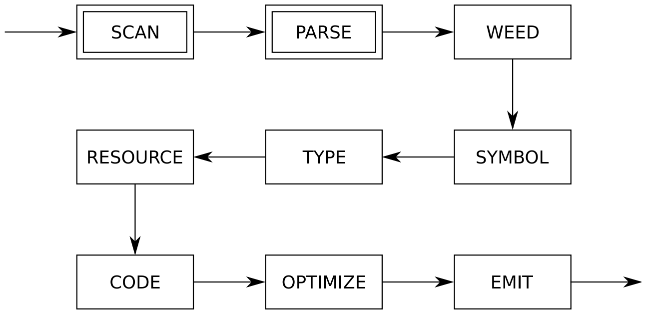
Parsing - Part 2

COMP 520: Compiler Design (4 credits)

Professor Laurie Hendren

`hendren@cs.mcgill.ca`

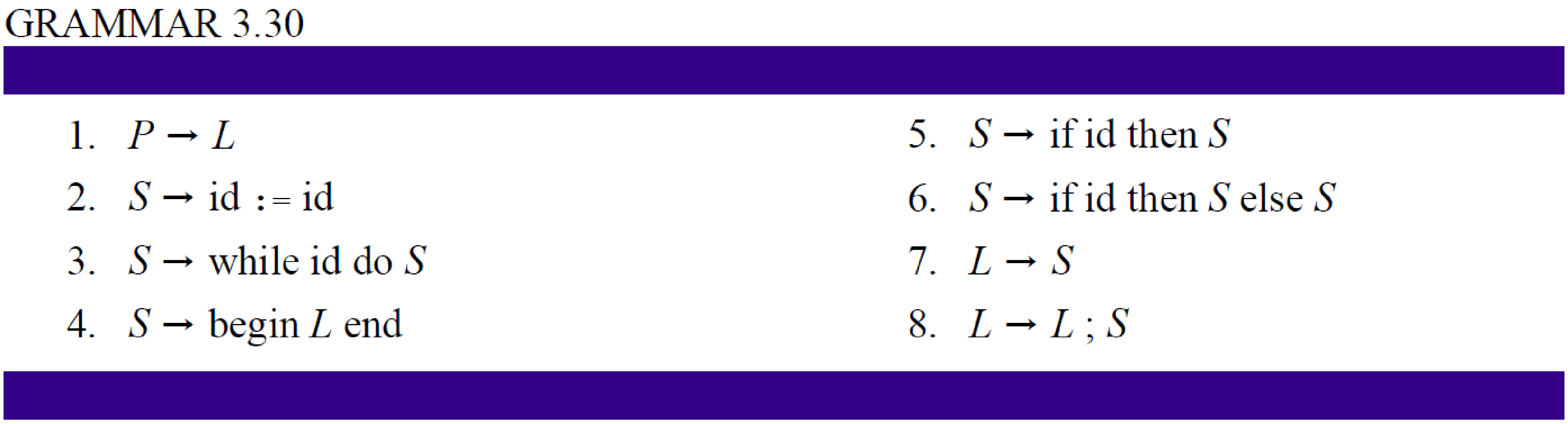

SCAN
PARSE
WEED
SYMBOL
TYPE
RESOURCE
CODE
OPTIMIZE
EMIT

READING - very important for this phase

- Crafting a Compiler:
  - Chapter 4.1 to 4.4 recommended
  - Chapter 4.5 optional
  - Chapter 5.1 to 5.2 recommended
  - Chapter 5.3 to 5.9 optional
  - Chapter 6.1, 6.2 and 6.4 recommended
  - Chapter 6.3 and 6.5 optional
- Modern Compiler Implementation in Java:
  - Chapter 3 (will help explain the slides)
- Tool Documentation: (links on `http://www.cs.mcgill.ca/~cs520/2015`
  - flex
  - bison
  - SableCC

