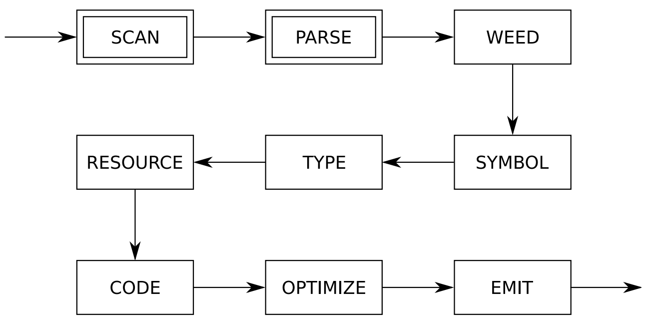
Parsing

COMP 520: Compiler Design (4 credits)

Professor Laurie Hendren

hendren@cs.mcgill.ca

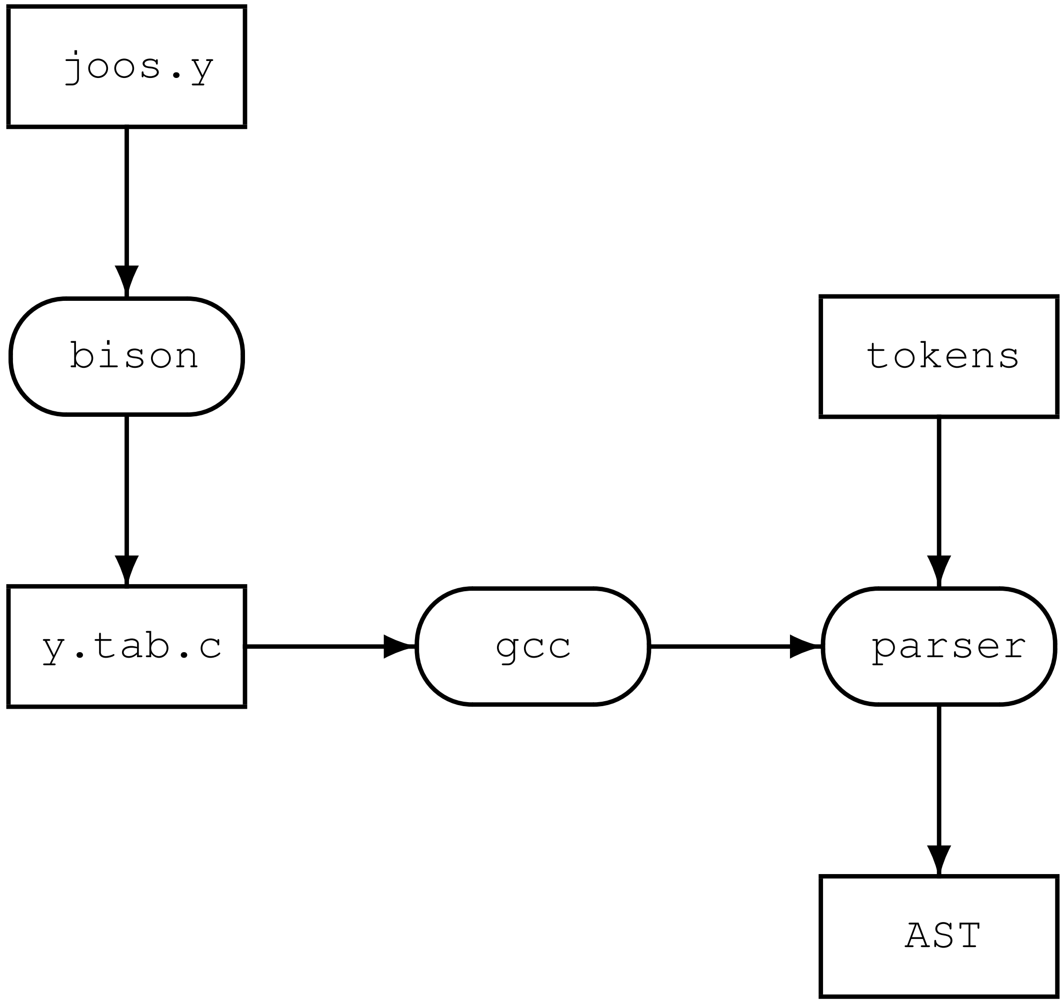


READING - very important for this phase

- 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.sable.mcgill.ca/%7Ehendren/520/2016`
  - flex
  - bison
  - SableCC

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, ...


joos.y
bison
y.tab.c
gcc
tokens
parser
AST

A *context-free* grammar is a 4-tuple (V, Σ, R, S), where we have:

- V, a set of *variables* (or *non-terminals*)
- Σ, a set of *terminals* such that $V \cap \Sigma = \emptyset$
- R, a set of *rules*, where the LHS is a variable in V and the RHS is a string of variables in V and terminals in Σ
- $S \in V$, the start variable

CFGs are stronger than regular expressions, and able to express recursively-defined constructs.

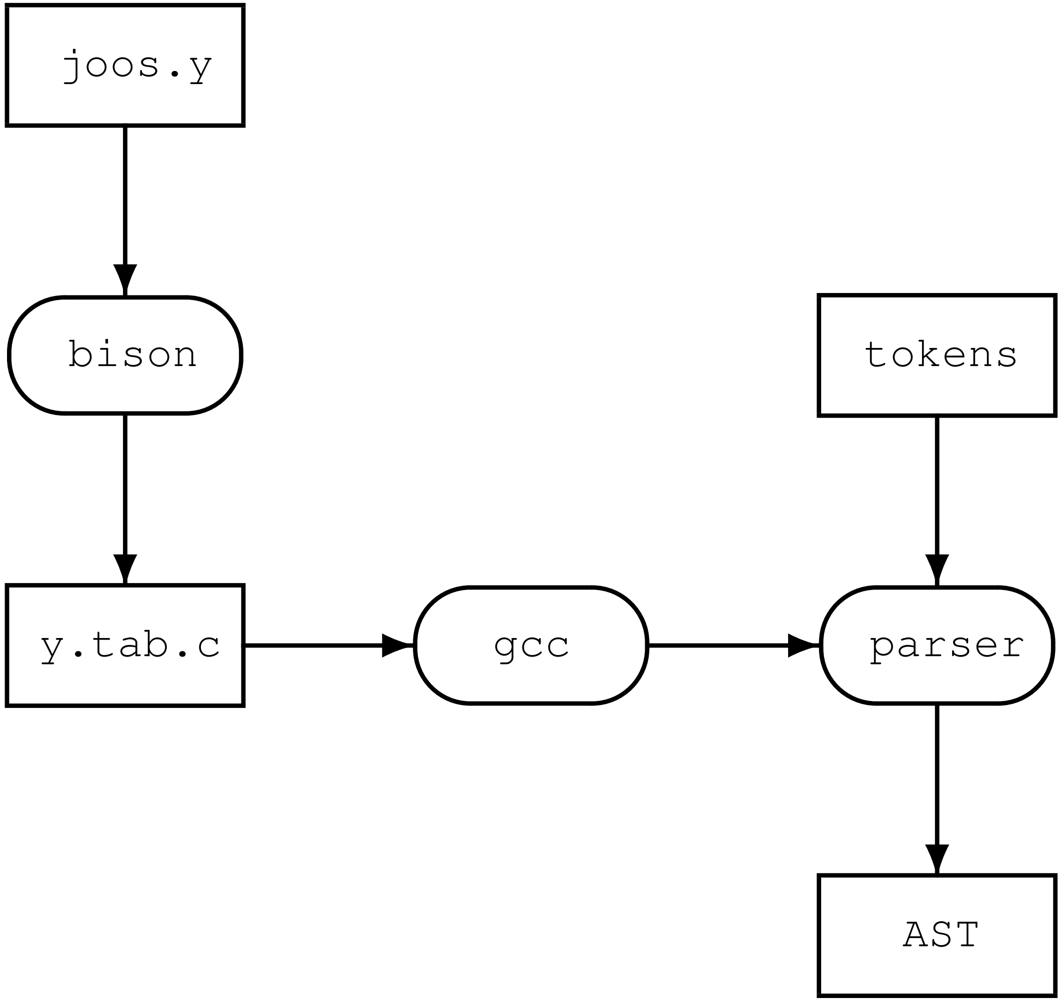
Example: we cannot write a regular expression for any number of matched parentheses:

`()`, `(())`, `((()))`, ...

Using a CFG:

$E \rightarrow (\, E \,) \mid \epsilon$

Automatic parser generators use CFGs as input and generate parsers using the machinery of a deterministic pushdown automaton.



By limiting the kind of CFG allowed, we get efficient parsers.

Simple CFG example:

$A \rightarrow$ a B

$A \rightarrow \epsilon$

$B \rightarrow$ b B

$B \rightarrow$ c

Alternatively:

$A \rightarrow$ a $B \mid \epsilon$

$B \rightarrow$ b $B \mid$ c

In both cases we specify $S = A$. Can you write this grammar as a regular expression?

We can perform a *rightmost derivation* by repeatedly replacing variables with their RHS until only terminals remain:

$\underline{A}$

a $\underline{B}$

a b $\underline{B}$

a b b $\underline{B}$

a b b c

Different grammar formalisms. First, consider BNF (Backus-Naur Form):

```
stmt ::= stmt_expr ";" |
         while_stmt |
         block |
         if_stmt
while_stmt ::= WHILE "(" expr ")" stmt
block ::= "{" stmt_list "}"
if_stmt ::= IF "(" expr ")" stmt |
     IF "(" expr ")" stmt ELSE stmt
```

We have four options for `stmt_list`:

1. `stmt_list ::= stmt_list stmt` | ϵ (0 or more, left-recursive)
2. `stmt_list ::= stmt stmt_list` | ϵ (0 or more, right-recursive)
3. `stmt_list ::= stmt_list stmt | stmt` (1 or more, left-recursive)
4. `stmt_list ::= stmt stmt_list | stmt` (1 or more, right-recursive)

Second, consider EBNF (Extended BNF):

BNF	derivations		EBNF
$\boldsymbol{A} \rightarrow \boldsymbol{A}$ a $\mid$ b (left-recursive)	b	$\underline{\boldsymbol{A}}$ a $\underline{\boldsymbol{A}}$ a a b a a	$\boldsymbol{A} \rightarrow$ b $\{$ a $\}$
$\boldsymbol{A} \rightarrow$ a $\boldsymbol{A} \mid$ b (right-recursive)	b	a $\underline{\boldsymbol{A}}$ a a $\underline{\boldsymbol{A}}$ a a b	$\boldsymbol{A} \rightarrow \{$ a $\}$ b

where '$\{$' and '$\}$' are like Kleene *'s in regular expressions.

Now, how to specify `stmt_list`:

Using EBNF repetition, our four choices for `stmt_list`

1. `stmt_list ::= stmt_list stmt` | ϵ (0 or more, left-recursive)
2. `stmt_list ::= stmt stmt_list` | ϵ (0 or more, right-recursive)
3. `stmt_list ::= stmt_list stmt | stmt` (1 or more, left-recursive)
4. `stmt_list ::= stmt stmt_list | stmt` (1 or more, right-recursive)

become:

1. `stmt_list ::= { stmt }`
2. `stmt_list ::= { stmt }`
3. `stmt_list ::= { stmt } stmt`
4. `stmt_list ::= stmt { stmt }`

EBNF also has an *optional*-construct. For example:

```
stmt_list ::= stmt stmt_list | stmt
```

could be written as:

```
stmt_list ::= stmt [ stmt_list ]
```

And similarly:

```
if_stmt ::= IF "(" expr ")" stmt |
    IF "(" expr ")" stmt ELSE stmt
```

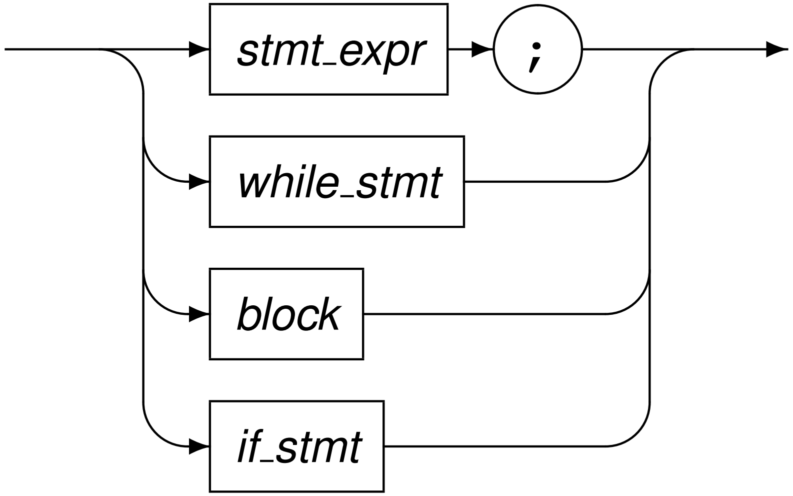
could be written as:

```
if_stmt ::=
    IF "(" expr ")" stmt [ ELSE stmt ]
```

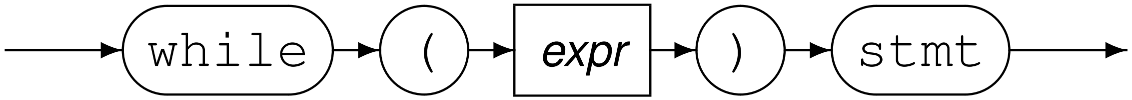
where '[' and ']' are like '?' in regular expressions.

