A scanner transforms a string of characters into a string of tokens:
- it corresponds to a deterministic finite-state automaton;
- plus some C-code to make it work;
- which is generated by flex (or lex).

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 C-code to make it work;
- which is generated by bison (or yacc).

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 UNIX notation:
- simple operators: "\*", "/", "+", "-"
- parentheses: "("", ")"
- integer constants: $0 | (1-9)[0-9]*$
- identifiers: [a-zA-Z] [a-zA-Z][0-9]*
- white space: [\t\n]+
flex accepts a list of regular expressions, converts each regex internally to an NFA, 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:

```plaintext
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
```

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

Using flex to create a scanner is really simple:

```bash
$ 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);
/*
[ \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-z|A-Z)(a-z|A-Z-9_)* printf ("identifier: %s\n", yytext);
%%
/* user code comes after the second %% */
main () {
    yylex ();
}
```

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

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

our print_tokens scanner outputs:

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

  lines++; chars++;
  .  chars++;
%
```

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

Remove vowels and increment integers:

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

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

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

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

- \(V\), a set of variables
- \(\Sigma\), 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:

\[ E \rightarrow ( E ) | \epsilon \]

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: Alternatively:

\[ A \rightarrow a B \]
\[ A \rightarrow \epsilon \]
\[ B \rightarrow b B \]
\[ B \rightarrow c \]

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:

\[ A \]
\[ a B \]
\[ a b B \]
\[ a b b B \]
\[ a b b c \]
There are several different grammar formalisms. First, consider BNF (Backus-Naur Form):

```plaintext
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):

<table>
<thead>
<tr>
<th>BNF</th>
<th>derivations</th>
<th>EBNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow A \ a \ b$</td>
<td>$\epsilon$</td>
<td>$A \rightarrow b { a }$</td>
</tr>
</tbody>
</table>

(left-recursive)

| $A \rightarrow a \ A \ b$ | $\epsilon$ | $A \rightarrow \{ a \} b$ |

(right-recursive)

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 \}`
3. `stmt_list ::= \{ stmt \} stmt`
4. `stmt_list ::= stmt \{ stmt \}`

EBNF also has optional constructs. For example:

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

could be written as:

```plaintext
stmt_list ::= stmt \ [ stmt_list \ ]
```

And similarly:

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

could be written as:

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

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

Third, consider “railroad” syntax diagrams:

(thanks rail.sty!)

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

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

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:

\[
E \rightarrow \text{id} \\
E \rightarrow E / E \\
E \rightarrow E + E \\
E \rightarrow E \times E \\
E \rightarrow E - E
\]

may be rewritten to become unambiguous:

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

There are fundamentally two kinds of parser:

1) **Top-down** 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(1) grammar:

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

Algorithm: look at beginning of input 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 $:

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

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 reduced
We can use a DF A 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

Example: show complete stack evolution for a := 7$

```
a:=7; b:=c+(d:=5+6,d)$  shift  
S'  →  S  E  L
1 S → S ; S  E → E + E
2 S → id := E  E → ( S , E )
3 S → print ( L )  S → S; S
4 E → id  L → L , E
```

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

- **Left-to-right parse:**
- **Rightmost-derivation:** and
- **↓ symbol lookahead**

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

An LR(1) item (A → α . β, x) consists of

1. A grammar production, A → αβ
2. The RHS position, represented by ' .'  
3. A lookahead symbol, x

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

The sequence α is on top of the stack, and the head of the input is derivable from βx. There are two cases for β, 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:

$S \rightarrow E$
$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow x$

Don’t worry about constructing the LR states, but do understand how to build the parse table from the state diagram. (It’s not hard.)

See Appel 3.3 for details.

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. \( E \rightarrow \text{id} \)
2. \( E \rightarrow \text{num} \)
3. \( E \rightarrow E \ast E \)
4. \( E \rightarrow E / E \)
5. \( E \rightarrow E + E \)
6. \( E \rightarrow E - E \)

is expressed in bison as:

```c
/* 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 ')' ;

/* 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.

[...]

