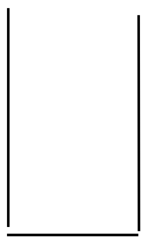
Bottom Up Parsing

Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Parsing, Section 3.

Bottom Up Parsing

An Example:

[1]	$E \rightarrow E + T$
[2]	$E \rightarrow T$
[3]	$T \rightarrow T * F$
[4]	$T \rightarrow F$
[5]	$F \rightarrow (E)$
[6]	$F \rightarrow id$



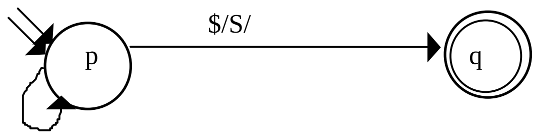
id + id * id $

Creating a Bottom Up PDA

There are two basic actions:

1. Shift an input symbol onto the stack
2. Reduce a string of stack symbols to a nonterminal

M will be:



So, to construct M from a grammar G, we need the following transitions:

(1) The shift transitions:

$((p, a, \varepsilon), (p, a))$, for each $a \in \Sigma$

(2) The reduce transitions:

$((p, \varepsilon, \alpha^R), (p, A))$, for each rule $A \rightarrow \alpha$ in G.

(3) The finish up transition (accept):

$((p, \$, S), (q, \varepsilon))$

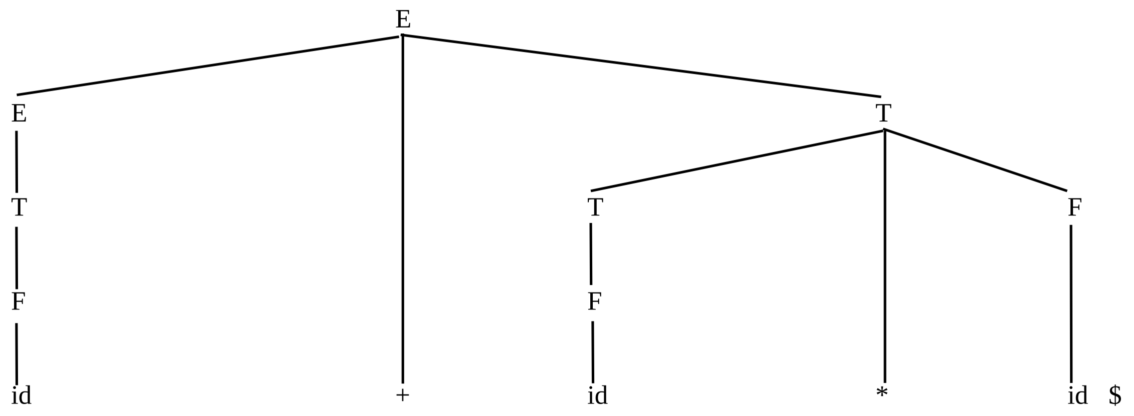
(This is the “bottom-up” CFG to PDA conversion algorithm.)

M for Expressions

0	(p, a, ε), (p, a) for each a ∈ Σ
1	(p, ε, T + E), (p, E)
2	(p, ε, T), (p, E)
3	(p, ε, F * T), (p, T)
4	(p, ε, F), (p, T)
5	(p, ε, ")"E"("), (p, F)
6	(p, ε, id), (p, F)
7	(p, $, E), (q, ε)

trans (action)	***state***	***unread input***	***stack***
	p	id + id * id$	ε
0 (shift)	p	+ id * id$	id
6 (reduce F → id)	p	+ id * id$	F
4 (reduce T → F)	p	+ id * id$	T
2 (reduce E → T)	p	+ id * id$	E
0 (shift)	p	id * id$	+E
0 (shift)	p	* id$	id+E
6 (reduce F → id)	p	* id$	F+E
4 (reduce T → F)	p	* id$	T+E (could also reduce)
0 (shift)	p	id$	*T+E
0 (shift)	p	$	id*T+E
6 (reduce F → id)	p	$	F*T+E (could also reduce T → F)
3 (reduce T → T * F)	p	$	T+E
1 (reduce E → E + T)	p	$	E
7 (accept)	q	$	ε

The Parse Tree



Producing the Rightmost Derivation

We can reconstruct the derivation that we found by reading the results of the parse bottom to top, producing:

E ⇒
E+ T ⇒
E+ T* F⇒
E+ T* id⇒
E+ F* id⇒
E+ id* id⇒
T+ id*id⇒
F+ id*id⇒
id+ id*id

This is exactly the rightmost derivation of the input string.