Third, consider "railroad" syntax diagrams: (thanks rail.sty!)

stmt

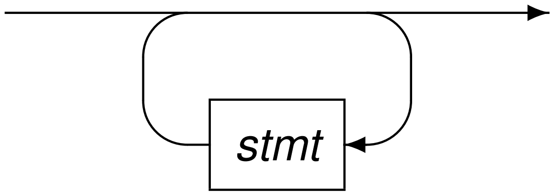


while_stmt

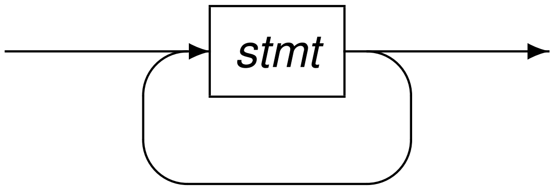


block

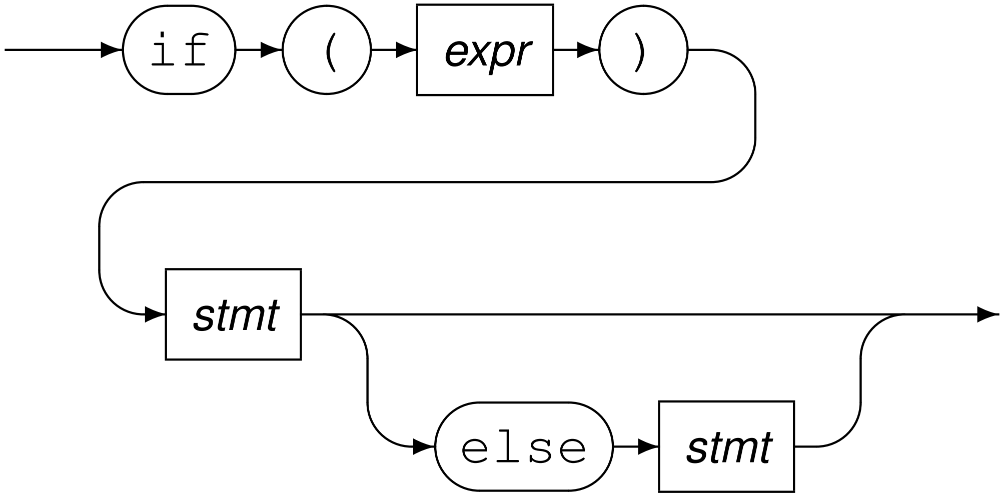
stmt_list (0 or more)



stmt_list (1 or more)



if_stmt



$S \rightarrow S\ ;\ S$ $\quad E \rightarrow \text{id}$ $\quad L \rightarrow E$

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

$S \rightarrow \text{print}\ (\ L\ )$ $\quad E \rightarrow E + E$

$E \rightarrow (\ S\ ,\ E\ )$

```
a := 7;
b := c + (d := 5 + 6, d)
```

$\underline{S}$ (rightmost derivation)

$S; \underline{S}$

$S;$ id := $\underline{E}$

$S;$ id := E + $\underline{E}$

$S;$ id := E + (S, $\underline{E}$)

$S;$ id := E + ($\underline{S}$, id)

$S;$ id := E + (id := $\underline{E}$, id)

$S;$ id := E + (id := E + $\underline{E}$, id)

$S;$ id := E + (id := $\underline{E}$ + num, id)

$S;$ id := $\underline{E}$ + (id := num + num, id)

$\underline{S};$ id := id + (id := num + num, id)

id := $\underline{E}$; id := id + (id := num + num, id)

id := num; id := id + (id := num + num, id)

$S \rightarrow S$; S

$S \rightarrow$ id := E

$S \rightarrow$ print (L)

$E \rightarrow$ id

$E \rightarrow$ num

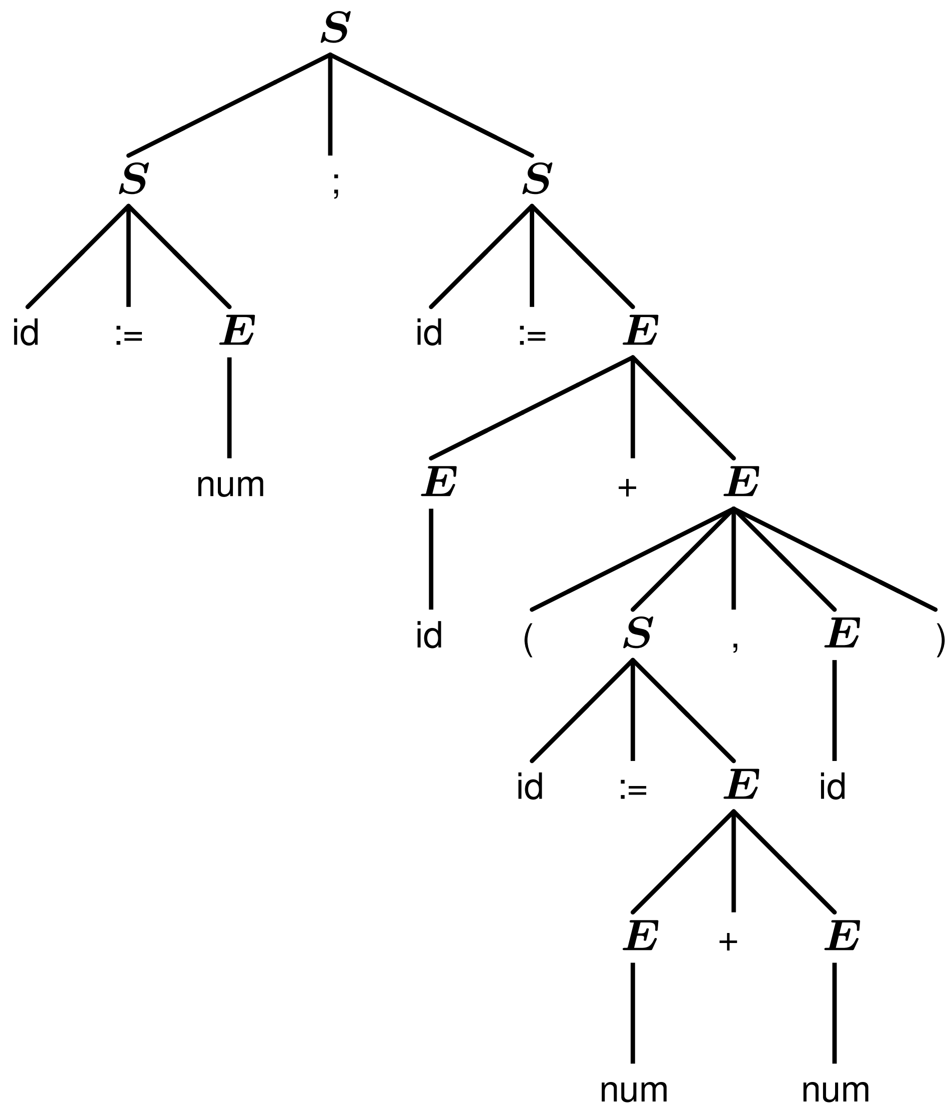
$E \rightarrow E$ + E

$E \rightarrow$ (S , E)