A bit more on SableCC and ambiguities

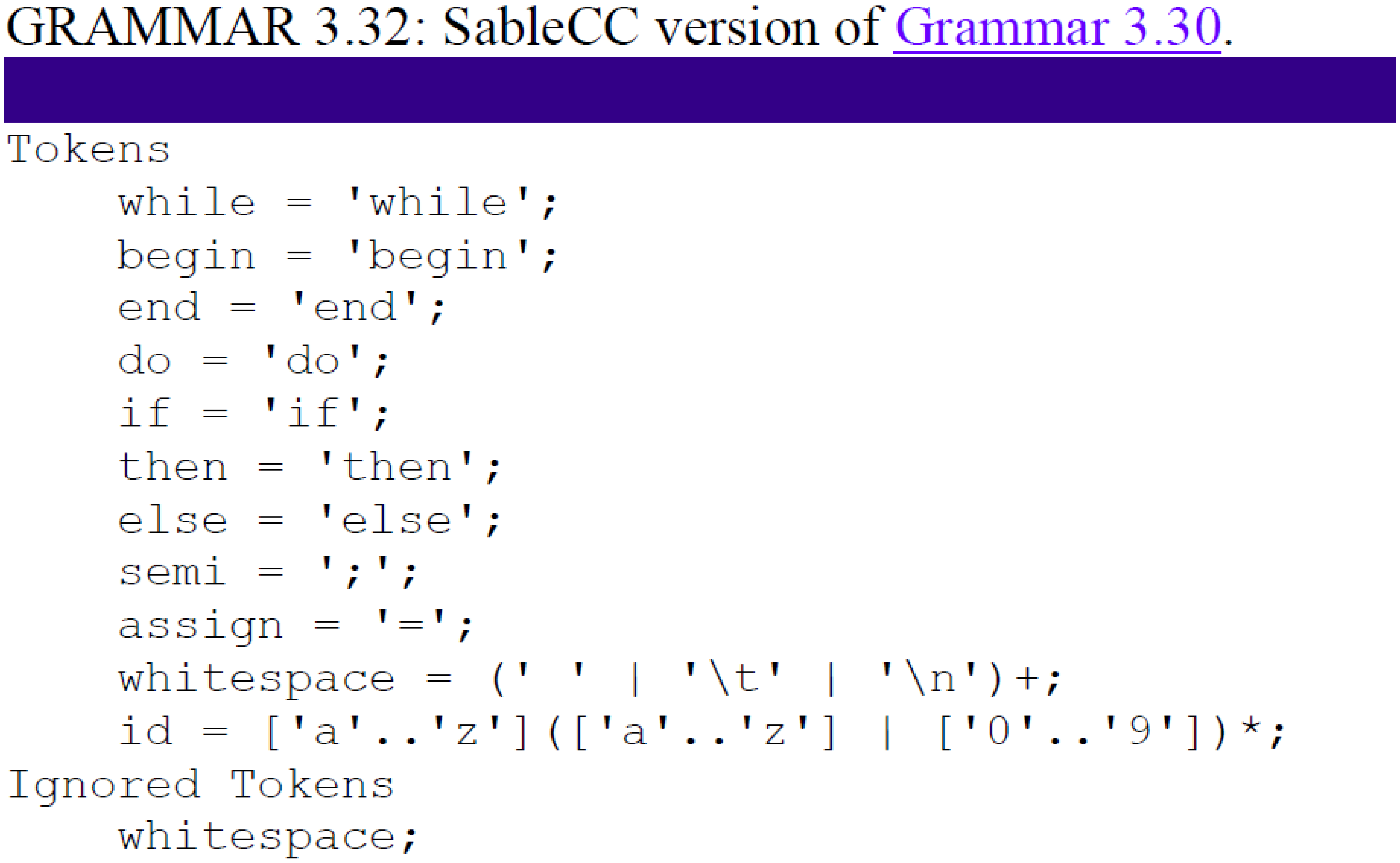
The next slides are from ”Modern Compiler Implementation in Java”, by Appel and Palsberg.



GRAMMAR 3.30

1. $P \rightarrow L$
2. $S \rightarrow$ id := id
3. $S \rightarrow$ while id do S
4. $S \rightarrow$ begin L end
5. $S \rightarrow$ if id then S
6. $S \rightarrow$ if id then S else S
7. $L \rightarrow S$
8. $L \rightarrow L$; S

