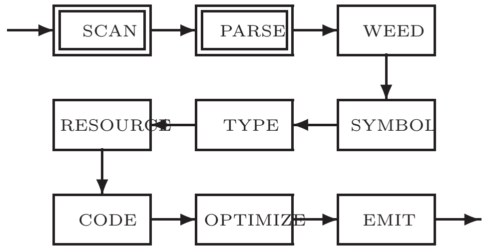
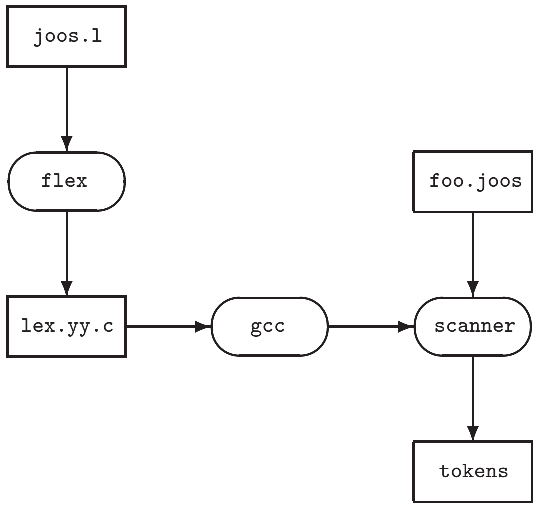
Scanners and parsers



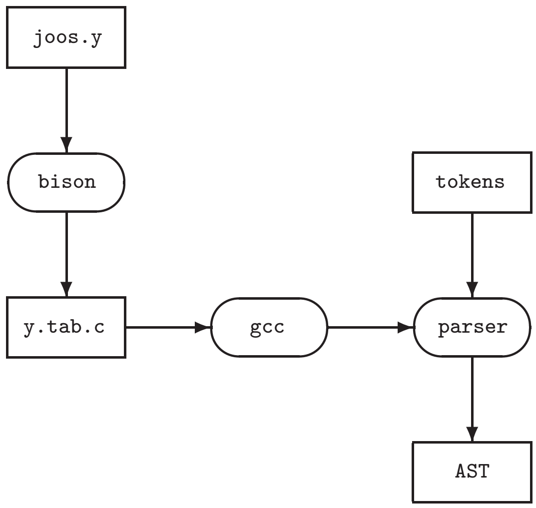
A *scanner* transforms a string of characters into a string of symbols:

- it corresponds to a *deterministic finite-state automaton*;
- plus some C-code to make it work;
- which is generated by `flex`.



A *parser* transforms a string of symbols into a parse tree (according to some grammar):

- it corresponds to a *deterministic push-down automaton*;
- plus some C-code to make it work;
- which is generated by `bison` (`yacc`).



Regular expressions:

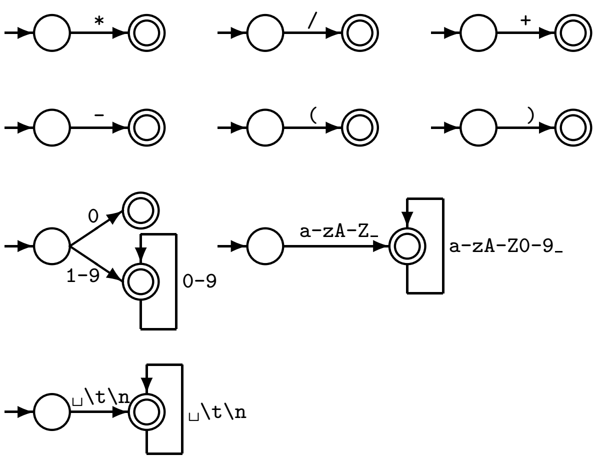
- a, where $a \in \Sigma$
- ε, the empty string
- ϕ, the language does not contain any strings
- $(R_1 \cup R_2)$, $(R_1 \circ R_2)$, or (R_1^*), where R_1 and R_2 are regular expressions

Symbols are typically defined by *regular expressions*:

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

We have used standard UNIX notation.

`flex` accepts a list of regular expressions, and creates a DFA for each:



Each DFA has an associated *action*.

