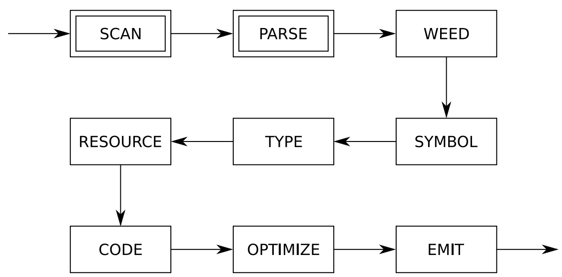
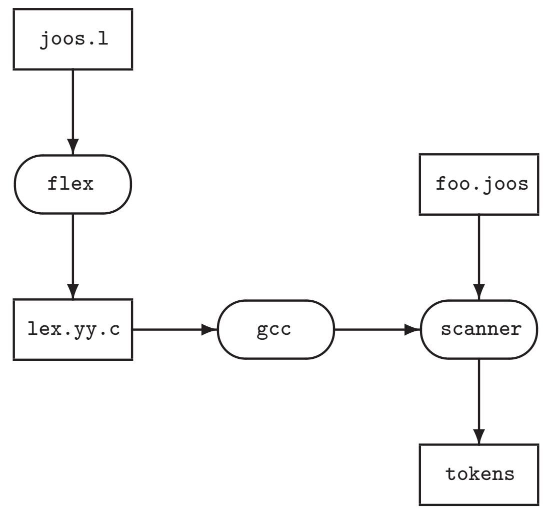
Scanners and parsers



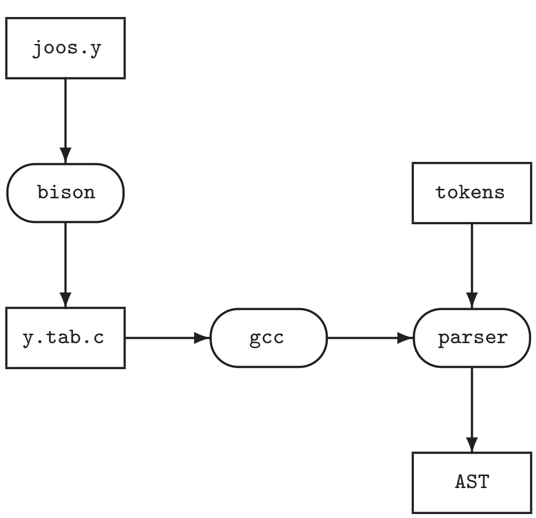
A *scanner* or *lexer* transforms a string of characters into a string of tokens:

- uses a combination of *deterministic finite automata* (DFA);
- plus some glue code to make it work;
- can be generated by tools like `flex` (or `lex`), `JFlex`, ...



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, ...



Tokens are defined by *regular expressions*:

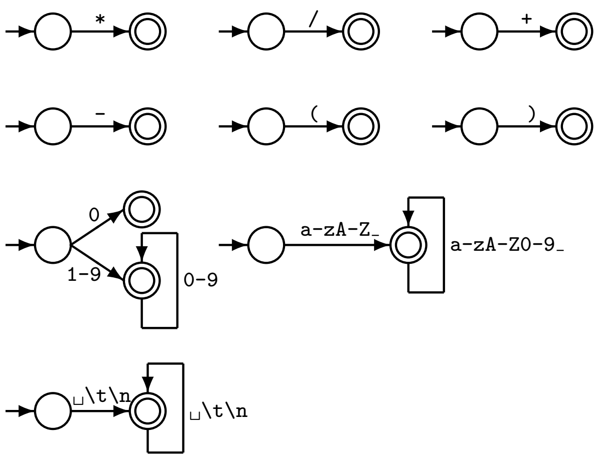
- $\emptyset$, the empty set: a language with no strings
- ε, the empty string
- $\boldsymbol{a}$, where $\boldsymbol{a} \in \boldsymbol{\Sigma}$ and $\boldsymbol{\Sigma}$ is our alphabet
- $\boldsymbol{M}|\boldsymbol{N}$, alternation: either $\boldsymbol{M}$ or $\boldsymbol{N}$
- $\boldsymbol{M} \cdot \boldsymbol{N}$, concatenation: $\boldsymbol{M}$ followed by $\boldsymbol{N}$
- $\boldsymbol{M}^*$, zero or more occurences of $\boldsymbol{M}$

where $\boldsymbol{M}$ and $\boldsymbol{N}$ are both regular expressions. What are $\boldsymbol{M}?$ and $\boldsymbol{M}^+$?