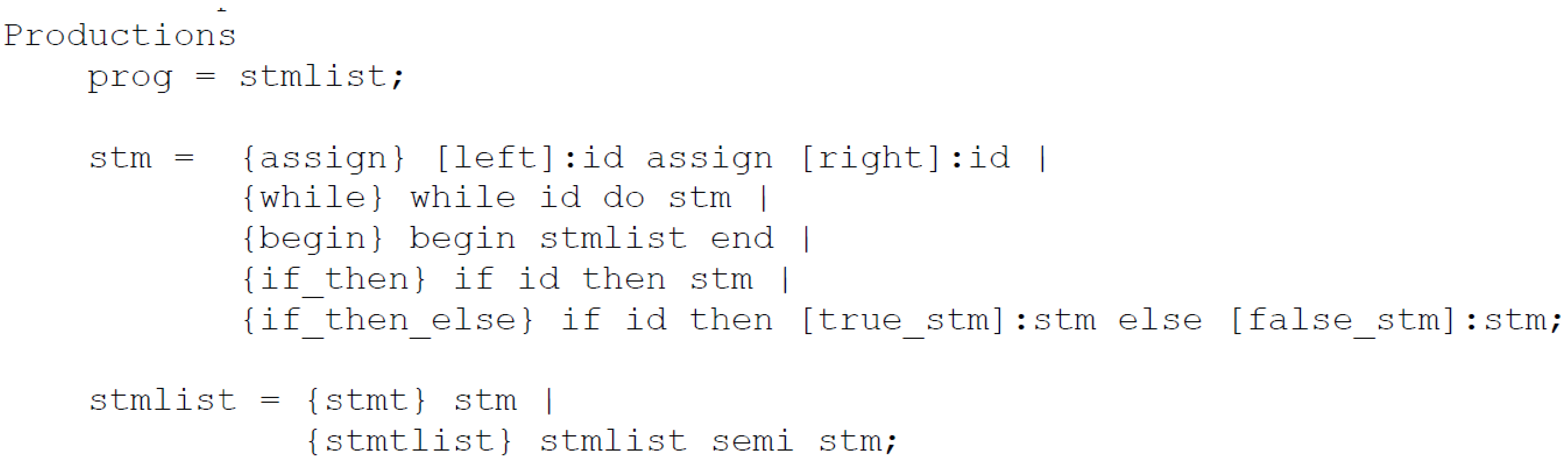
First part of SableCC specfication (scanner)



GRAMMAR 3.32: SableCC version of Grammar 3.30.

```
Tokens
    while = 'while';
    begin = 'begin';
    end = 'end';
    do = 'do';
    if = 'if';
    then = 'then';
    else = 'else';
    semi = ';';
    assign = '=';
    whitespace = (' ' | '\t' | '\n')+;
    id = ['a'..'z'](['a'..'z'] | ['0'..'9'])*;
Ignored Tokens
    whitespace;
```

Second part of SableCC specfication (parser)

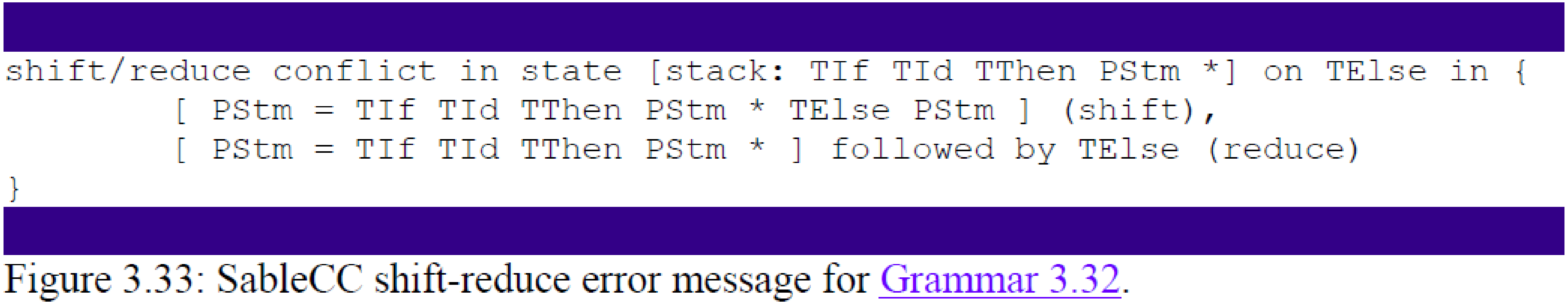


```
Productions
    prog = stmlist;

    stm =  {assign} [left]:id assign [right]:id |
           {while} while id do stm |
           {begin} begin stmlist end |
           {if_then} if id then stm |
           {if_then_else} if id then [true_stm]:stm else [false_stm]:stm;

    stmlist = {stmt} stm |
              {stmtlist} stmlist semi stm;
```

Shift reduce confict because of ”dangling else problem”



```
shift/reduce conflict in state [stack: TIf TId TThen PStm *] on TElse in {
        [ PStm = TIf TId TThen PStm * TElse PStm ] (shift),
        [ PStm = TIf TId TThen PStm * ] followed by TElse (reduce)
}
```

Figure 3.33: SableCC shift-reduce error message for Grammar 3.32.