Possible Solutions to the Nondeterminism Problem

1) **Modify the language**
   - Add a terminator $

2) **Change the parsing algorithm**
   - *Add one character look ahead*
   - *Use a parsing table*
   - *Tailor parsing table entries by hand*
   - **Switch to a bottom-up parser**

3) **Modify the grammar**
   - *Left factor*
   - *Get rid of left recursion*

Solving the Shift vs. Reduce Ambiguity With a Precedence Relation

Let's return to the problem of deciding when to shift and when to reduce (as in our example).

We chose, correctly, to shift * onto the stack, instead of reducing T+E to E.

This corresponds to knowing that "+" has low precedence, so if there are any other operations, we need to do them first.

Solution:
1. Add a one character lookahead capability.
2. Define the precedence relation

$$P \subseteq \left(\underset{\substack{\text{top}\\ \text{stack}\\ \text{symbol}}}{V} \times \underset{\substack{\text{next}\\ \text{input}\\ \text{symbol}}}{\{\Sigma \cup \$\}} \right)$$

If (a,b) is in P, we reduce (without consuming the input) . Otherwise we shift (consuming the input).

How Does It Work?

We're reconstructing rightmost derivations backwards. So suppose a rightmost derivation contains

$$\begin{array}{l} \beta\gamma abx \\ \Uparrow \quad \longleftarrow \quad \text{corresponding to a rule } A \rightarrow \gamma a \text{ and not some rule } X \rightarrow ab \\ \beta Abx \\ \Uparrow^{*} \\ S \end{array}$$

We want to undo rule A. So if the top of the stack is

$$\begin{array}{|c|} a \\ \gamma \\ \hline \end{array}$$

and the next input character is b, we reduce now, before we put the b on the stack.

To make this happen, we put (a, b) in P. That means we'll try to reduce if a is on top of the stack and b is the next character. We will actually succeed if the next part of the stack is γ.

Example

$$\begin{tikzcd}[row sep=small]
T*F \\
T \arrow[u, Rightarrow] & & \arrow[ll, "\text{corresponding to a rule } T\to T*F"'] \\
E \arrow[u, Rightarrow, "*"']
\end{tikzcd}$$

Input: $\underline{\text{id * id}}$ * id

We want to undo rule T. So if the top of the stack is

$$\left|\begin{array}{c} F \\ * \\ T \\ \hline \end{array}\right.$$

and the next input character is anything legal, we reduce.

The precedence relation for expressions:

V\Σ	(	)	id	+	*	$
(						
)		●		●	●	●
id		●		●	●	●
+						
*						
E						
T		●		●		●
F		●		●	●	●

A Different Example

$$\begin{tikzcd}[row sep=small]
E+T & & \arrow[ll, "\text{corresponding to a rule } E\to E+T"'] \\
E \arrow[u, Rightarrow, "*"']
\end{tikzcd}$$

We want to undo rule E if the input is $\underline{\text{E + T}}$ $

or $\underline{\text{E + T}}$ + id

but not $\underline{\text{E + T}}$ * id

The top of the stack is

$$\left|\begin{array}{c} T \\ + \\ E \\ \hline \end{array}\right.$$

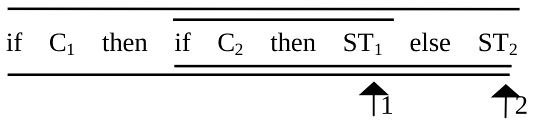
The precedence relation for expressions:

V\Σ	(	)	id	+	*	$
(						
)		●		●	●	●
id		●		●	●	●
+						
*						
E						
T		●		●		●
F		●		●	●	●

What About If Then Else?

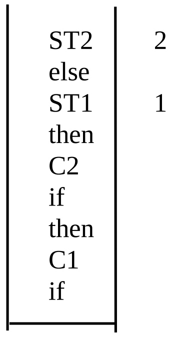
ST → if C then ST else ST
ST → if C then ST

What if the input is



Which bracketing (rule) should we choose?