We can write regular expressions for the tokens in our source language using standard POSIX notation:

- simple operators: `"*"`, `"/"`, `"+"`, `"-"`
- parentheses: `"("`, `")"`
- integer constants: `0|([1-9][0-9]*)`
- identifiers: `[a-zA-Z_][a-zA-Z0-9_]*`
- white space: `[␣\t\n]+`

`flex` accepts a list of regular expressions (regex), converts each regex internally to an NFA (Thompson construction), and then converts each NFA to a DFA (see Appel, Ch. 2):



Each DFA has an associated *action*.

Given DFAs $D_1, \ldots, D_n$, ordered by the input rule order, the behaviour of a `flex`-generated scanner on an input string is:

```
while input is not empty do
    s_i := the longest prefix that D_i accepts
    l := max{|s_i|}
    if l > 0 then
        j := min{i : |s_i| = l}
        remove s_j from input
        perform the j^th action
    else (error case)
        move one character from input to output
    end
end
```

In English:

- The *longest* initial substring match forms the next token, and it is subject to some action
- The *first* rule to match breaks any ties
- Non-matching characters are echoed back

Why the "longest match" principle?

Example: keywords

```
[ \t]+
    /* ignore */;
...
import
    return tIMPORT;
...
[a-zA-Z_][a-zA-Z0-9_]* {
    yylval.stringconst = (char *)malloc(strlen(yytext)+1);
    printf(yylval.stringconst,"%s",yytext);
    return tIDENTIFIER; }
```

Want to match ``‘‘importedFiles’’`` as `tIDENTIFIER(importedFiles)` and not as `tIMPORT tIDENTIFIER(edFiles)`.

Because we prefer longer matches, we get the right result.

Why the "first match" principle?

Again — Example: keywords

```
[ \t]+
    /* ignore */;
...
continue
    return tCONTINUE;
...
[a-zA-Z_][a-zA-Z0-9_]* {
    yylval.stringconst = (char *)malloc(strlen(yytext)+1);
    printf(yylval.stringconst,"%s",yytext);
    return tIDENTIFIER; }
```

Want to match ``‘‘continue foo’’`` as `tCONTINUE tIDENTIFIER(foo)` and not as `tIDENTIFIER(continue) tIDENTIFIER(foo)`.

"First match" rule gives us the right answer: When both `tCONTINUE` and `tIDENTIFIER` match, prefer the first.

When "first longest match" (flm) is not enough, look-ahead may help.

FORTRAN allows for the following tokens:
`.EQ., 363, 363., .363`

flm analysis of `363.EQ.363` gives us:
`tFLOAT(363) E Q tFLOAT(0.363)`

What we actually want is:
`tINTEGER(363) tEQ tINTEGER(363)`

`flex` allows us to use look-ahead, using '/':

```
363/.EQ. return tINTEGER;
```


Another example taken from FORTRAN:
Fortran ignores whitespace

1. `DO5I = 1.25` ⇝ `DO5I=1.25`
   in C: `do5i = 1.25;`
2. `DO 5 I = 1,25` ⇝ `DO5I=1,25`
   in C: `for(i=1;i<25;++i){...}`
   (5 is interpreted as a line number here)

Case 1: flm analysis correct:
`tID(DO5I) tEQ tREAL(1.25)`

Case 2: want:
`tDO tINT(5) tID(I) tEQ tINT(1) tCOMMA tINT(25)`

Cannot make decision on `tDO` until we see the comma!

Look-ahead comes to the rescue:

```
DO/({letter}|{digit})*=({letter}|{digit})*,
    return tDO;                             ↑
```



```
$ cat print_tokens.l # flex source code

/* includes and other arbitrary C code */
%{
#include <stdio.h> /* for printf */
%}

/* helper definitions */
DIGIT [0-9]

/* regex + action rules come after the first %% */
%%

[ \t\n]+        printf ("white space, length %i\n", yyleng);

"*"             printf ("times\n");
"/"             printf ("div\n");
"+"             printf ("plus\n");
"-"             printf ("minus\n");
"("             printf ("left parenthesis\n");
")"             printf ("right parenthesis\n");

0|([1-9]{DIGIT}*) printf ("integer constant: %s\n", yytext);
[a-zA-Z_][a-zA-Z0-9_]* printf ("identifier: %s\n", yytext);

%%
/* user code comes after the second %% */

main () {
  yylex ();
}
```


Using `flex` to create a scanner is really simple:

```
$ emacs print_tokens.l
$ flex print_tokens.l
$ gcc -o print_tokens lex.yy.c -lfl
```

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

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

our `print_tokens` scanner outputs:

```
identifier: a
times
left parenthesis
identifier: b
minus
integer constant: 17
right parenthesis
white space, length 1
plus
white space, length 1
integer constant: 5
div
identifier: c
white space, length 1
```

You should confirm this for yourself!