$L \rightarrow E$

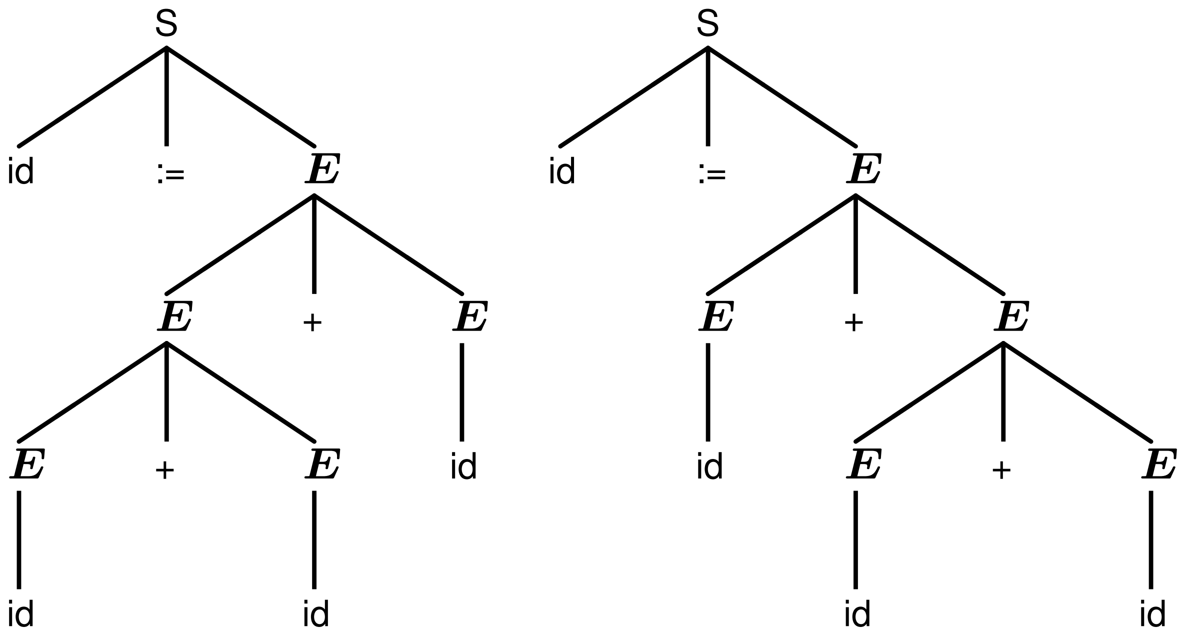
$L \rightarrow L$, E

```
a := 7;
b := c + (d := 5 + 6, d)
```



A grammar is *ambiguous* if a sentence has different parse trees:

`id := id + id + id`



The above is harmless, but consider:

`id := id - id - id`

`id := id + id * id`

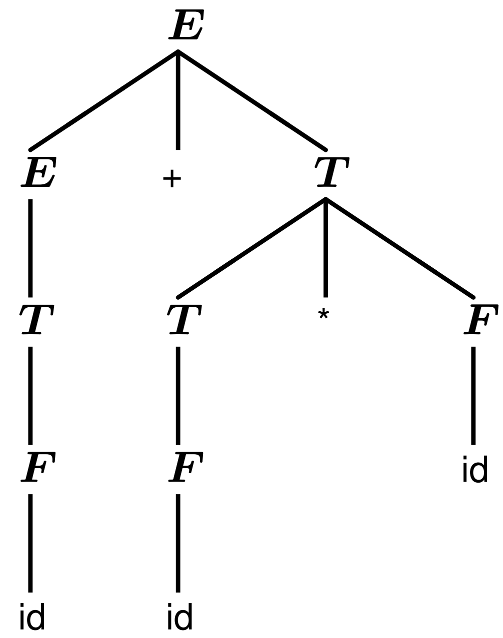
Clearly, we need to consider associativity and precedence when designing grammars.

An ambiguous grammar:

$$E \rightarrow \text{id} \qquad E \rightarrow E \;/\; E \qquad E \rightarrow (\; E \;)$$

$$E \rightarrow \text{num} \qquad E \rightarrow E + E$$

$$E \rightarrow E * E \qquad E \rightarrow E - E$$



may be rewritten to become unambiguous:

$$E \rightarrow E + T \qquad T \rightarrow T * F \qquad F \rightarrow \text{id}$$

$$E \rightarrow E - T \qquad T \rightarrow T \;/\; F \qquad F \rightarrow \text{num}$$

$$E \rightarrow T \qquad T \rightarrow F \qquad F \rightarrow (\; E \;)$$

There are fundamentally two kinds of parser:

1) Top-down, *predictive* or *recursive descent* parsers. Used in all languages designed by Wirth, e.g. Pascal, Modula, and Oberon.

One can (easily) write a predictive parser by hand, or generate one from an LL(k) grammar:

- *Left-to-right parse*;
- *Leftmost-derivation*; and
- *k symbol lookahead*.

Algorithm: look at beginning of input (up to k characters) and unambiguously expand leftmost non-terminal.

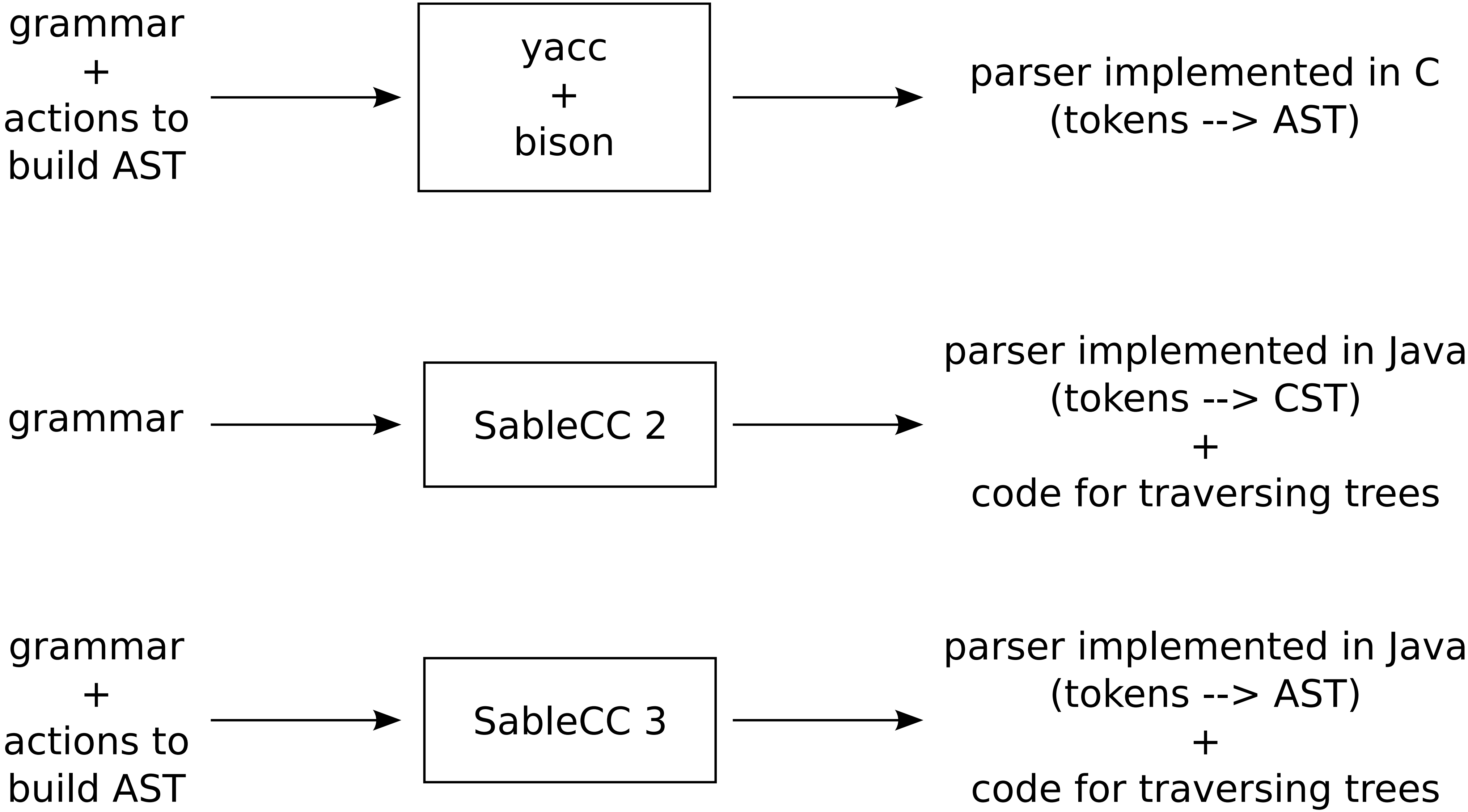
2) Bottom-up parsers.

