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, ...

Tokens are defined by regular expressions:
- $\emptyset$, the empty set: a language with no strings
- $\varepsilon$, the empty string
- $a$, where $a \in \Sigma$ and $\Sigma$ is our alphabet
- $M | N$, alternation: either $M$ or $N$
- $M \cdot N$, concatenation: $M$ followed by $N$
- $M^*$, zero or more occurrences of $M$
where $M$ and $N$ are both regular expressions. What are $M^*$ and $M^+$?

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: [\w\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):

![Diagram of NFA and DFA conversion]

Each DFA has an associated *action*.

---

### Why the “longest match” principle?

**Example: keywords**

```c
[ \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**

```c
[ \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  ــ DO5I=1.25
   in C: do5i = 1.25;

2. DO 5 I = 1,25  ــ 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;
```

Using flex to create a scanner is really simple:

```
$ 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 %*/
%%
[	\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 ();
}
```

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:

```c
#include <stdlib.h> /* for atoi */
#include <stdio.h> /* for printf */

int lines = 0, chars = 0;

int main () {
    yylex();
    printf ("#lines = %i, #chars = %i\n", lines, chars);
    return 0;
}
```

Remove vowels and increment integers:

```c
#include <stdlib.h> /* for atoi */
#include <stdio.h> /* for printf */

int main () {
    yylex();
    printf ("%i", atoi(yytext) + 1);
    return 0;
}
```

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

- \( V \), a set of **variables** (or **non-terminals**)
- \( \Sigma \), 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 \( \Sigma \)
- \( S \in V \), 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:

\( ( ) , ( () ) , ( ( () ) ) , \ldots \)

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.

![Diagram](joos.y -> bison -> y.tab.c -> gcc -> parser -> AST)

Simple CFG example: Alternatively:

\[
\begin{align*}
A &\rightarrow a B \\
A &\rightarrow \epsilon \\
B &\rightarrow b B \\
B &\rightarrow c
\end{align*}
\]

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:

\[
\begin{align*}
A &\rightarrow a B \\
a &\rightarrow b B \\
a b b &\rightarrow B \\
a b b c
\end{align*}
\]
There are several different grammar formalisms. First, consider BNF (Backus-Naur Form):

\[
\text{stmt} ::= \text{stmt_expr} ";" \mid \\
\text{while_stmt} \mid \\
\text{block} \mid \\
\text{if_stmt} \mid \\
\text{while_stmt} := \text{WHILE} "(" \text{expr} ")" \text{stmt} \\
\text{block} ::= "\{" \text{stmt_list} "}" \\
\text{if_stmt} ::= \text{IF} "(" \text{expr} ")" \text{stmt} \mid \\
\text{IF} "(" \text{expr} ")" \text{stmt} \text{ELSE} \text{stmt}
\]

We have four options for \text{stmt_list}:

1. \text{stmt_list} ::= \text{stmt_list stmt} \mid \epsilon \\
   \to 0 \text{ or more, left-recursive}
2. \text{stmt_list} ::= \text{stmt} \text{stmt_list} \mid \epsilon \\
   \to 0 \text{ or more, right-recursive}
3. \text{stmt_list} ::= \text{stmt_list stmt} \mid \text{stmt} \\
   \to 1 \text{ or more, left-recursive}
4. \text{stmt_list} ::= \text{stmt_list stmt} \mid \text{stmt} \\
   \to 1 \text{ or more, right-recursive}

Second, consider EBNF (Extended BNF):

\[
\begin{array}{|c|c|c|}
\hline
\text{BNF} & \text{derivations} & \text{EBNF} \\
\hline
A \to A a | b & \text{a a} & A \to \{ a \} \\
& \text{b a a} & \text{A} \\
(\text{left-recursive}) & & \text{A} \\
\hline
A \to a A | b & \text{a a A} & A \to \{ a \} b \\
& \text{a a} & \text{A} \\
(\text{right-recursive}) & & \text{A} \\
\hline
\end{array}
\]

where '{' and '}' are like Kleene *'s in regular expressions. Using EBNF repetition, our four choices for \text{stmt_list} become:

1. \text{stmt_list} ::= \{ \text{stmt} \}
2. \text{stmt_list} ::= \{ \text{stmt} \} \text{stmt}
3. \text{stmt_list} ::= \text{stmt} \{ \text{stmt} \}

EBNF also has an optional-construct. For example:

\[
\text{stmt_list} ::= \text{stmt stmt_list} | \text{stmt}
\]