Given DFAs $D_1, \ldots, D_n$, the action 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;
    k := max{|s_i|};
    if k>0 then
        j := min{i : |s_i| = k};
        remove s_j from input;
        perform the j'th action
    else
        move one character from input to output
    end
end
```



```
%{
#include <stdio.h>
%}

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

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


On input:

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

the `flex` scanner will produce the output:

```
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
```

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>
#include <stdio.h>
%}

%%
[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

- $\boldsymbol{V}$, a set of *variables*
- $\boldsymbol{\Sigma}$, a set of *terminals*, $\boldsymbol{V} \cap \boldsymbol{\Sigma} = \phi$
- $\boldsymbol{R}$, a set of *rules*, the LHS is a `variable` and the RHS is a string of `variables` and `terminals`
- $\boldsymbol{S} \in \boldsymbol{V}$, the start variable

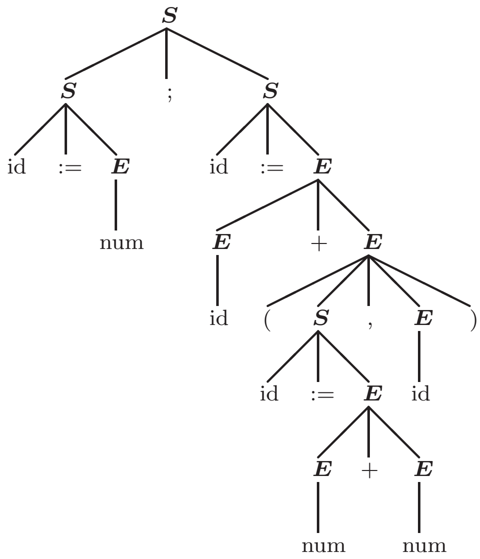
$\boldsymbol{S} \longrightarrow \boldsymbol{S}\ ;\ \boldsymbol{S}$	$\boldsymbol{E} \longrightarrow \text{id}$	$\boldsymbol{L} \longrightarrow \boldsymbol{E}$
$\boldsymbol{S} \longrightarrow \text{id} := \boldsymbol{E}$	$\boldsymbol{E} \longrightarrow \text{num}$	$\boldsymbol{L} \longrightarrow \boldsymbol{L}\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

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

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

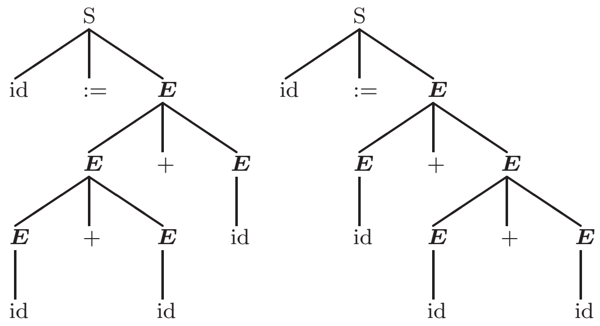
$\boldsymbol{S} \longrightarrow \boldsymbol{S}\ ;\ \boldsymbol{S}$	$\boldsymbol{E} \longrightarrow \text{id}$	$\boldsymbol{L} \longrightarrow \boldsymbol{E}$
$\boldsymbol{S} \longrightarrow \text{id} := \boldsymbol{E}$	$\boldsymbol{E} \longrightarrow \text{num}$	$\boldsymbol{L} \longrightarrow \boldsymbol{L}\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

```
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:

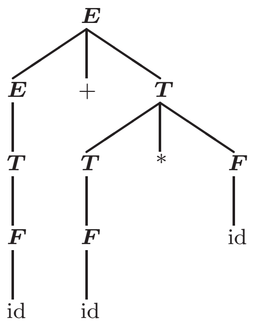
```
id := id - id - id
id := id + id * id
```


An ambiguous grammar:

$\boldsymbol{E} \rightarrow \text{id}$	$\boldsymbol{E} \rightarrow \boldsymbol{E} \,/\, \boldsymbol{E}$	$\boldsymbol{E} \rightarrow (\ \boldsymbol{E}\ )$
$\boldsymbol{E} \rightarrow \text{num}$	$\boldsymbol{E} \rightarrow \boldsymbol{E} + \boldsymbol{E}$	
$\boldsymbol{E} \rightarrow \boldsymbol{E} * \boldsymbol{E}$	$\boldsymbol{E} \rightarrow \boldsymbol{E} - \boldsymbol{E}$	

may be rewritten to become unambiguous:

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



The *shift-reduce* parsing technique.

Extend the grammar with an end-of-file \$:

$\boldsymbol{S'} \rightarrow \boldsymbol{S}\$$

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

Choose between the following actions:

- shift:
  move first input token to top of stack
- reduce:
  replace $\boldsymbol{\alpha}$ on top of stack by $\boldsymbol{X}$
  for some rule $\boldsymbol{X} \rightarrow \boldsymbol{\alpha}$
- accept:
  when $\boldsymbol{S}\$$ is reduced

	`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	`$`	accept

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

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

Start with the initial state 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

	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				

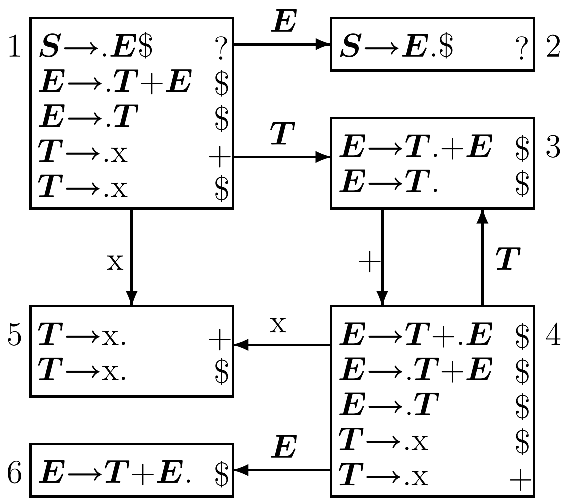
LR(1) is an algorithm that attempts to construct a parsing table:

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

If no conflicts (shift/reduce, reduce/reduce) arise, then we are happy; otherwise, it's just too bad.

Follow construction on the tiny grammar:

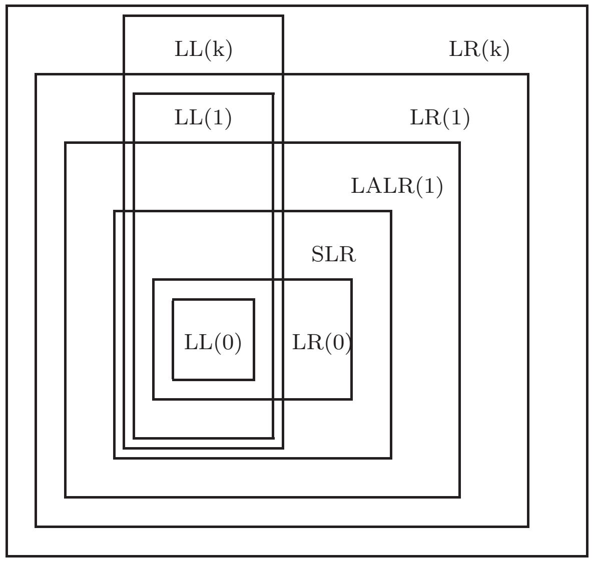
$_0$ $S \to E\$$	$_2$ $E \to T$
$_1$ $E \to T + E$	$_3$ $T \to$ x



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

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

Nobody writes (simple) parsers by hand anymore.

The grammar:

$_1\ E \rightarrow \text{id}$ $\quad _4\ E \rightarrow E \,/\, E$ $\quad _7\ E \rightarrow (\ E\ )$

$_2\ E \rightarrow \text{num}$ $\quad _5\ E \rightarrow E + E$

$_3\ E \rightarrow E * E$ $\quad _6\ E \rightarrow E - E$

is in `bison` expressed as:

```
%token tIDENTIFIER tINTCONST

%start exp

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

%%
```

The grammar is ambiguous:

```
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)
    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:

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

```
%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 '*' '/'

%%
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.
Conflict in state 11 between rule 5 and token '-'
         resolved as reduce.
Conflict in state 11 between rule 5 and token '*'
         resolved as shift.
Conflict in state 11 between rule 5 and token '/'
         resolved as shift.
```


The precedence directives are:

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

An action is chosen based on the precedence of the last symbol on the right-hand side of the rule.

If the precedences are equal, then:

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

These defaults actually often work.

```
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
```



```
%{
#include <stdio.h>
extern char *yytext;
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 ')' {}
;

%%
```



```
%{
#include "y.tab.h"
#include <string.h>
%}

%%
[ \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;
        }
%%
```


The program:

```
void yyparse();

void main()
{ yyparse();
}
```

will on input:

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

produce the output:

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

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

We may ask it to recover from the error:

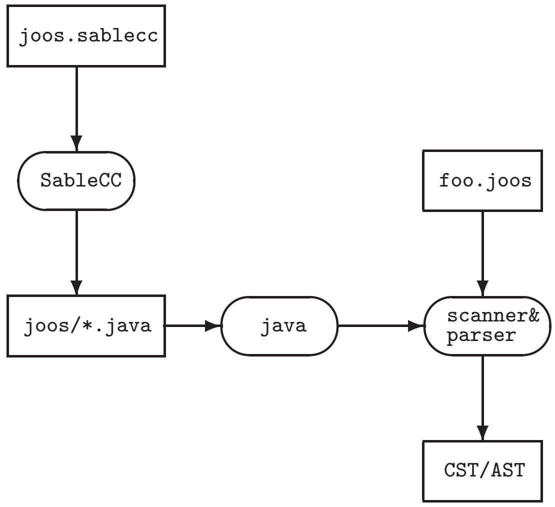
```
exp : tIDENTIFIER
      { printf("load %s\n",$1); }
    .
    .
    .
    | '(' exp ')'
    | error
      { yyerror(); }
```

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) is a *compiler compiler*, it takes a grammatical description of the souce language as input, and generates a lexer and a parser for it.



The SableCC (version 2) grammar of our tiny EXP 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 (Concrete Syntax Trees).

The SableCC (version 3) grammar of our tiny EXP 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