We don't put (ST, else) in the precedence relation, so we will not reduce at 1. At 2, we reduce:



Resolving Reduce vs. Reduce Ambiguities

0	(p, a, ε), (p, a) for each a ∈ Σ
1	(p, ε, T + E), (p, E)
2	(p, ε, T), (p, E)
3	(p, ε, F * T), (p, T)
4	(p, ε, F), (p, T)
5	(p, ε, ")" E "("), (p, F)
6	(p, ε, id), (p, F)
7	(p, $, E), (q, ε)

trans (action)	***state***	***unread input***	***stack***
	p	id + id * id$	ε
0 (shift)	p	+ id * id$	id
6 (reduce F → id)	p	+ id * id$	F
4 (reduce T → F)	p	+ id * id$	T
2 (reduce E → T)	p	+ id * id$	E
0 (shift)	p	id * id$	+E
0 (shift)	p	* id$	id+E
6 (reduce F → id)	p	* id$	F+E
4 (reduce T → F)	p	* id$	T+E (could also reduce)
0 (shift)	p	id$	*T+E
0 (shift)	p	$	id*T+E
6 (reduce F → id)	p	$	F*T+E (**could also reduce T → F**)
3 (reduce T → T * F)	p	$	T+E
1 (reduce E → E + T)	p	$	E
7 (accept)	q	$	ε

The Longest Prefix Heuristic

A simple to implement heuristic rule, when faced with competing reductions, is:

Choose the longest possible stack string to reduce.

Example:

Suppose the stack has $$\overset{\overset{T}{\Uparrow}}{\overline{\underset{\underset{T}{\Downarrow}}{F} \; * \; T}} \quad + \quad E$$

We call grammars that become unambiguous with the addition of a precedence relation and the longest string reduction heuristic **weak precedence grammars**.

Possible Solutions to the Nondeterminism Problem in a Bottom Up Parser

1) **Modify the language**
   - Add a terminator $

2) **Change the parsing algorithm**
   - Add one character lookahead
   - Use a precedence table
   - Add the longest first heuristic for reduction
   - **Use an LR parser**

3) **Modify the grammar**

LR Parsers

LR parsers scan each input **L**eft to right and build a **R**ightmost derivation. They operate bottom up and deterministically using a parsing table derived from a grammar for the language to be recognized.

A grammar that can be parsed by an LR parser examining up to k input symbols on each move is an **LR(k)** grammar. Practical LR parsers set k to 1.

An LALR (or Look Ahead LR) parser is a specific kind of LR parser that has two desirable properties:

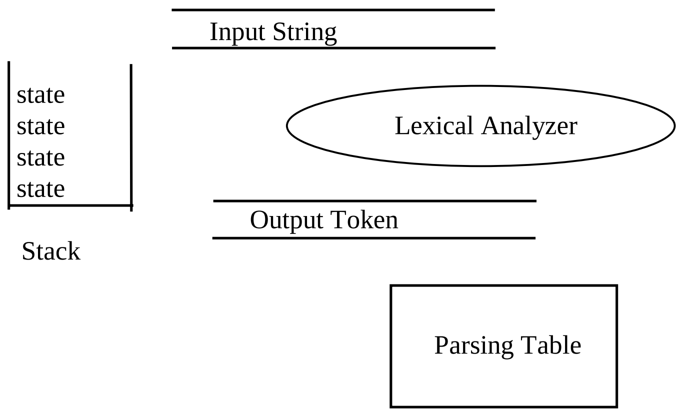
- The parsing table is not huge.
- Most useful languages can be parsed.

Another big reason to use an LALR parser:

There are automatic tools that will construct the required parsing table from a grammar and some optional additional information.

We will be using such a tool: **yacc**

How an LR Parser Works



In simple cases, think of the "states" on the stack as corresponding to either terminal or nonterminal characters.

In more complicated cases, the states contain more information: they encode both the top stack symbol and some facts about lower objects in the stack. This information is used to determine which action to take in situations that would otherwise be ambiguous.

The Actions the Parser Can Take

At each step of its operation, an LR parser does the following two things:

1) Based on its current state, it decides whether it needs a lookahead token. If it does, it gets one.
2) Based on its current state and the lookahead token if there is one, it chooses one of four possible actions:
   - Shift the lookahead token onto the stack and clear the lookahead token.
   - Reduce the top elements of the stack according to some rule of the grammar.
   - Detect the end of the input and accept the input string.
   - Detect an error in the input.