could be written as:

\[
\text{stmt_list} ::= \text{stmt} \{ \text{stmt_list} \}
\]

And similarly:

\[
\text{if_stmt} ::= \text{IF} "(" \text{expr} ")" \text{stmt} \mid \\
\text{IF} "(" \text{expr} ")" \text{stmt} \text{ELSE} \text{stmt}
\]

could be written as:

\[
\text{if_stmt} ::= \\
\text{IF} "(" \text{expr} ")" \text{stmt} \{ \text{ELSE} \text{stmt} \}
\]

where '{' and '}' are like '?' in regular expressions.

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

\[
\text{stmt}
\]

(\text{stmt_expr} \rightarrow \text{;} \text{ while_stmt} \text{ block} \text{ if_stmt})

\[
\text{while_stmt}
\]

(\text{while} \{ \text{expr} \} \text{stmt})

\[
\text{block}
\]

(\{ \text{stmt_list} \})
A grammar is *ambiguous* if a sentence has different parse trees:

\[
S \rightarrow S ; S \\
S \rightarrow \text{id} := E \\
S \rightarrow \text{print } ( L ) \\
S \rightarrow S , E \\
\]

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

\[
S \rightarrow S ; S \\
S \rightarrow \text{id} := E \\
S \rightarrow \text{print } ( L ) \\
S \rightarrow S , E \\
\]

The above is harmless, but consider:

\[
\text{id} := \text{id} - \text{id} - \text{id} \\
\text{id} := \text{id} + \text{id} * \text{id} \\
\]

Clearly, we need to consider associativity and precedence when designing grammars.
An ambiguous grammar:

\[
E \rightarrow \text{id} \\
E \rightarrow E \, / \, E \\
E \rightarrow E \, + \, E \\
E \rightarrow E \, - \, E \\
E \rightarrow \text{num} \\
E \rightarrow E \, * \, E \\
E \rightarrow E \, - \, E
\]

may be rewritten to become unambiguous:

\[
E \rightarrow E \, + \, T \, T \\
T \rightarrow T \, * \, F \\
F \rightarrow \text{id} \\
E \rightarrow E \, - \, T \, T \\
T \rightarrow T \, / \, F \\
F \rightarrow \text{num} \\
E \rightarrow T \, T \\
T \rightarrow F \\
F \rightarrow ( \, E 
\]

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

2) Choose between the following actions:
- **shift**: move first input token to top of stack
- **reduce**: replace \(\alpha\) 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 := num ; b := c+(d := 5+6,d)$ E → num
id := E ; b := c+(d := 5+6,d)$ S → 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 := id + (d := 5+6,d)$ E → id
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 := num + 6,d)$ E → num
S ; id := E + (id := E + 6,d)$ shift
S ; id := E + (id := E + 6,d)$ shift
S ; id := E + (id := E + E + 6,d)$ E → E + E
S ; id := E + (id := E + E + 6,d)$ E → E + E
S ; id := E + (id := E + 6,d)$ shift
S ; id := E + (id := E + 6,d)$ shift
S ; id := E + (id := E + 6,d)$ E → (S,E)
S ; id := E + E $ S → id := E
S ; id := E + E $ S → E + E
S ; id := E $ S → S ; S
S $ shift
S$ $ S' → S$
S$ accept

0 $ S' → S$
5 $ E → num$
1 $ S → S ; S$
6 $ E → E + E$
2 $ S → id := E$
7 $ E → (S , E )$
3 $ S → print (L)$
8 $ L → E$
4 $ E → id$
9 $ L → 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 $X → α$; pop $|α|$ times; lookup (stack top, X) in table
- goto(n): push state n
- accept: report success
- error: report failure

<table>
<thead>
<tr>
<th>DFA state</th>
<th>terminals</th>
<th>non-terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>num</td>
<td>print</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
<td>s7</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td>s7</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>s6</td>
</tr>
<tr>
<td>5</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>6</td>
<td>s20</td>
<td>s10</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>s9</td>
</tr>
<tr>
<td>8</td>
<td>s4</td>
<td>s7</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>11</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>12</td>
<td>s3</td>
<td>s18</td>
</tr>
<tr>
<td>13</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>14</td>
<td>s19</td>
<td>s13</td>
</tr>
<tr>
<td>15</td>
<td>r8</td>
<td>r8</td>
</tr>
<tr>
<td>16</td>
<td>s20</td>
<td>s10</td>
</tr>
<tr>
<td>17</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>18</td>
<td>s20</td>
<td>s10</td>
</tr>
<tr>
<td>19</td>
<td>s20</td>
<td>s10</td>
</tr>
<tr>
<td>20</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>r7</td>