Count lines and characters:

```
%{
int lines = 0, chars = 0;
%}

%%
\n      lines++; chars++;
.       chars++;

%%
main () {
  yylex ();
  printf ("#lines = %i, #chars = %i\n", lines, chars);
}
```

Remove vowels and increment integers:

```
%{
#include <stdlib.h> /* for atoi */
#include <stdio.h>  /* for printf */
%}

%%
[aeiouy]      /* ignore */
[0-9]+        printf ("%i", atoi (yytext) + 1);

%%
main () {
  yylex ();
}
```


A *context-free* grammar is a 4-tuple $(\boldsymbol{V}, \boldsymbol{\Sigma}, \boldsymbol{R}, \boldsymbol{S})$, where we have:

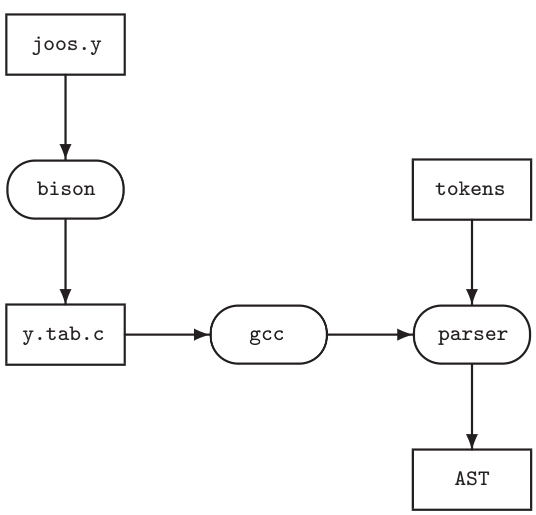
- $\boldsymbol{V}$, a set of *variables* (or *non-terminals*)
- $\boldsymbol{\Sigma}$, a set of *terminals* such that $\boldsymbol{V} \cap \boldsymbol{\Sigma} = \emptyset$
- $\boldsymbol{R}$, a set of *rules*, where the LHS is a variable in $\boldsymbol{V}$ and the RHS is a string of variables in $\boldsymbol{V}$ and terminals in $\boldsymbol{\Sigma}$
- $\boldsymbol{S} \in \boldsymbol{V}$, 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:
$\boldsymbol{E} \rightarrow (\ \boldsymbol{E}\ ) \mid \boldsymbol{\epsilon}$

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:

$\boldsymbol{A} \rightarrow \text{a } \boldsymbol{B}$

$\boldsymbol{A} \rightarrow \boldsymbol{\epsilon}$

$\boldsymbol{B} \rightarrow \text{b } \boldsymbol{B}$

$\boldsymbol{B} \rightarrow \text{c}$

Alternatively:

$\boldsymbol{A} \rightarrow \text{a } \boldsymbol{B} \mid \boldsymbol{\epsilon}$

$\boldsymbol{B} \rightarrow \text{b } \boldsymbol{B} \mid \text{c}$

In both cases we specify $\boldsymbol{S} = \boldsymbol{A}$. 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:

$\underline{\boldsymbol{A}}$

$\text{a } \underline{\boldsymbol{B}}$

$\text{a b } \underline{\boldsymbol{B}}$

$\text{a b b } \underline{\boldsymbol{B}}$

a b b c

There are several 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
   $\rightarrow$ 0 or more, left-recursive
2. `stmt_list ::= stmt stmt_list |` ϵ
   $\rightarrow$ 0 or more, right-recursive
3. `stmt_list ::= stmt_list stmt | stmt`
   $\rightarrow$ 1 or more, left-recursive
4. `stmt_list ::= stmt stmt_list | stmt`
   $\rightarrow$ 1 or more, right-recursive

Second, consider EBNF (Extended BNF):