```
%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.

Rewrite the grammar to force reductions:

```c
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 )
```

```c
%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:

```c
%left '@' '-' /* left-associative, lower precedence */
%left '+' '-' /* left-associative, higher precedence */

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

```
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.

```c
$ 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 %d\n", $1); }
  | exp '*' exp { printf ("mul\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 '(';
')' return ')';
0|([1-9]\d*) { /* ignore */;
yylval.intconst = atoi (yytext);
return tINTCONST;
}
[a-zA-Z_][a-zA-Z0-9_] { /* ignore */;
yylval.stringconst = malloc (strlen (yytext) + 1);
printf (yylval.stringconst, "%s", yytext);
return tIDENTIFIER;
}
```
Using `flex/bison` to create a parser is simple:

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

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

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

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

- load a
- load b
- push 17
- minus
- 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("\n\n\nload %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.

![SableCC Diagram](image)

The SableCC 2 grammar for our Tiny language:

```plaintext
Package tiny;

Helpers
  tab = 9;
  cr = 13;
  lf = 10;
  digit = ['0'..'9'];
  lowercase = ['a'..'z'];
  uppercase = ['A'..'Z'];
  letter = lowercase | uppercase;
  idletter = letter | '_' | digit;
  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

\[
\begin{align*}
\text{exp} & = \{ \text{plus} \} \text{ exp plus factor |} \\
& \quad \{ \text{minus} \} \text{ exp minus factor |} \\
& \quad \{ \text{factor} \} \text{ factor;}
\end{align*}
\]

\[
\begin{align*}
\text{factor} & = \{ \text{mult} \} \text{ factor star term |} \\
& \quad \{ \text{divd} \} \text{ factor slash term |} \\
& \quad \{ \text{term} \} \text{ term;}
\end{align*}
\]

\[
\begin{align*}
\text{term} & = \{ \text{paren} \} \text{ l_par exp r_par |} \\
& \quad \{ \text{id} \} \text{ id |} \\
& \quad \{ \text{number} \} \text{ number;}
\end{align*}
\]

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

The SableCC 3 grammar for our Tiny language:

Productions

\[
\begin{align*}
\text{cst_exp} \rightarrow \text{exp} & = \{\text{cst_plus}\} \text{ cst_exp plus factor} \\
& \quad \{\text{cst_minus}\} \text{ cst_exp minus factor} \\
& \quad \{\text{factor}\} \text{ factor \rightarrow cst_exp.exp;}
\end{align*}
\]

\[
\begin{align*}
\text{factor} \rightarrow \text{exp} & = \{\text{cst_mult}\} \text{ factor star term} \\
& \quad \{\text{cst_divd}\} \text{ factor slash term} \\
& \quad \{\text{term}\} \text{ term \rightarrow term.exp;}
\end{align*}
\]

\[
\begin{align*}
\text{term} \rightarrow \text{exp} & = \{\text{paren}\} \text{ l_par cst_exp r_par \rightarrow cst_exp.exp} \\
& \quad \{\text{id}\} \text{ id \rightarrow New exp.id(id)} \\
& \quad \{\text{number}\} \text{ number \rightarrow New exp.number(number);} \\
\end{align*}
\]

Abstract Syntax Tree

\[
\begin{align*}
\text{exp} & = \{\text{plus}\} \{l\}:\text{exp} \{r\}:\text{exp} | \\
& \quad \{\text{minus}\} \{l\}:\text{exp} \{r\}:\text{exp} | \\
& \quad \{\text{mult}\} \{l\}:\text{exp} \{r\}:\text{exp} | \\
& \quad \{\text{divd}\} \{l\}:\text{exp} \{r\}:\text{exp} | \\
& \quad \{\text{id}\} \text{ id |} \\
& \quad \{\text{number}\} \text{ number;}
\end{align*}
\]

Version 3 generates abstract syntax trees (ASTs).