaton (Example)



And some more states  
and transitions...

# Use of LR(0) Automaton

- Repeat
  - If possible, then shift on next input symbol
  - Otherwise, reduce

until error or accept

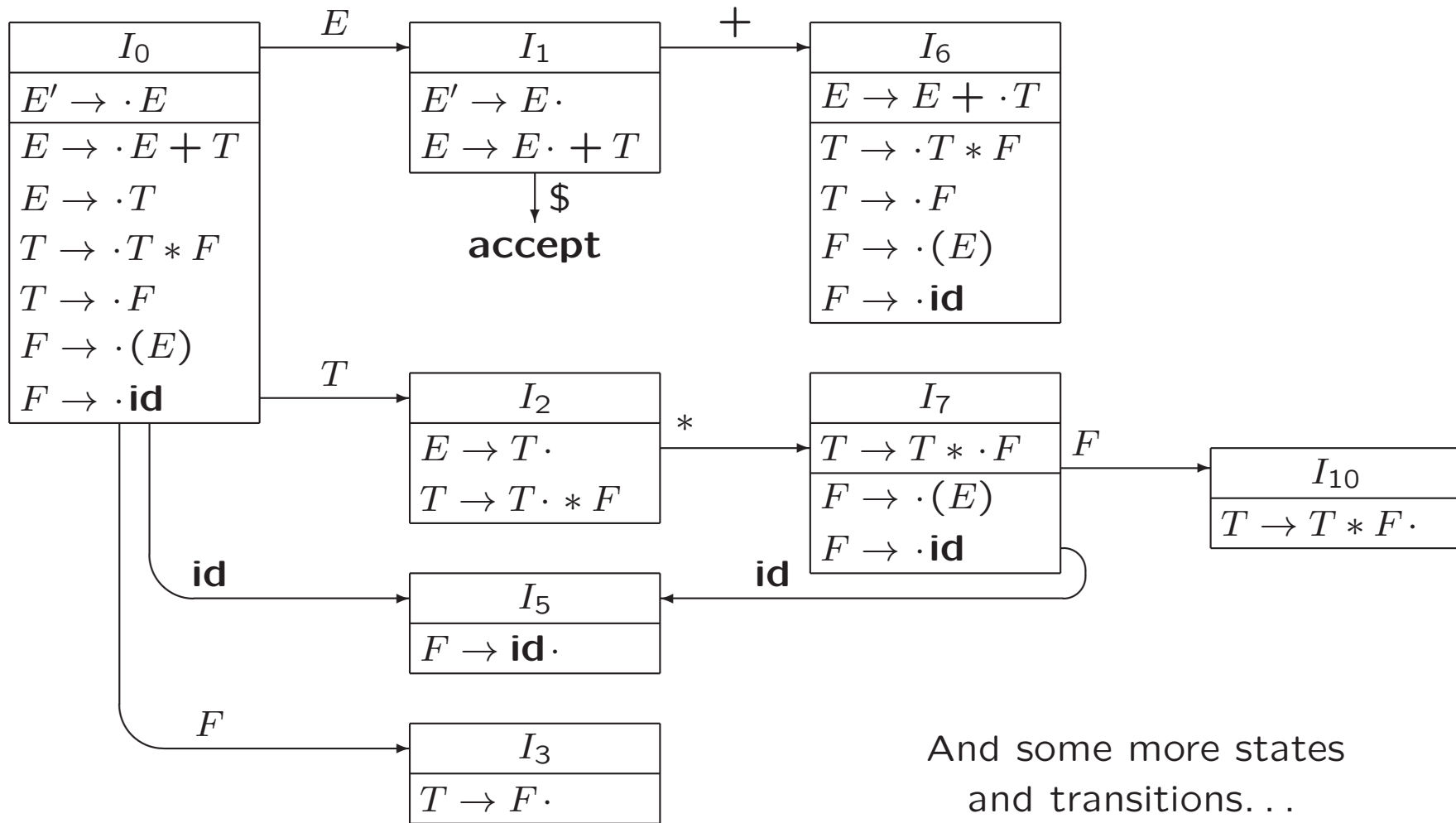
- Example: parsing **id \* id**

Stack	Symbols	Input	Action
0	\$	<b>id * id</b> \$	...

# Use of LR(0) Automaton

- Repeat
  - If possible, then shift on next input symbol
  - Otherwise, reduceuntil error or accept
- It is not as simple as this: there may be
  - shift/reduce conflicts
  - reduce/reduce conflicts

# LR(0) Automaton (Example)



And some more states and transitions...