Algorithm: look for a sequence matching RHS and reduce to LHS. Postpone any decision until entire RHS is seen, plus *k* tokens lookahead.

Can write a bottom-up parser by hand (tricky), or generate one from an LR(*k*) grammar (easy):

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

LALR Parser Tools



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	$E \rightarrow$ id	$L \rightarrow E$
$S \rightarrow$ id := E	$E \rightarrow$ num	$L \rightarrow L$, E
$S \rightarrow$ print (L)	$E \rightarrow E$ + 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)$`	shift
id := num	`; b:=c+(d:=5+6,d)$`	$E\rightarrow$num
id := E	`; b:=c+(d:=5+6,d)$`	$S\rightarrow$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)$`	$E\rightarrow$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)$`	shift
S; id := E + (id := num	`+6,d)$`	$E\rightarrow$num
S; id := E + (id := E	`+6,d)$`	shift
S; id := E + (id := E +	`6,d)$`	shift
S; id := E + (id := E + num	`,d)$`	$E\rightarrow$num
S; id := E + (id := E + E	`,d)$`	$E\rightarrow E$+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	;	,	+	:=	(	)	$	$\boldsymbol{S}$	$\boldsymbol{E}$	$\boldsymbol{L}$
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	

`bison` (`yacc`) is a parser generator:

- it inputs a grammar;
- it computes an LALR(1) parser table;
- it reports conflicts;
- it resolves conflicts using defaults (!); and
- it creates a C program.

Nobody writes (simple) parsers by hand anymore.

The grammar:

$$\begin{array}{lll} {}_1\ E \rightarrow \text{id} & {}_4\ E \rightarrow E \ / \ E & {}_7\ E \rightarrow (\ E\ ) \\ {}_2\ E \rightarrow \text{num} & {}_5\ E \rightarrow E + E & \\ {}_3\ E \rightarrow E * E & {}_6\ E \rightarrow E - E & \end{array}$$

is expressed in `bison` as:

```
%{
/* C declarations */
%}
/* Bison declarations; tokens come from lexer (scanner) */
%token tIDENTIFIER tINTCONST
%start exp
/* Grammar rules after the first %% */
%%
exp : tIDENTIFIER
    | tINTCONST
    | exp '*' exp
    | exp '/' exp
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
;
%% /* User C code after the second %% */
```

The grammar is ambiguous:

```
$ bison --verbose exp.y # --verbose produces exp.output
exp.y contains 16 shift/reduce conflicts.

$ cat exp.output
State 11 contains 4 shift/reduce conflicts.
State 12 contains 4 shift/reduce conflicts.
State 13 contains 4 shift/reduce conflicts.
State 14 contains 4 shift/reduce conflicts.

[...]
```

With more details about each state

```
state 11

    exp  ->  exp . '*' exp   (rule 3)
    exp  ->  exp '*' exp .   (rule 3) <-- problem is here
    exp  ->  exp . '/' exp   (rule 4)
    exp  ->  exp . '+' exp   (rule 5)
    exp  ->  exp . '-' exp   (rule 6)

    '*'          shift, and go to state 6
    '/'          shift, and go to state 7
    '+'          shift, and go to state 8
    '-'          shift, and go to state 9

    '*'          [reduce using rule 3 (exp)]
    '/'          [reduce using rule 3 (exp)]
    '+'          [reduce using rule 3 (exp)]
    '-'          [reduce using rule 3 (exp)]
    $default     reduce using rule 3 (exp)
```

Rewrite the grammar to force reductions:

$$E \rightarrow E + T \qquad T \rightarrow T * F \qquad F \rightarrow \text{id}$$

$$E \rightarrow E \text{ - } T \qquad T \rightarrow T \,/\, F \qquad F \rightarrow \text{num}$$

$$E \rightarrow T \qquad T \rightarrow F \qquad F \rightarrow (\, E \,)$$

```
%token tIDENTIFIER tINTCONST
%start exp
%%
exp : exp '+' term
    | exp '-' term
    | term
;
term : term '*' factor
     | term '/' factor
     | factor
;
factor : tIDENTIFIER
       | tINTCONST
       | '(' exp ')'
;
%%
```

Or use precedence directives:

```
%token tIDENTIFIER tINTCONST
%start exp
%left '+' '-'     /* left-associative, lower precedence */
%left '*' '/'     /* left-associative, higher precedence */

%%
exp : tIDENTIFIER
    | tINTCONST
    | exp '*' exp
    | exp '/' exp
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
;
%%
```

Which resolve shift/reduce conflicts:

```
Conflict in state 11 between rule 5 and token '+'
        resolved as reduce. <-- Reduce exp + exp . +
Conflict in state 11 between rule 5 and token '-'
        resolved as reduce. <-- Reduce exp + exp . -
Conflict in state 11 between rule 5 and token '*'
        resolved as shift.  <-- Shift exp + exp . *
Conflict in state 11 between rule 5 and token '/'
        resolved as shift.  <-- Shift exp + exp . /
```

Note that this is not the same state 11 as before.

The precedence directives are:

- `%left` *(left-associative)*
- `%right` *(right-associative)*
- `%nonassoc` *(non-associative)*

When constructing a parse table, an action is chosen based on the precedence of the last symbol on the right-hand side of the rule.

Precedences are ordered from lowest to highest on a linewise basis.