BNF	derivations		EBNF
$\boldsymbol{A} \rightarrow \boldsymbol{A}$ a | 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 $\boldsymbol{A}$ | 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. Using EBNF repetition, our four choices for `stmt_list` become:

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

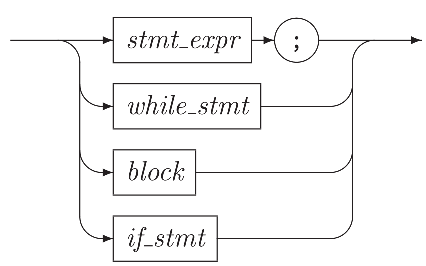
could be written as:

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

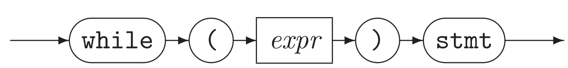
where '[' and ']' are like '?' in regular expressions.

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

stmt

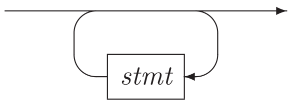


while_stmt

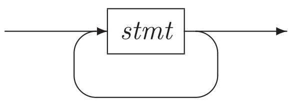


block

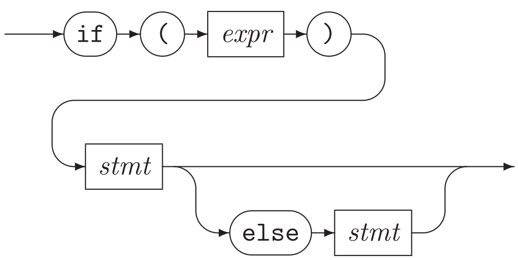
stmt_list (0 or more)



stmt_list (1 or more)



if_stmt



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

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

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

$\boldsymbol{E} \rightarrow (\ \boldsymbol{S}\ ,\ \boldsymbol{E}\ )$

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

$\underline{\boldsymbol{S}}$ (rightmost derivation)

$\boldsymbol{S};\ \underline{\boldsymbol{S}}$

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

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

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

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

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

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

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

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

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

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

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

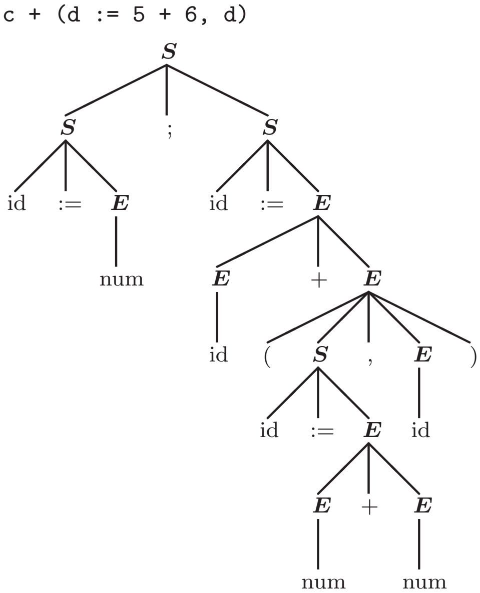
$\boldsymbol{S} \rightarrow \boldsymbol{S}\ ;\ \boldsymbol{S}$ $\quad \boldsymbol{E} \rightarrow \text{id}$ $\quad \boldsymbol{L} \rightarrow \boldsymbol{E}$

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

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

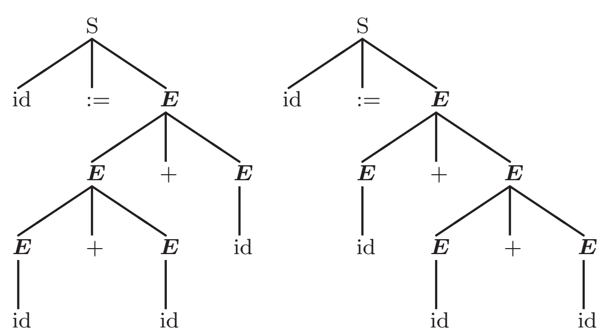
$\boldsymbol{E} \rightarrow (\ \boldsymbol{S}\ ,\ \boldsymbol{E}\ )$

```
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
```

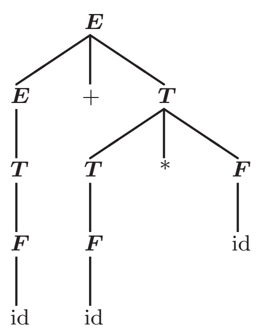
Clearly, we need to consider associativity and precedence when designing grammars.