GRAMMAR 3.34: SableCC productions of Grammar 3.32 with conflicts resolved.

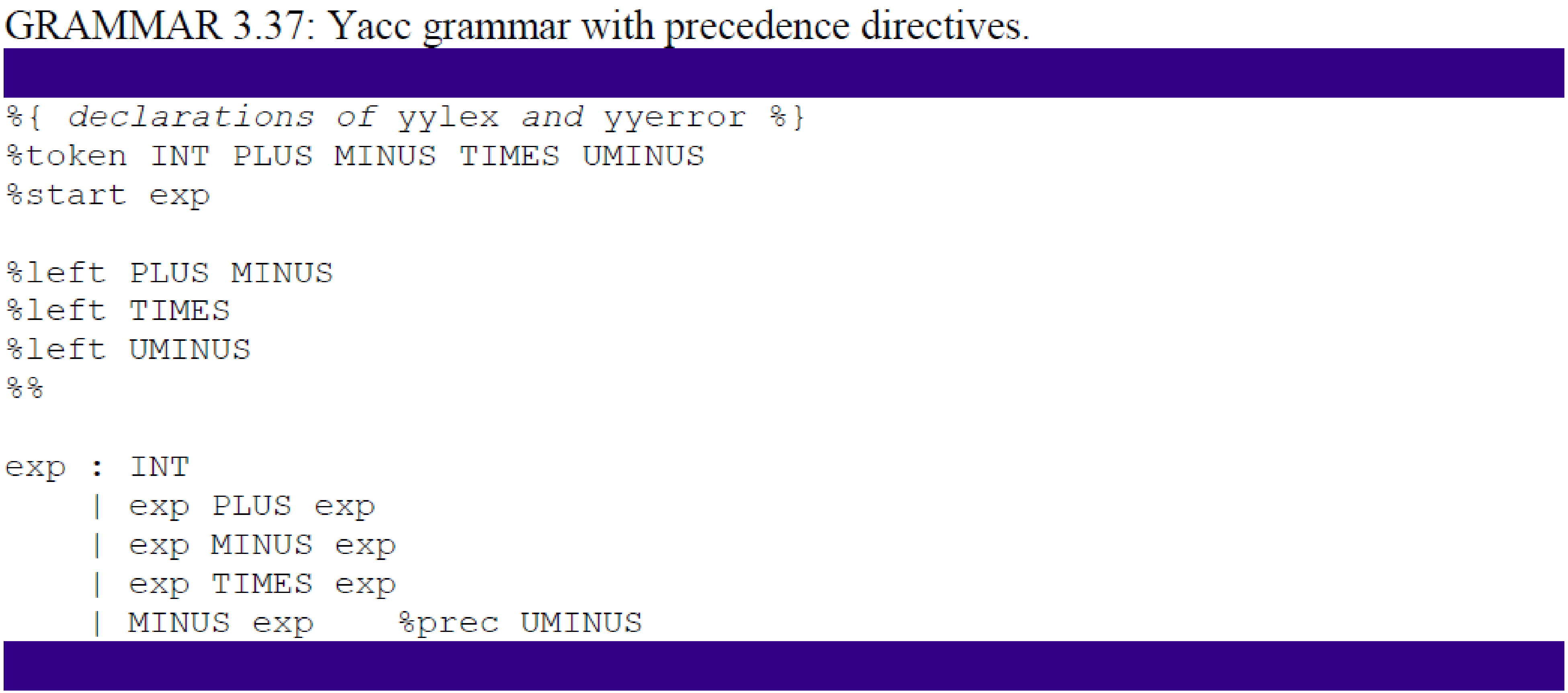
```
Productions
    prog = stmlist;

    stm = {stm_without_trailing_substm}
               stm_without_trailing_substm |
           {while} while id do stm |
           {if_then} if id then stm |
           {if_then_else} if id then stm_no_short_if
                          else [false_stm]:stm;

    stm_no_short_if = {stm_without_trailing_substm}
                           stm_without_trailing_substm |
                          {while_no_short_if}
                           while id do stm_no_short_if |
                        {if_then_else_no_short_if}
                          if id then [true_stm]:stm_no_short_if
                                else [fals_stm]:stm_no_short_if;

    stm_without_trailing_substm = {assign} [left]:id assign [right]:id |
                                      {begin} begin stmlist end ;
    stmlist = {stmt} stm | {stmtlist} stmlist semi stm;
```