If precedences are equal, then:

- `%left` favors reducing
- `%right` favors shifting
- `%nonassoc` yields an error

This usually ends up working.

```
state 0
    tIDENTIFIER shift, and go to state 1
    tINTCONST   shift, and go to state 2
    '('         shift, and go to state 3
    exp         go to state 4
state 1
    exp  ->  tIDENTIFIER .   (rule 1)
    $default    reduce using rule 1 (exp)
state 2
    exp  ->  tINTCONST .   (rule 2)
    $default    reduce using rule 2 (exp)
...
state 14
    exp  ->  exp . '*' exp   (rule 3)
    exp  ->  exp . '/' exp   (rule 4)
    exp  ->  exp '/' exp .   (rule 4)
    exp  ->  exp . '+' exp   (rule 5)
    exp  ->  exp . '-' exp   (rule 6)
    $default    reduce using rule 4 (exp)
state 15
    $           go to state 16
state 16
    $default    accept
```

```
$ cat exp.y

%{
#include <stdio.h>   /* for printf */
extern char *yytext; /* string from scanner */
void yyerror() { printf ("syntax error before %s\n", yytext); }
%}
%union {
   int intconst;
   char *stringconst;
}
%token <intconst> tINTCONST
%token <stringconst> tIDENTIFIER
%start exp
%left '+' '-'
%left '*' '/'
%%
exp : tIDENTIFIER { printf ("load %s\n", $1); }
    | tINTCONST   { printf ("push %i\n", $1); }
    | exp '*' exp { printf ("mult\n"); }
    | exp '/' exp { printf ("div\n"); }
    | exp '+' exp { printf ("plus\n"); }
    | exp '-' exp { printf ("minus\n"); }
    | '(' exp ')' {}
;
%%
```

```
$ cat exp.l
%{
#include "y.tab.h"  /* for exp.y types */
#include <string.h> /* for strlen */
#include <stdlib.h> /* for malloc and atoi */
%}
%%
[ \t\n]+  /* ignore */;
"*"       return '*';
"/"       return '/';
"+"       return '+';
"-"       return '-';
"("       return '(';
")"       return ')';
0|([1-9][0-9]*) {
  yylval.intconst = atoi (yytext);
  return tINTCONST;
}
[a-zA-Z_][a-zA-Z0-9_]* {
  yylval.stringconst =
    (char *) malloc (strlen (yytext) + 1);
  sprintf (yylval.stringconst, "%s", yytext);
  return tIDENTIFIER;
}
.         /* ignore */
%%
```

```
$ cat main.c
void yyparse();
int main (void)
{
  yyparse ();
}
```

Using `flex`/`bison` to create a parser is simple:

```
$ flex exp.l
$ bison --yacc --defines exp.y # note compatability options
$ gcc lex.yy.c y.tab.c y.tab.h main.c -o exp -lfl
```

When input `a*(b-17) + 5/c`:

```
$ echo "a*(b-17) + 5/c" | ./exp
```

our `exp` parser outputs the correct order of operations:

```
load a
load b
push 17
minus
mult
push 5
load c
div
plus
```

You should confirm this for yourself!

If the input contains syntax errors, then the `bison`-generated parser calls `yyerror` and stops.

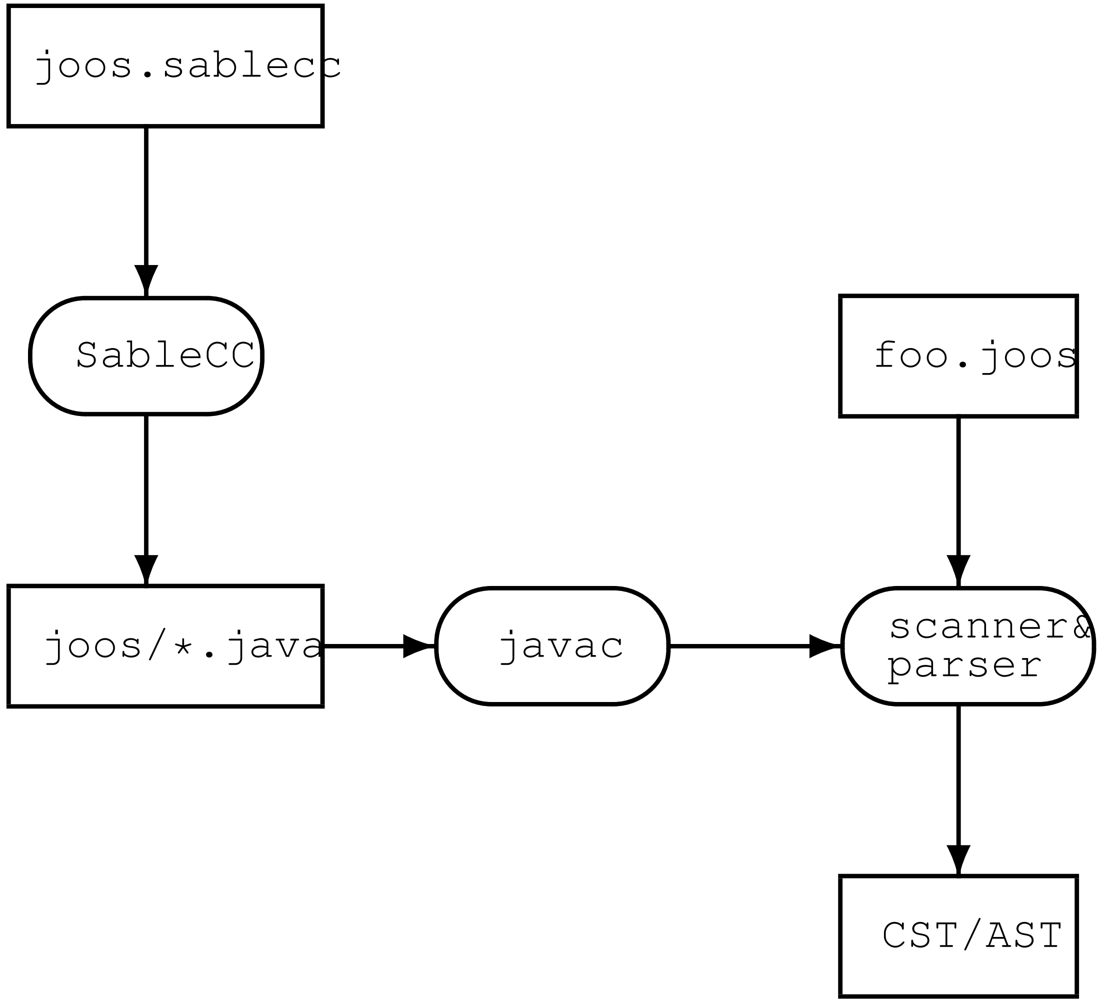
We may ask it to recover from the error:

```
exp : tIDENTIFIER { printf ("load %s\n", $1); }
    ...
    | '(' exp ')'
    | error { yyerror(); }
;
```

and on input `a@(b-17) ++ 5/c` get the output:

```
load a
syntax error before (
syntax error before (
syntax error before (
syntax error before b
push 17
minus
syntax error before )
syntax error before )
syntax error before +
plus
push 5
load c
div
plus
```

SableCC (by Etienne Gagnon, McGill alumnus) is a *compiler compiler*: it takes a grammatical description of the source language as input, and generates a lexer (scanner) and parser for it.