## 4.6.4 Constructing SLR-Parsing Tables

For state  $i$  and input symbol  $a$ ,

- if  $[A \rightarrow \alpha \cdot a \beta]$  is in  $I_i$  and  $\text{GOTO}(I_i, a) = I_j$   
then shift  $j$  is possible  
( $a$  must be terminal, not \$)
- if  $[A \rightarrow \alpha \cdot]$  is in  $I_i$  and  $a \in \text{FOLLOW}(A)$ ,  
then reduce  $A \rightarrow \alpha$  is possible ( $A$  may not be  $S'$ )
- if  $[S' \rightarrow S \cdot]$  is in  $I_i$  and  $a = \$$ , then accept is possible

If conflicting actions result from this, then grammar is not SLR(1)

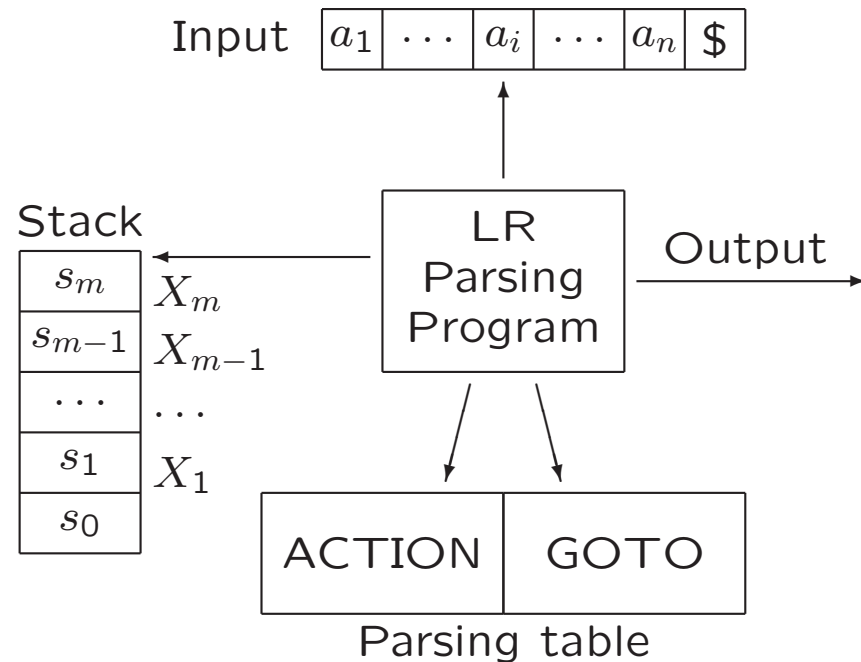
## 4.6.3 The LR-Parsing Algorithm

### SLR, LR, LALR

For state  $i$  and terminal  $a$ ,  $\text{ACTION}[i, a]$ , can have four possible values:

1. shift (state)  $j$
2. reduce  $A \rightarrow \beta$
3. accept
4. error

For state  $i$  and nonterminal  $A$ ,  $\text{GOTO}[i, A]$  is state  $j$



# Behaviour of the LR Parser

LR parser configuration is pair (stack contents, remaining input):

$$(s_0s_1s_2 \dots s_m, a_ia_{i+1} \dots a_n\$)$$

which represents right-sentential form

$$X_1X_2 \dots X_ma_ia_{i+1} \dots a_n$$

1. If  $\text{ACTION}[s_m, a_i] = \text{shift } s$ , then push  $s$  and advance input:

$$(s_0s_1s_2 \dots s_ms, a_{i+1} \dots a_n\$)$$

2. If  $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$ , where  $|\beta| = r$ , then pop  $r$  symbols. If  $\text{GOTO}[s_{m-r}, A] = s$ , then push  $s$ :

$$(s_0s_1s_2 \dots s_{m-r}s, a_ia_{i+1} \dots a_n\$)$$

3. If  $\text{ACTION}[s_m, a_i] = \text{accept}$ , then stop
4. If  $\text{ACTION}[s_m, a_i] = \text{error}$ , then call error recovery routine