Shortcut for giving precedence to unary minus in bison/yacc



GRAMMAR 3.37: Yacc grammar with precedence directives.

```
%{ declarations of yylex and yyerror %}
%token INT PLUS MINUS TIMES UMINUS
%start exp

%left PLUS MINUS
%left TIMES
%left UMINUS
%%

exp : INT
    | exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | MINUS exp    %prec UMINUS
```

Back to Foundations:

Reminder, a *parser* transforms a string of tokens into a parse tree, according to some grammar:

- it corresponds to a *deterministic push-down automaton*;
- plus some glue code to make it work;
- can be generated by `bison` (or `yacc`), CUP, ANTLR, SableCC, Beaver, JavaCC, ...

The *shift-reduce* bottom-up parsing technique.

1) Extend the grammar with an end-of-file \$, introduce fresh start symbol S':

$S' \rightarrow S\$$

$S \rightarrow S$; S $\quad$ $E \rightarrow$ id $\quad$ $L \rightarrow E$

$S \rightarrow$ id := E $\quad$ $E \rightarrow$ num $\quad$ $L \rightarrow L$, E

$S \rightarrow$ print (L) $\quad$ $E \rightarrow E$ + E

$E \rightarrow$ (S , E)

2) Choose between the following actions:

- shift:

  move first input token to top of stack

- reduce:

  replace α on top of stack by X

  for some rule $X \rightarrow \alpha$

- accept:

  when S' is on the stack

	`a:=7; b:=c+(d:=5+6,d)$`	shift
id	`:=7; b:=c+(d:=5+6,d)$`	shift
id :=	`7; b:=c+(d:=5+6,d)$`	shift
id := num	`; b:=c+(d:=5+6,d)$`	$E \longrightarrow$ num
id := E	`; b:=c+(d:=5+6,d)$`	$S \longrightarrow$ id:=E
S	`; b:=c+(d:=5+6,d)$`	shift
S;	`b:=c+(d:=5+6,d)$`	shift
S; id	`:=c+(d:=5+6,d)$`	shift
S; id :=	`c+(d:=5+6,d)$`	shift
S; id := id	`+(d:=5+6,d)$`	$E \longrightarrow$ id
S; id := E	`+(d:=5+6,d)$`	shift
S; id := E +	`(d:=5+6,d)$`	shift
S; id := E + (	`d:=5+6,d)$`	shift
S; id := E + (id	`:=5+6,d)$`	shift
S; id := E + (id :=	`5+6,d)$`	shift
S; id := E + (id := num	`+6,d)$`	$E \longrightarrow$ num
S; id := E + (id := E	`+6,d)$`	shift
S; id := E + (id := E +	`6,d)$`	shift
S; id := E + (id := E + num	`,d)$`	$E \longrightarrow$ num
S; id := E + (id := E + E	`,d)$`	$E \longrightarrow E$+E

S; id := E + (id := E + E	, d) \$	$E \rightarrow E$+E
S; id := E + (id := E	, d) \$	$S \rightarrow$ id:=E
S; id := E + (S	, d) \$	shift
S; id := E + (S,	d) \$	shift
S; id := E + (S, id	) \$	$E \rightarrow$ id
S; id := E + (S, E	) \$	shift
S; id := E + (S, E)	\$	$E \rightarrow$ (S;E)
S; id := E + E	\$	$E \rightarrow E$+E
S; id := E	\$	$S \rightarrow$ id:=E
S; S	\$	$S \rightarrow S$;S
S	\$	shift
S\$		$S' \rightarrow S$\$
S'		accept

$_0\ S' \rightarrow S\$$	$_5\ E \rightarrow$ num
$_1\ S \rightarrow S$; S	$_6\ E \rightarrow E + E$
$_2\ S \rightarrow$ id := E	$_7\ E \rightarrow$ (S , E)
$_3\ S \rightarrow$ print (L)	$_8\ L \rightarrow E$
$_4\ E \rightarrow$ id	$_9\ L \rightarrow L$, E

Use a DFA to choose the action; the stack only contains DFA states now.