The SableCC 2 grammar for our Tiny language:

```
Package tiny;
Helpers
  tab   = 9;
  cr    = 13;
  lf    = 10;
  digit = ['0'..'9'];
  lowercase = ['a'..'z'];
  uppercase = ['A'..'Z'];
  letter  = lowercase | uppercase;
  idletter = letter | '_';
  idchar  = letter | '_' | digit;
Tokens
  eol   = cr | lf | cr lf;
  blank = ' ' | tab;
  star  = '*';
  slash = '/';
  plus  = '+';
  minus = '-';
  l_par = '(';
  r_par = ')';
  number  = '0'| [digit-'0'] digit*;
  id    = idletter idchar*;
Ignored Tokens
  blank, eol;
```

```
Productions
  exp =
      {plus}    exp plus factor |
      {minus}   exp minus factor |
      {factor}  factor;

  factor  =
      {mult}    factor star term |
      {divd}    factor slash term |
      {term}    term;

  term  =
      {paren}   l_par exp r_par |
      {id}      id |
      {number}  number;
```

Version 2 produces parse trees, a.k.a. concrete syntax trees (CSTs).

The SableCC 3 grammar for our Tiny language:

```
Productions
cst_exp {-> exp} =
  {cst_plus}    cst_exp plus factor
                {-> New exp.plus(cst_exp.exp,factor.exp)} |
  {cst_minus}   cst_exp minus factor
                {-> New exp.minus(cst_exp.exp,factor.exp)} |
  {factor}      factor {-> factor.exp};

factor {-> exp} =
  {cst_mult}    factor star term
                {-> New exp.mult(factor.exp,term.exp)} |
  {cst_divd}    factor slash term
                {-> New exp.divd(factor.exp,term.exp)} |
  {term}        term {-> term.exp};

term {-> exp} =
  {paren}       l_par cst_exp r_par {-> cst_exp.exp} |
  {cst_id}      id {-> New exp.id(id)} |
  {cst_number}  number {-> New exp.number(number)};
```

```
Abstract Syntax Tree
exp =
  {plus}      [l]:exp [r]:exp |
  {minus}     [l]:exp [r]:exp |
  {mult}      [l]:exp [r]:exp |
  {divd}      [l]:exp [r]:exp |
  {id}        id |
  {number}    number;
```

Version 3 generates abstract syntax trees (ASTs).

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

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	$E \rightarrow$ id	$L \rightarrow E$
$S \rightarrow$ id := E	$E \rightarrow$ num	$L \rightarrow L$, E
$S \rightarrow$ print (L)	$E \rightarrow E$ + 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)$`	shift
id := num	`; b:=c+(d:=5+6,d)$`	$\boldsymbol{E}\rightarrow$num
id := $\boldsymbol{E}$	`; b:=c+(d:=5+6,d)$`	$\boldsymbol{S}\rightarrow$id:=$\boldsymbol{E}$
$\boldsymbol{S}$	`; b:=c+(d:=5+6,d)$`	shift
$\boldsymbol{S}$;	`b:=c+(d:=5+6,d)$`	shift
$\boldsymbol{S}$; id	`:=c+(d:=5+6,d)$`	shift
$\boldsymbol{S}$; id :=	`c+(d:=5+6,d)$`	shift
$\boldsymbol{S}$; id := id	`+(d:=5+6,d)$`	$\boldsymbol{E}\rightarrow$id
$\boldsymbol{S}$; id := $\boldsymbol{E}$	`+(d:=5+6,d)$`	shift
$\boldsymbol{S}$; id := $\boldsymbol{E}$ +	`(d:=5+6,d)$`	shift
$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (	`d:=5+6,d)$`	shift
$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id	`:=5+6,d)$`	shift
$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id :=	`5+6,d)$`	shift
$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := num	`+6,d)$`	$\boldsymbol{E}\rightarrow$num
$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\boldsymbol{E}$	`+6,d)$`	shift
$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\boldsymbol{E}$ +	`6,d)$`	shift
$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\boldsymbol{E}$ + num	`,d)$`	$\boldsymbol{E}\rightarrow$num
$\boldsymbol{S}$; id := $\boldsymbol{E}$ + (id := $\boldsymbol{E}$ + $\boldsymbol{E}$	`,d)$`	$\boldsymbol{E}\rightarrow\boldsymbol{E}$+$\boldsymbol{E}$

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 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			

DFA state	id	num	print	;	,	+	:=	(	)	$	S	E	L
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:

$_0\ S \rightarrow E\$$ $\qquad$ $_2\ E \rightarrow T$

$_1\ E \rightarrow T + E$ $\qquad$ $_3\ T \rightarrow \mathrm{x}$

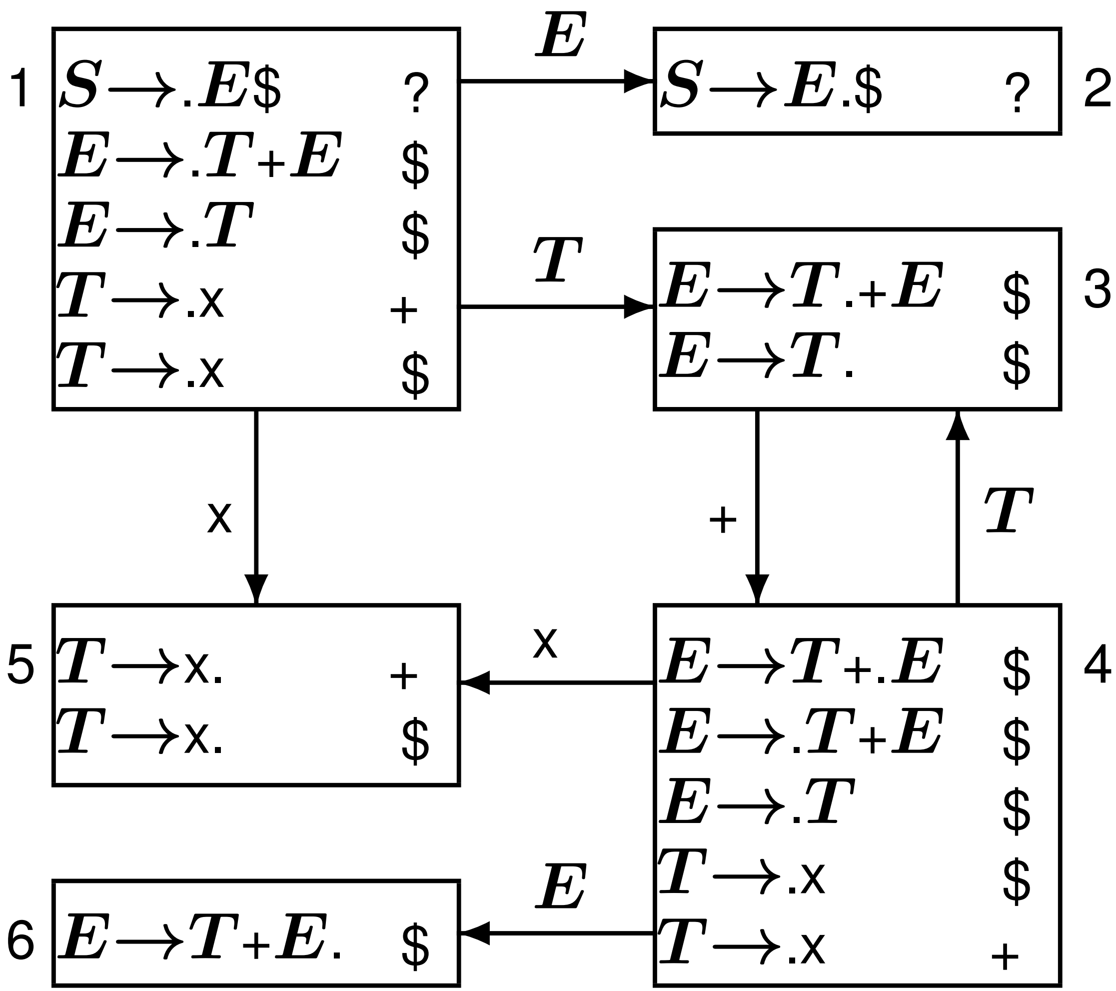
Constructing the LR(1) NFA:

- start with state $\boxed{\boldsymbol{S} \rightarrow .\ \boldsymbol{E}\$ \quad ?}$
- state $\boxed{\boldsymbol{A} \rightarrow \boldsymbol{\alpha}\ .\ \boldsymbol{B}\ \boldsymbol{\beta} \quad \mathsf{l}}$ has:
  - ϵ-successor $\boxed{\boldsymbol{B} \rightarrow .\ \gamma \quad \mathsf{x}}$, if:
    - exists rule $\boldsymbol{B} \rightarrow \boldsymbol{\gamma}$, and
    - x $\in$ lookahead($\boldsymbol{\beta}$)
  - $\boldsymbol{B}$-successor $\boxed{\boldsymbol{A} \rightarrow \boldsymbol{\alpha}\ \boldsymbol{B}\ .\ \boldsymbol{\beta} \quad \mathsf{l}}$
- state $\boxed{\boldsymbol{A} \rightarrow \boldsymbol{\alpha}\ .\ \mathsf{x}\ \boldsymbol{\beta} \quad \mathsf{l}}$ has:

  x-successor $\boxed{\boldsymbol{A} \rightarrow \boldsymbol{\alpha}\ \mathsf{x}\ .\ \boldsymbol{\beta} \quad \mathsf{l}}$

Constructing the LR(1) DFA:

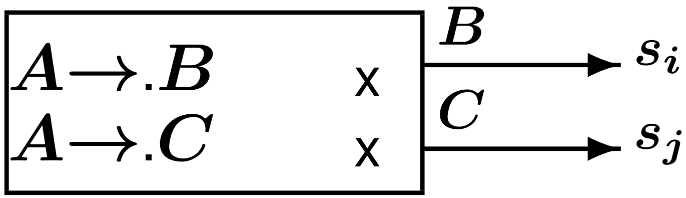
Standard power-set construction, “inlining” ϵ-transitions.



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

Conflicts

$A \rightarrow .B$ $A \rightarrow C.$	x y	no conflict (lookahead decides)
$A \rightarrow .B$ $A \rightarrow C.$	x x	shift/reduce conflict
$A \rightarrow .\mathrm{x}$ $A \rightarrow C.$	y x	shift/reduce conflict
$A \rightarrow B.$ $A \rightarrow C.$	x x	reduce/reduce conflict



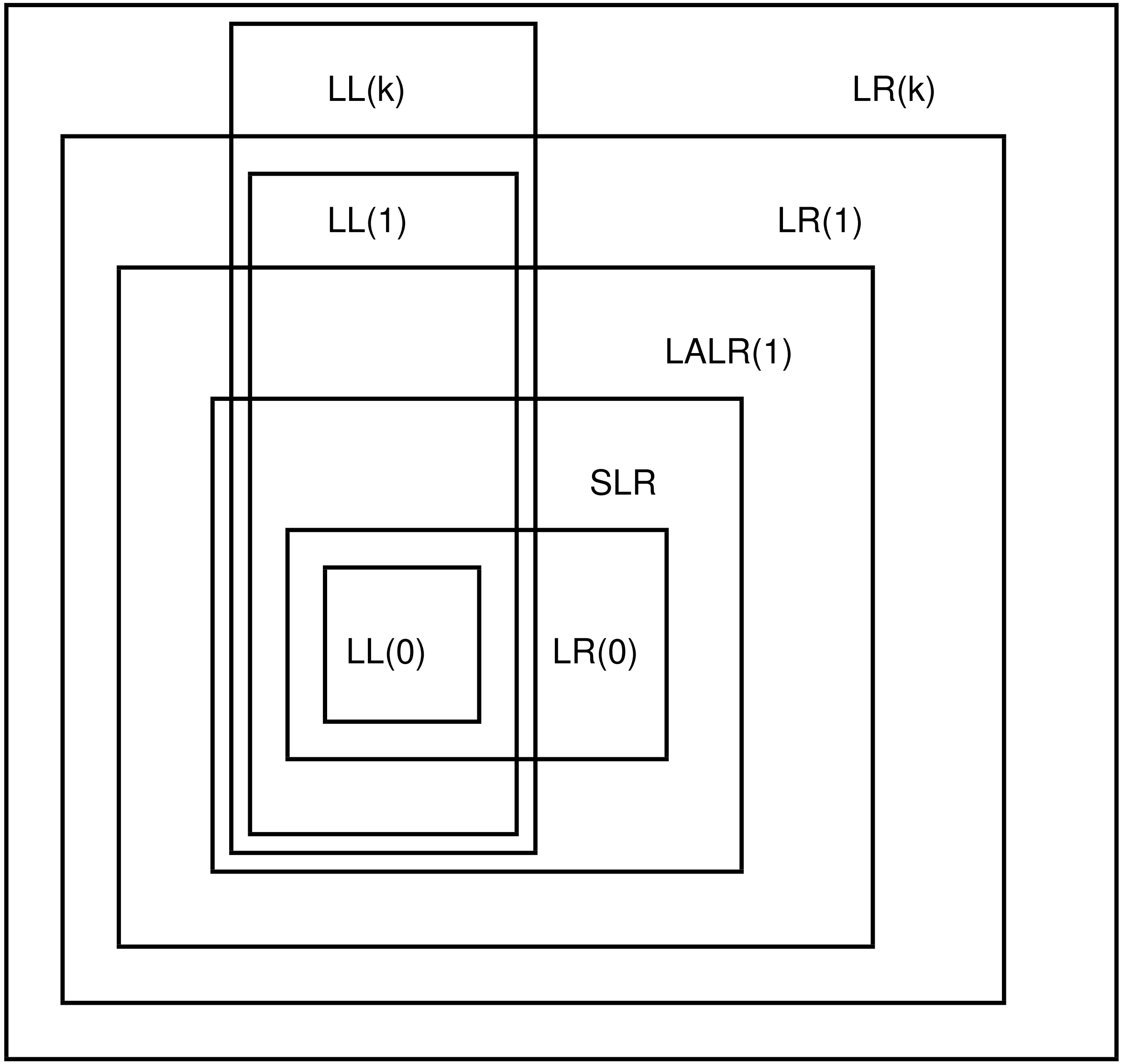
shift/shift conflict?

$\Longrightarrow$ by construction of the <u>D</u>FA

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)