# Possible Misconceptions

Reduction by  $A \rightarrow \beta$  is possible

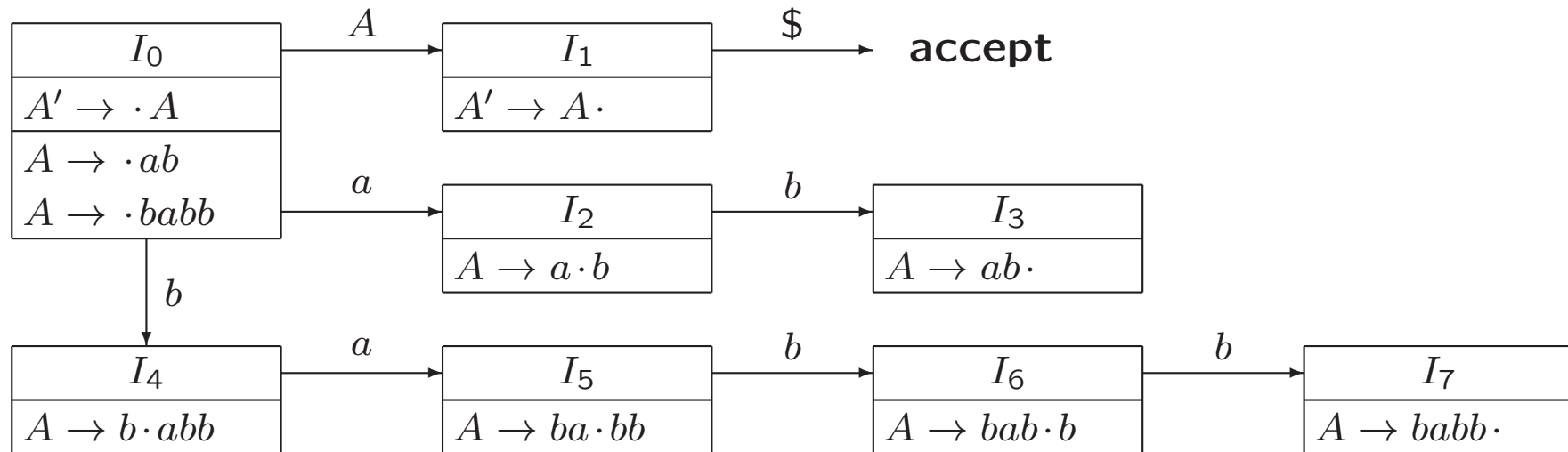
- **not** always when  $\beta$  is on top of stack...

Reduction by  $A \rightarrow \beta$

- does **not** simply mean that we pop symbols from stack until we see state with  $A$ -transition...

# Misconception 1 (Example)

$A \rightarrow ab \mid babb$



Consider input  $babb$

# SLR Parsing Table (Example)

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \mathbf{id}$

State	ACTION					GOTO			
	id	+	*	(	)	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Blank means error

Stack	Symbols	Input	Action
0	\$	<b>id * id\$</b>	shift to 5
05	<b>\$id</b>	<b>*id\$</b>	reduce by $F \rightarrow \mathbf{id}$
03	<b>\$F</b>	<b>*id\$</b>	...

# Different LR Parsing Methods

- Simple LR = SLR
  - Easiest to implement, least powerful
- Canonical LR
  - Augment SLR with lookahead information
  - LR(1) items:  $[A \rightarrow \alpha \cdot \beta, a]$
  - Most expensive to implement, most powerful
- Look-ahead LR = LALR
  - Merge sets of LR(1)-items, so fewer states
  - Often used in practice
- All parsers have same behaviour  
They differ in how parsing table is built

## 4.7.6 Compaction of LR Parsing Tables

- Typical grammar: 100 terminals and productions
  - Several hundreds of states, 20,000 action entries
- Two-dimensional array is not efficient
- Compacting action field of parsing table
  - Many rows are identical, so create pointer for each state into one-dimensional array
  - Make list for actions of each state, consisting of pairs (terminal-symbol, action)



# Compaction of Parsing Table (Example)

State	ACTION					GOTO			
	id	+	*	(	)	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

List for states

0, 4, 6, 7:

Symbol	Action
<b>id</b>	s5
(	s4
<b>any</b>	error

List for state 1:

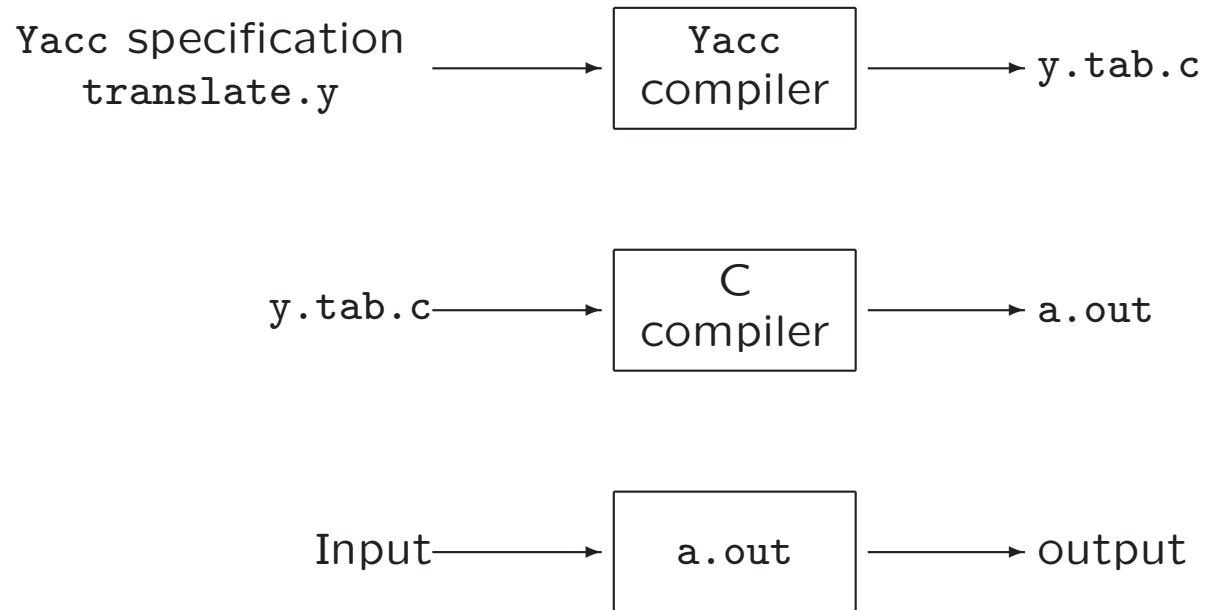
Symbol	Action
+	s6
\$	acc
<b>any</b>	error

## 4.9 Parser Generators

Yacc: Yet Another Compiler Compiler

- Is an LALR(1) parser generator
- Automatically produces parser for CFG
- Deals with ambiguity and difficult-to-parse constructs
  - Reports on conflicts
- Available as command on Unix

## 4.9.1 The Parser Generator Yacc



```
yacc translate.y  
gcc y.tab.c -ly  
./a.out
```

# Yacc Specification

- A Yacc program consists of three parts:

declarations

%%

translation rules

%%

auxiliary functions

- Translation rules are of the form

production { semantic actions }

```
⟨head⟩ : ⟨body⟩1 {⟨semantic action⟩1}  
       | ⟨body⟩2 {⟨semantic action⟩2}  
       ...  
       | ⟨body⟩n {⟨semantic action⟩n}  
       ;
```

# Yacc Specification (Example)

Example: Desktop calculator with following grammar

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{digit} \end{aligned}$$

```
/* declarations section */
%{
#include <ctype.h>
}%

%token DIGIT

%%
/* translation rules section */

line    : expr '\n'          { printf("%d\n", $1); }
        ;
```

# Yacc Specification (Example)

```
expr    : expr '+' term    { $$ = $1 + $3; }
        | term
        ;
term     : term '*' factor  { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'     { $$ = $2; }
        | DIGIT
        ;
```

```
%%
/* auxiliary functions section */
yylex()
{   int c;
    c = getchar();
    if (isdigit(c))
    {   yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

## 4.9.2 Using Yacc with Ambig. Grammars

- Ambiguous grammar for our calculator:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid \mathbf{number}$$

- Allow sequence of expressions and blank lines:

```
lines : lines expr '\n'    { printf("%f\n", $2); }
      | lines '\n'
      | /* empty */
      ;
```

- LALR algorithm will generate parsing action conflicts
  - invoke Yacc with `-v` option

# Yacc Specification (Example)

```
/* declarations section */
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* double type for Yacc stack */
*}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
/* translation rules section */

lines : lines expr '\n'    { printf("%f\n", $2); }
      | lines '\n'
      | /* empty */
      ;
```



# Yacc Specification (Example)

```
expr  : expr '+' expr      { $$ = $1 + $3; }
      | expr '-' expr      { $$ = $1 - $3; }
      | expr '*' expr      { $$ = $1 * $3; }
      | expr '/' expr      { $$ = $1 / $3; }
      | '(' expr ')'       { $$ = $2; }
      | '-' expr %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
```