Start with the initial state (s1) on the stack.

Lookup (stack top, next input symbol):

- shift(n): skip next input symbol and push state n
- reduce(k): rule k is $X \rightarrow \alpha$; pop $|\alpha|$ times; lookup (stack top, X) in table
- goto(n): push state n
- accept: report success
- error: report failure

DFA	terminals										non-terminals		
state	id	num	print	;	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9												g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

Error transitions omitted.

s_1	a := 7$
shift(4)	
s_1 s_4	:= 7$
shift(6)	
s_1 s_4 s_6	7$
shift(10)	
s_1 s_4 s_6 s_{10}	$
reduce(5): $E \longrightarrow$ num	
s_1 s_4 s_6 $\cancel{s_{10}}$	$
lookup(s_6,E) = goto(11)	
s_1 s_4 s_6 s_{11}	$
reduce(2): $S \longrightarrow$ id := E	
s_1 $\cancel{s_4}$ $\cancel{s_6}$ $\cancel{s_{11}}$	$
lookup(s_1,S) = goto(2)	
s_1 s_2	$
accept	

LR(1) is an algorithm that attempts to construct a parsing table:

- *Left-to-right parse*;
- *Rightmost-derivation*; and
- *1 symbol lookahead*.

If no conflicts (shift/reduce, reduce/reduce) arise, then we are happy; otherwise, fix grammar.

An LR(1) item (A $\rightarrow \boldsymbol{\alpha}$. $\boldsymbol{\beta\gamma}$, x) consists of

1. A grammar production, A $\rightarrow \boldsymbol{\alpha\beta\gamma}$
2. The RHS position, represented by '.'
3. A lookahead symbol, x

An LR(1) state is a set of LR(1) items.

The sequence $\boldsymbol{\alpha}$ is on top of the stack, and the head of the input is derivable from $\boldsymbol{\beta\gamma}$x. There are two cases for $\boldsymbol{\beta}$, terminal or non-terminal.

We first compute a set of LR(1) states from our grammar, and then use them to build a parse table. There are four kinds of entry to make:

1. goto: when β is non-terminal
2. shift: when β is terminal
3. reduce: when β is empty (the next state is the number of the production used)
4. accept: when we have A $\longrightarrow$ B . \$

Follow construction on the tiny grammar:

$_{0}\; S \rightarrow E\$$ $\qquad$ $_{2}\; E \rightarrow T$

$_{1}\; E \rightarrow T + E$ $\qquad$ $_{3}\; T \rightarrow \mathsf{x}$

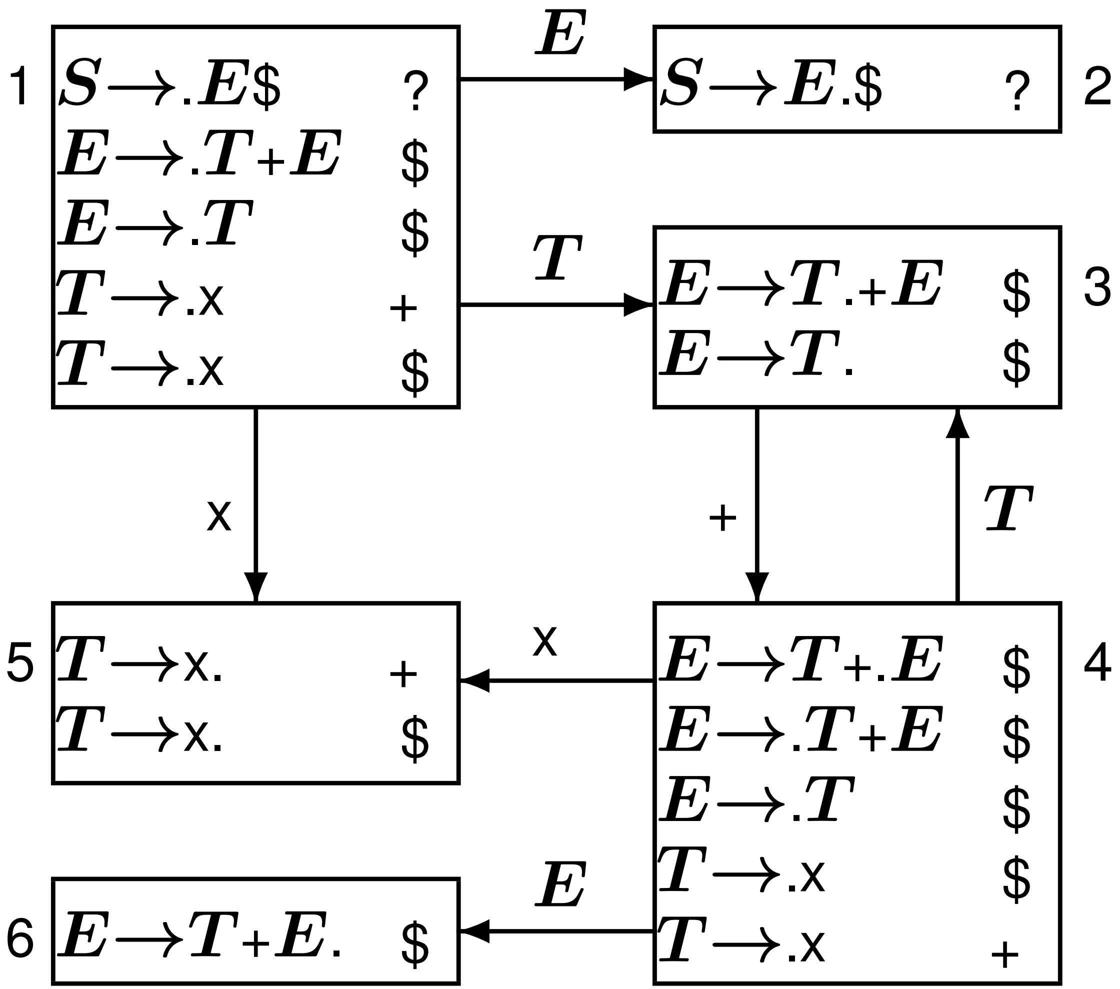
Constructing the LR(1) NFA:

- start with state $\boxed{S \to .\, E\$ \qquad \text{?}}$
- state $\boxed{A \to \alpha \,.\, B\, \beta \qquad \text{l}}$ has:
  - ϵ-successor $\boxed{B \to .\, \gamma \qquad \text{x}}$, if:
    - ∗ exists rule $B \to \gamma$, and
    - ∗ x ∈ lookahead(β)
  - B-successor $\boxed{A \to \alpha\, B \,.\, \beta \qquad \text{l}}$
- state $\boxed{A \to \alpha \,.\, \text{x}\, \beta \qquad \text{l}}$ has:

  x-successor $\boxed{A \to \alpha\, \text{x} \,.\, \beta \qquad \text{l}}$

Constructing the LR(1) DFA:

Standard power-set construction, “inlining” ϵ-transitions.



1
$S \to .E\$$?
$E \to .T{+}E$ \$
$E \to .T$ \$
$T \to .\text{x}$ +
$T \to .\text{x}$ \$

2
$S \to E.\$$?

3
$E \to T.{+}E$ \$
$E \to T.$ \$

4
$E \to T{+}.E$ \$
$E \to .T{+}E$ \$
$E \to .T$ \$
$T \to .\text{x}$ \$
$T \to .\text{x}$ +

5
$T \to \text{x}.$ +
$T \to \text{x}.$ \$

6
$E \to T{+}E.$ \$

	x	+	\$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

Conflicts

$A \rightarrow .B$ x
$A \rightarrow C.$ y

no conflict (lookahead decides)