A Simple Example

0: S → rhyme $end ;
1: rhyme → sound place ;
2: sound → DING DONG ;
3: place → DELL

```
state 0  (empty)
        $accept : _rhyme $end                       ⇐ the rule this came from
        DING  shift  3                                state 3
      .  error                                        current position of input
        rhyme  goto 1                                 if none of the others match
        sound  goto 2                                 push state 2
state 1  (rhyme)
        $accept : rhyme_$end
        $end  accept                                  if we see EOF, accept
        .  error
state 2  (sound)
        rhyme : sound_place
        DELL  shift 5
        .  error                                    by rule 1
        place   goto 4
state 3  (DING)
        sound : DING_DONG                        state 5  (DELL)
        DONG  shift 6                                    place : DELL_  (3)
        .  error                                         .   reduce  3
state 4  (place)                                 state 6  (DONG)
        rhyme : sound place_ (1)                     sound : DING DONG_ (2)
        .   reduce  1                                    .   reduce  2
```

When the States Are More than Just Stack Symbols

[1] <stmt> → procname (<paramlist>)
[2] <stmt> → <exp> := <exp>
[3] <paramlist> → <paramlist>, <param> | <param>
[4] <param> → id
[5] <exp> → arrayname (<subscriptlist>)
[6] <subscriptlist> → <subscriptlist>, <sub> | <sub>
[7] <sub> → id

Example:

```
  procname ( id)                                   | id       |
               ↑                                   | (        |
                                                   | procname |
                                                   |__________|
```

Should we reduce id by rule 4 or rule 7?

```
| procid   |
| proc(    |
| procname |
|__________|
```

The parsing table can get complicated as we incorporate more stack history into the states.

The Language Interpretation Problem:

$$\begin{tikzcd}
\text{Input: } -(17 * 83.56) + 72 / 12 \arrow[d] \\
\text{Output: } -1414.52
\end{tikzcd}$$

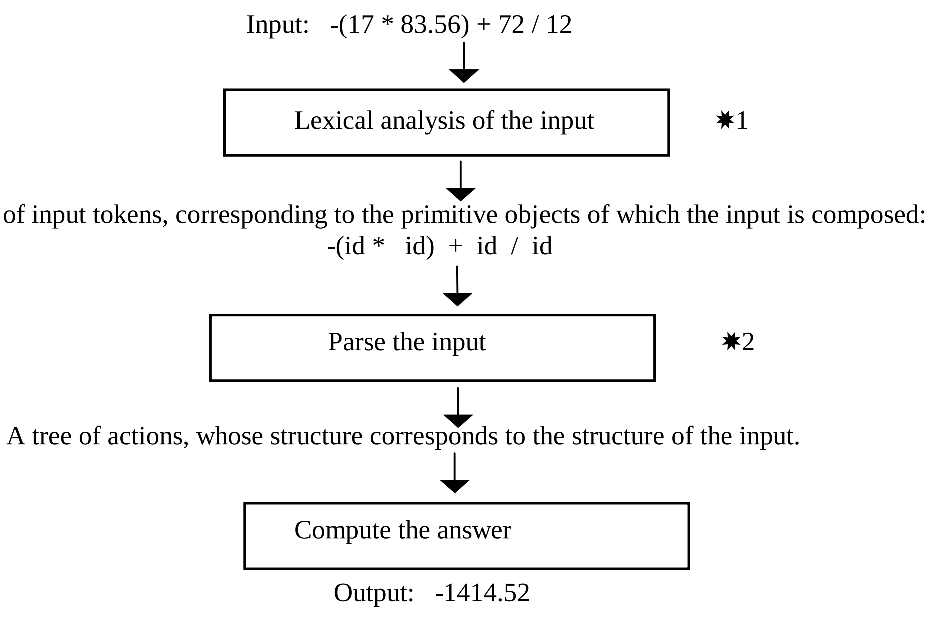
The Language Interpretation Problem:

$$\begin{tikzcd}
\text{Input: } -(17 * 83.56) + 72 / 12 \arrow[d] \\
\boxed{\text{Compute the answer}} \arrow[d] \\
\text{Output: } -1414.52
\end{tikzcd}$$

The Language Interpretation Problem:

$$\begin{tikzcd}
\text{Input: } -(17 * 83.56) + 72 / 12 \arrow[d] \\
\boxed{\text{Parse the input}} \arrow[d] & ✷2 \\
\text{A tree of actions, whose structure corresponds to the structure of the input.} \arrow[d] \\
\boxed{\text{Compute the answer}} \arrow[d] \\
\text{Output: } -1414.52
\end{tikzcd}$$

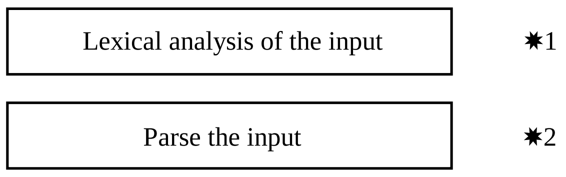
The Language Interpretation Problem:



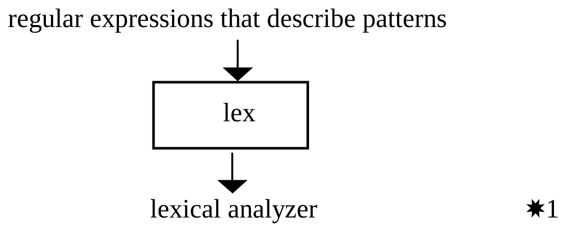
A string of input tokens, corresponding to the primitive objects of which the input is composed:

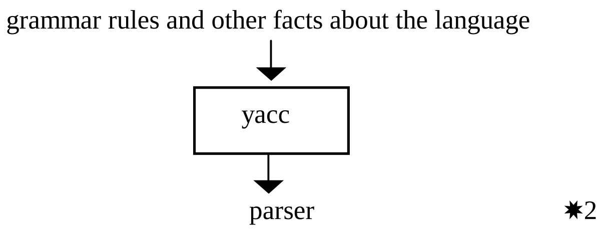
A tree of actions, whose structure corresponds to the structure of the input.

yacc and lex



Where do the procedures to do these things come from?





lex

The input to lex:

```
definitions
%%
rules
%%
user routines
```

All strings that are not matched by any rule are simply copied to the output.

Rules:

```
[ \t]+;                                           get rid of blanks and tabs

[A-Za-z][A-Za-z0-9]*    return(ID);               find identifiers

[0-9]+          {       sscanf(yytext, "%d",  &yylval);
                        return (INTEGER);   }     return INTEGER and put the value in yylval
```

How Does lex Deal with Ambiguity in Rules?

lex invokes two disambiguating rules:

1. The longest match is prefered.
2. Among rules that matched the same number of characters, the rule given first is preferred.

Example:

```
        integer     action 1
        [a-z]+      action 2
```

input:	<u>integers</u>	take action 2
	<u>integer</u>	take action 1

yacc
(**Y**et **A**nother **C**ompiler **C**ompiler)

The input to yacc:

```
declarations
%%
rules
%%
#include "lex.yy.c"
any other programs
```

This structure means that lex.yy.c will be compiled as part of y.tab.c, so it will have access to the same token names.

Declarations:

```
%token name1  name2  …
```

Rules:

```
V       : a   b   c
V       : a   b   c                 {action}
V       : a   b   c                 {$$ = $2}        returns the value of b
```

Example

Input to yacc:

```
        %token  DING  DONG  DELL
        %%
        rhyme   :     sound  place  ;
        sound   :     DING  DONG  ;
        place   :     DELL
        %%
        #include "lex.yy.c"
```

```
state 0  (empty)
        $accept : _rhyme $end
        DING  shift 3
        .  error
        rhyme  goto 1
        sound  goto 2
state 1  (rhyme)
        $accept : rhyme_$end
        $end  accept
        .  error
state 2  (sound)
        rhyme : sound_place
        DELL  shift 5
        .  error
        place   goto 4
```

```
state 3  (DING)
        sound : DING_DONG
        DONG  shift 6
        .  error
state 4  (place)
        rhyme : sound place_ (1)
        .   reduce  1
state 5  (DELL)
        place : DELL_  (3)
        .   reduce  3
state 6  (DONG)
         sound : DING DONG_ (2)
        .   reduce  2
```

How Does yacc Deal with Ambiguity in Grammars?

The parser table that yacc creates represents some decision about what to do if there is ambiguity in the input grammar rules. How does yacc make those decisions? By default, yacc invokes two disambiguating rules:

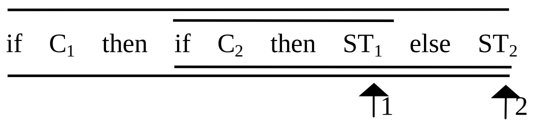
1. In the case of a shift/reduce conflict, shift.
2. In the case of a reduce/reduce conflict, reduce by the earlier grammar rule.

yacc tells you when it has had to invoke these rules.

Shift/Reduce Conflicts - If Then Else

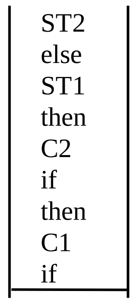
$ST \rightarrow$ if C then ST else ST
$ST \rightarrow$ if C then ST

What if the input is



Which bracketing (rule) should we choose?

yacc will choose to shift rather than reduce.