```
%%
/* auxiliary functions section */
yylex()
{   int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c== '.') || (isdigit(c)) )
    { ungetc(c, stdin);
      scanf("%lf", &yylval);
      return NUMBER;
    }
    return c;
}
```

# Precedence and Associativity

- Same precedence and left associative:

```
%left '+' '-'
```

- Right associative:

```
%right '^'
```

- Increasing precedence:

```
%left '+' '-'
```

```
%left '*' '/'
```

```
%right UMINUS
```

- Non-associative binary operator:

```
%nonassoc '<'
```

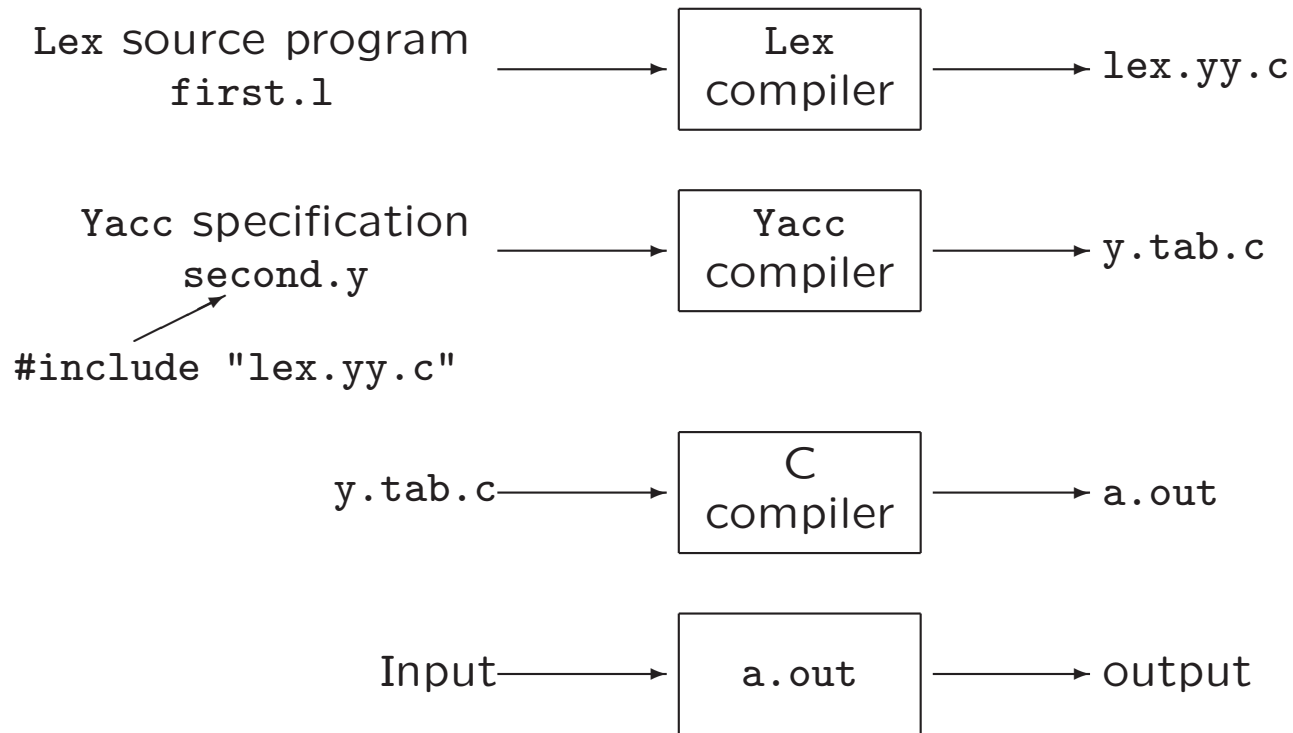
- Precedence and associativity to each production

- Default: rightmost operator

- Otherwise: `%prec <terminal>`

```
expr : '-' expr %prec UMINUS { $$ = - $2; }
```

## 4.9.3 Creating Yacc Lexers with Lex



```
lex first.l
yacc second.y
gcc y.tab.c -ly -ll
./a.out
```

# Volgende week

- Practicum over opdracht 1
- Eerst naar 402, daarna naar 302/304
- Komt wellicht al eerder online
- Inleveren 6 oktober

# Compilerconstructie

college 4

Syntax Analysis (2)

Chapters for reading: 4.5, 4.6, 4.7.6, 4.9