$A \rightarrow .B$ x
$A \rightarrow C.$ x

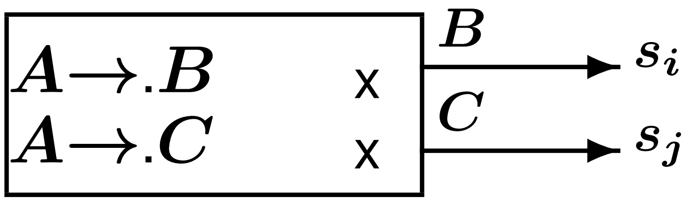
shift/reduce conflict

$A \rightarrow .\mathrm{x}$ y
$A \rightarrow C.$ x

shift/reduce conflict

$A \rightarrow B.$ x
$A \rightarrow C.$ x

reduce/reduce conflict



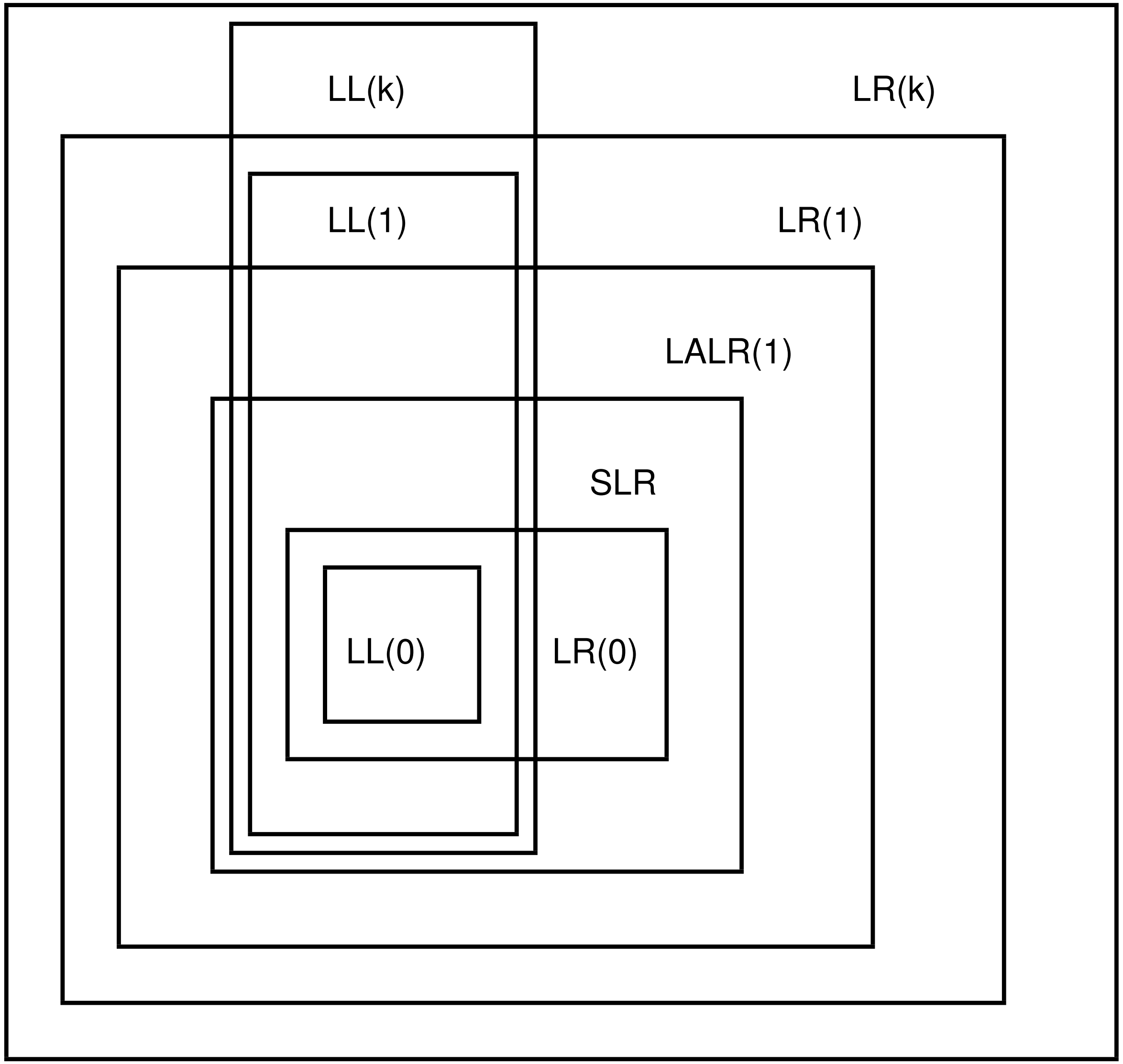
shift/shift conflict?

$\Rightarrow$ by construction of the DFA

we have $s_i = s_j$

LR(1) tables may become very large.

Parser generators use LALR(1), which merges states that are identical except for lookaheads.


LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR
LL(0)
LR(0)