An ambiguous grammar:

$E \rightarrow$ id	$E \rightarrow E \mathbin{/} E$	$E \rightarrow (E)$
$E \rightarrow$ num	$E \rightarrow E + E$	
$E \rightarrow E * E$	$E \rightarrow E - E$	

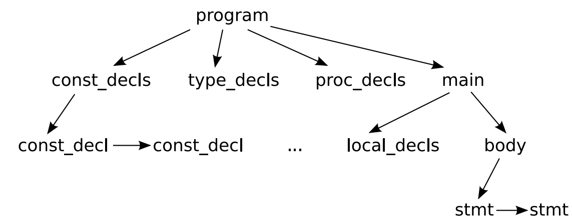
may be rewritten to become unambiguous:

$E \rightarrow E + T$	$T \rightarrow T * F$	$F \rightarrow$ id
$E \rightarrow E - T$	$T \rightarrow T \mathbin{/} F$	$F \rightarrow$ num
$E \rightarrow T$	$T \rightarrow F$	$F \rightarrow (E)$



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.*

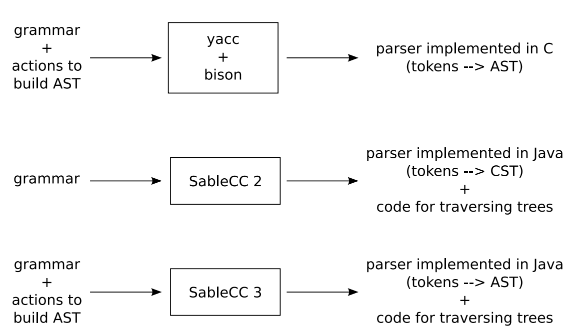
Algorithm: look at beginning of input (up to k characters) and unambiguously expand leftmost non-terminal.

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.*



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)$`	shift
id := num	`; b:=c+(d:=5+6,d)$`	$E \to$num
id := E	`; b:=c+(d:=5+6,d)$`	$S \to$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)$`	$E \to$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)$`	shift
S; id := E + (id := num	`+6,d)$`	$E \to$num
S; id := E + (id := E	`+6,d)$`	shift
S; id := E + (id := E +	`6,d)$`	shift
S; id := E + (id := E + num	`,d)$`	$E \to$num
S; id := E + (id := E + E	`,d)$`	$E \to E$+E
S; id := E + (id := E	`,d)$`	$S \to$id:=E
S; id := E + (S	`,d)$`	shift
S; id := E + (S,	`d)$`	shift
S; id := E + (S, id	`)$`	$E \to$id
S; id := E + (S, E	`)$`	shift
S; id := E + (S, E)	`$`	$E \to$(S;E)
S; id := E + E	`$`	$E \to E$+E
S; id := E	`$`	$S \to$id:=E
S; S	`$`	$S \to S$;S
S	`$`	shift
S\$		$S' \to S$\$
S'		accept

$_0$ $S' \to S$\$

$_1$ $S \to S$; S

$_2$ $S \to$ id := E

$_3$ $S \to$ print (L)

$_4$ $E \to$ id

$_5$ $E \to$ num

$_6$ $E \to E + E$

$_7$ $E \to$ (S , E)

$_8$ $L \to E$

$_9$ $L \to L$, 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 $X \to \alpha$; 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): $E \to$ num

s_1 s_4 s_6 ~~s_{10}~~ \$

lookup(s_6,E) = goto(11)

s_1 s_4 s_6 s_{11} \$

reduce(2): $S \to$ id := E

s_1 ~~s_4~~ ~~s_6~~ ~~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 $\boldsymbol{\beta}$ is non-terminal
2. shift: when $\boldsymbol{\beta}$ is terminal
3. reduce: when $\boldsymbol{\beta}$ is empty (the next state is the number of the production used)
4. accept: when we have A $\rightarrow$ B . \$

Follow construction on the tiny grammar:

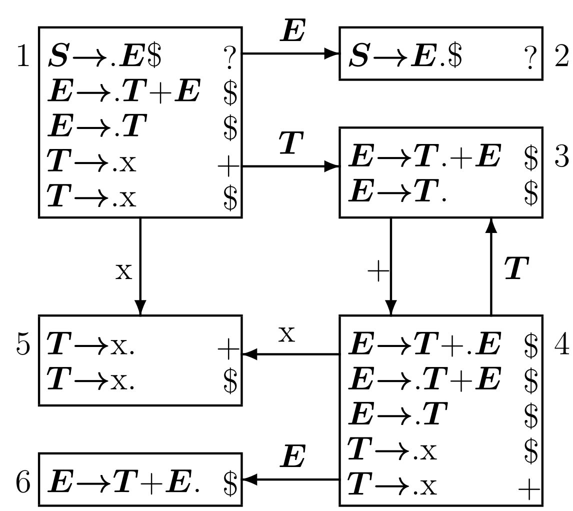
$_0$ $\boldsymbol{S} \rightarrow \boldsymbol{E}\$$	$_2$ $\boldsymbol{E} \rightarrow \boldsymbol{T}$
$_1$ $\boldsymbol{E} \rightarrow \boldsymbol{T} + \boldsymbol{E}$	$_3$ $\boldsymbol{T} \rightarrow$ x

Constructing the LR(1) NFA:

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

  x-successor [$\boldsymbol{A} \rightarrow \boldsymbol{\alpha}$ x . $\boldsymbol{\beta}$ l]

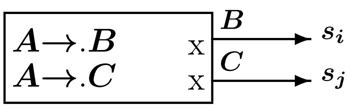
Constructing the LR(1) DFA:
Standard power-set construction, "inlining" $\boldsymbol{\epsilon}$-transitions.



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

Conflicts

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

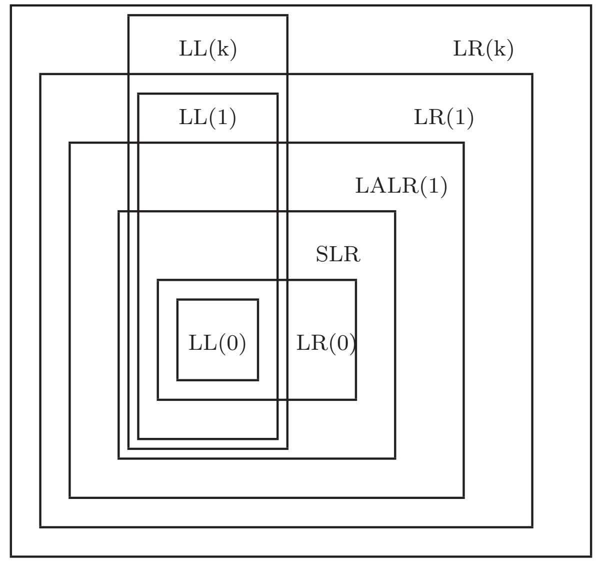


shift/shift conflict?

$\Rightarrow$ 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.



`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:

1 $E \to \mathrm{id}$	4 $E \to E \,/\, E$	7 $E \to (\ E\ )$
2 $E \to \mathrm{num}$	5 $E \to E + E$	
3 $E \to E * E$	6 $E \to E - E$	

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 %% */
```

Input this code into exp.y to follow the example.

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.

[...]

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:

$$\begin{array}{lll} E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \text{id} \\ E \rightarrow E - T & T \rightarrow T / F & F \rightarrow \text{num} \\ E \rightarrow T & T \rightarrow F & F \rightarrow (E) \end{array}$$

```
%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:

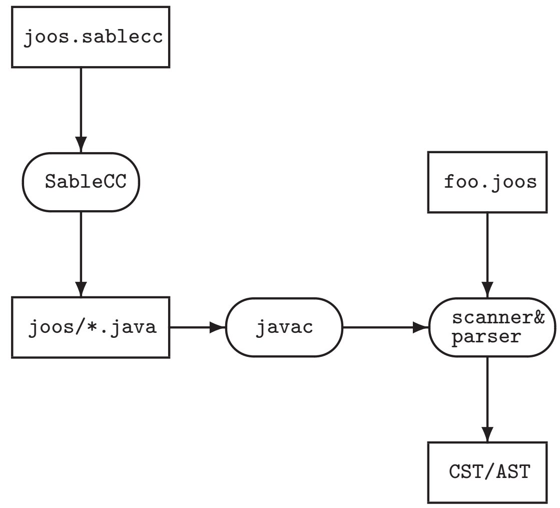
```
exp : tIDENTIFIER { printf ("load %s\n", $1); }
    .
    .
    .
    | '(' exp ')'
    | error { yyerror(); }
;
```

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
```

Error recovery hardly ever works.

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).