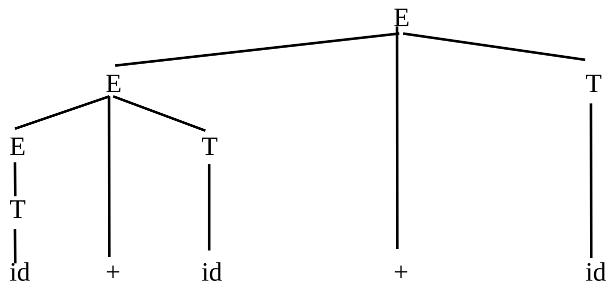


Shift/Reduce Conflicts - Left Associativity

We know that we can force left associativity by writing it into our grammars.

Example:

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow id$

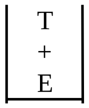


What does the shift rather than reduce heuristic if we instead write:

$E \rightarrow E + E$ id + id + id
$E \rightarrow id$

Shift/Reduce Conflicts - Operator Precedence

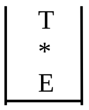
Recall the problem: input: $\underline{\text{id + id}}$ * id



Should we reduce or shift on * ?

The "always shift" rule solves this problem.

But what about: $\underline{\text{id * id}}$ + id



Should we reduce or shift on + ?

This time, if we shift, we'll fail.

One solution was the precedence table, derived from an unambiguous grammar, which can be encoded into the parsing table of an LR parser, since it tells us what to do for each top-of-stack, input character combination.

Operator Precedence

We know that we can write an unambiguous grammar for arithmetic expressions that gets the precedence right. But it turns out that we can build a faster parser if we instead write:

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

And, in addition, we specify operator precedence. In yacc, we specify associativity (since we might not always want left) and precedence using statements in the declaration section of our grammar:

```
%left '+'  '-'
%left '*'  '/'
```

Operators on the first line have lower precedence than operators on the second line, and so forth.

Reduce/Reduce Conflicts

Recall:

2. In the case of a reduce/reduce conflict, reduce by the earlier grammar rule.

This can easily be used to simulate the longest prefix heuristic, "Choose the longest possible stack string to reduce."

[1]	$E \rightarrow E + T$
[2]	$E \rightarrow T$
[3]	$T \rightarrow T * F$
[4]	$T \rightarrow F$
[5]	$F \rightarrow (E)$
[6]	$F \rightarrow id$

Generating an Executable System

Step 1: Create the input to lex and the input to yacc.

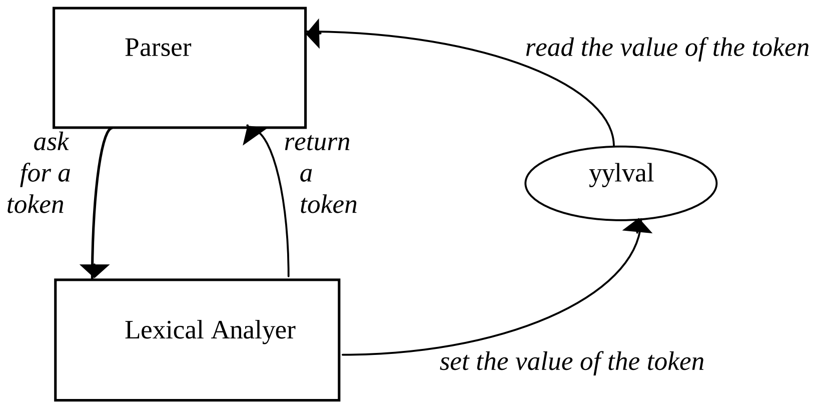
Step 2:

```
$  lex ourlex.l                      creates lex.yy.c
$  yacc ouryacc.y                    creates y.tab.c
$  cc -o ourprog y.tab.c -ly -ll     actually compiles y.tab.c and lex.yy.c, which is included.
                                            -ly links the yacc library, which includes main and yyerror.
                                            -ll links the lex library
```

Step 3: Run the program

```
$  ourprog
```

Runtime Communication Between lex and yacc-Generated Modules



Summary

Efficient parsers for languages with the complexity of a typical programming language or command line interface:

- Make use of special purpose constructs, like precedence, that are very important in the target languages.
- May need complex transition functions to capture all the relevant history in the stack.
- Use heuristic rules, like shift instead of reduce, that have been shown to work most of the time.
- Would be very difficult to construct by hand (as a result of all of the above).
- Can easily be built using a tool like yacc.