<td>r7</td>
</tr>
<tr>
<td>23</td>
<td>s16</td>
<td>s9</td>
</tr>
</tbody>
</table>

Error transitions omitted.
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 \( A \rightarrow \alpha \cdot \beta \gamma \cdot x \) consists of

1. A grammar production, \( A \rightarrow \alpha \beta \gamma \)
2. The RHS position, represented by \( \cdot \)
3. A lookahead symbol, \( x \)

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

The sequence \( \alpha \) is on top of the stack, and the head of the input is derivable from \( \beta \gamma x \). There are two cases for \( \beta \), 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 \( \beta \) is non-terminal
2. shift: when \( \beta \) is terminal
3. reduce: when \( \beta \) is empty (the next state is the number of the production used)
4. accept: when we have \( A \rightarrow B \cdot $ \)

Follow construction on the tiny grammar:

\[
\begin{align*}
0 & \quad S \rightarrow E $ \\
2 & \quad E \rightarrow T \\
1 & \quad E \rightarrow T + E \\
3 & \quad T \rightarrow x \\
\end{align*}
\]

Constructing the LR(1) NFA:

- start with state \( S \rightarrow . E $ ? \)
- state \( A \rightarrow \alpha \cdot B \beta 1 \) has:
  - \( \epsilon \)-successor \( B \rightarrow . \gamma \cdot x \), if:
    * \( \exists \) rule \( B \rightarrow \gamma \), and
    * \( x \in \text{lookahead}(\beta) \)
  - \( B \)-successor \( A \rightarrow \alpha B \cdot \beta 1 \)
- state \( A \rightarrow \alpha \cdot x \beta 1 \) has:
  - \( x \)-successor \( A \rightarrow \alpha x \cdot \beta 1 \)

Constructing the LR(1) DFA:

Standard power-set construction, “inlining” \( \epsilon \)-transitions.
Conflicts

\[
\begin{align*}
A \rightarrow B & \quad x \\
A \rightarrow C & \quad y
\end{align*}
\]
no conflict (lookahead decides)

\[
\begin{align*}
A \rightarrow B & \quad x \\
A \rightarrow C & \quad x
\end{align*}
\]
shift/reduce conflict

\[
\begin{align*}
A \rightarrow x & \quad y \\
A \rightarrow C & \quad x
\end{align*}
\]
shift/reduce conflict

\[
\begin{align*}
A \rightarrow B & \quad x \\
A \rightarrow C & \quad x
\end{align*}
\]
reduce/reduce conflict

\[
\begin{align*}
A \rightarrow B & \quad x \\
A \rightarrow C & \quad x
\end{align*}
\]
shift/shift conflict?

⇒ by construction of the DFA
we have \( s_i = s_j \)

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.

LR(1) tables may become very large.
Parser generators use LALR(1), which merges states that are identical except for lookaheads.

The grammar:

\[
\begin{align*}
1 & \quad E \rightarrow id \\
4 & \quad E \rightarrow E / E \\
7 & \quad E \rightarrow ( E ) \\
2 & \quad E \rightarrow num \\
5 & \quad E \rightarrow E + E \\
3 & \quad E \rightarrow E * E \\
6 & \quad E \rightarrow E - E
\end{align*}
\]

is expressed in bison as:

\[
\text{%}
/* 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 4)
exp -> exp . '+' exp (rule 5)
exp -> exp . '-' exp (rule 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:

\[ E \rightarrow E + T \] \[ T \rightarrow T * F \] \[ F \rightarrow id \]
\[ E \rightarrow E - T \] \[ T \rightarrow T / F \] \[ F \rightarrow num \]

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 */
%%
[ 	
]+ /* ignore */;
"*" 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:

```c
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.

```
joos.sablecc
```

```
SableCC
```

```
joos/*.java javac scanner& parser
```

```
CST/AST
```

**The SableCC 2 grammar for our Tiny language:**

**Package tiny;**

**Helpers**

```c
tab = 9;

cr = 13;

lf = 10;

digit = ['0'..'9'];

lowercase = ['a'..'z'];

uppercase = ['A'..'Z'];

letter = lowercase | uppercase;

idletter = letter | '_';

idchar = letter | '_' | digit;
```

**Tokens**

```c
eol = cr | lf | cr lf;

blank = ' ' | cr;

star = '*';

slash = '/';

plus = '+';

minus = '-';

l_par = '(';

r_par = ')';

number = '0'| [digit-'0'] digit*;

id = idletter idchar*;
```

**Ignored Tokens**

```c
blank, eol;
```

**Productions**

```c
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) |
   (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).