Scanning and Parsing

Stephen A. Edwards

Columbia University

Fall 2008

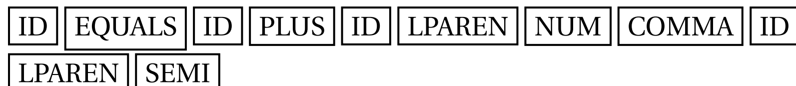
Part I

Lexical Analysis and Ocamllex

Lexical Analysis (Scanning)

Translate a stream of characters to a stream of tokens

f o o ␣ = ␣ a + ␣ bar (0 , ␣ 42 , ␣ q) ;



Token	**Lexemes**	**Pattern**
EQUALS	=	an equals sign
PLUS	+	a plus sign
ID	`a foo bar`	letter followed by letters or digits
NUM	`0 42`	one or more digits

Lexical Analysis

Goal: simplify the job of the parser.

Scanners are usually much faster than parsers.

Discard as many irrelevant details as possible (e.g., whitespace, comments).

Parser does not care that the the identifer is "supercalifragilisticexpialidocious."

Parser rules are only concerned with tokens.

Describing Tokens

Alphabet: A finite set of symbols

Examples: $\{0, 1\}$, $\{A, B, C, \ldots, Z\}$, ASCII, Unicode

String: A finite sequence of symbols from an alphabet

Examples: ϵ (the empty string), Stephen, $\alpha\beta\gamma$

Language: A set of strings over an alphabet

Examples: $\emptyset$ (the empty language), $\{1, 11, 111, 1111\}$, all English words, strings that start with a letter followed by any sequence of letters and digits

Operations on Languages

Let $L = \{ \epsilon, \text{wo} \}$, $M = \{ \text{man}, \text{men} \}$

Concatenation: Strings from one followed by the other

$LM = \{ \text{man}, \text{men}, \text{woman}, \text{women} \}$

Union: All strings from each language

$L \cup M = \{\epsilon, \text{wo}, \text{man}, \text{men} \}$

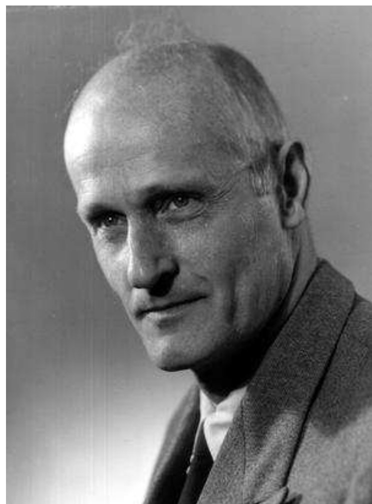
Kleene Closure: Zero or more concatenations

$M^* = \{\epsilon\} \cup M \cup MM \cup MMM \cdots =$
$\{\epsilon, \text{man}, \text{men}, \text{manman}, \text{manmen}, \text{menman}, \text{menmen}, \text{manmanman}, \text{manmanmen}, \text{manmenman}, \ldots\}$

Kleene Closure

The asterisk operator (*) is called the Kleene Closure operator after the inventor of regular expressions, Stephen Cole Kleene, who pronounced his last name "CLAY-nee."

His son Ken writes "As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father."



Regular Expressions over an Alphabet Σ

A standard way to express languages for tokens.

1. ϵ is a regular expression that denotes $\{\epsilon\}$
2. If $a \in \Sigma$, a is an RE that denotes $\{a\}$
3. If r and s denote languages $L(r)$ and $L(s)$,
   - $(r)|(s)$ denotes $L(r) \cup L(s)$
   - $(r)(s)$ denotes $\{tu : t \in L(r), u \in L(s)\}$
   - $(r)^*$ denotes $\cup_{i=0}^{\infty} L^i$ ($L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$)

Regular Expression Examples

$\Sigma = \{a, b\}$

RE	Language
$a|b$	$\{a, b\}$
$(a|b)(a|b)$	$\{aa, ab, ba, bb\}$
a^*	$\{\epsilon, a, aa, aaa, aaaa, \ldots\}$
$(a|b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots\}$
$a|a^*b$	$\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Specifying Tokens with REs

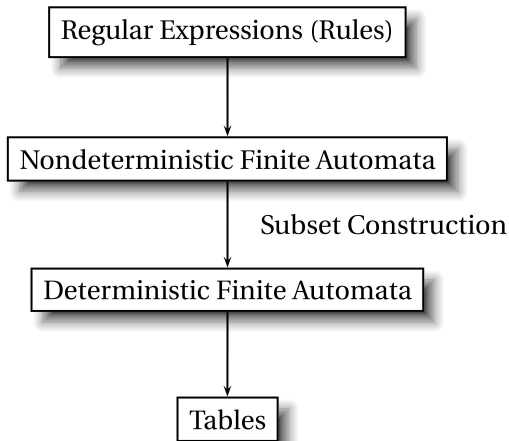
Typical choice: Σ = ASCII characters, i.e.,
{␣, !, ", #, $, ..., 0, 1, ..., 9, ..., A, ..., Z, ..., ~}

letters: $A|B|\cdots|Z|a|\cdots|z$

digits: $0|1|\cdots|9$

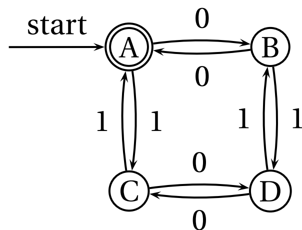
identifier: **letter** (**letter** | **digit**)*

Implementing Scanners Automatically



Nondeterministic Finite Automata

"All strings containing an even number of 0's and 1's"



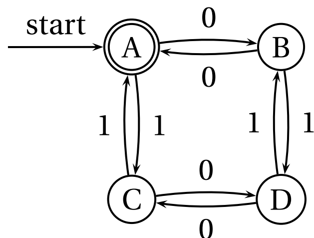
1. Set of states S: {(A), (B), (C), (D)}
2. Set of input symbols Σ: $\{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \rightarrow 2^S$

state	ϵ	0	1
A	–	{B}	{C}
B	–	{A}	{D}
C	–	{D}	{A}
D	–	{C}	{B}

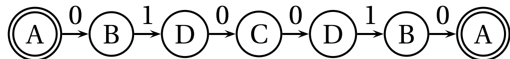
4. Start state s_0 : (A)
5. Set of accepting states F: {(A)}

The Language induced by an NFA

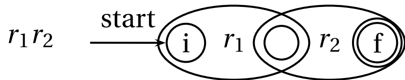
An NFA accepts an input string x iff there is a path from the start state to an accepting state that "spells out" x.

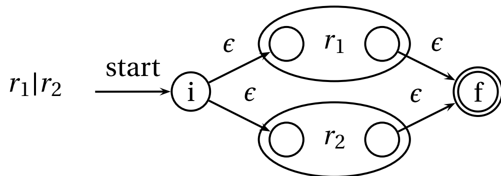


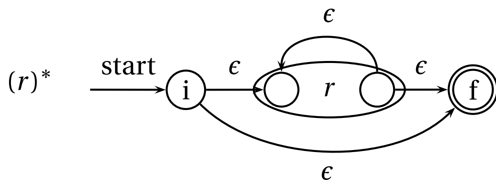
Show that the string "010010" is accepted.



Translating REs into NFAs



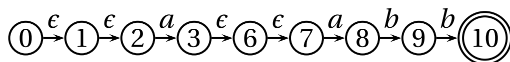




Translating REs into NFAs

Example: translate $(a|b)^* abb$ into an NFA

Show that the string "*aabb*" is accepted.



Simulating NFAs

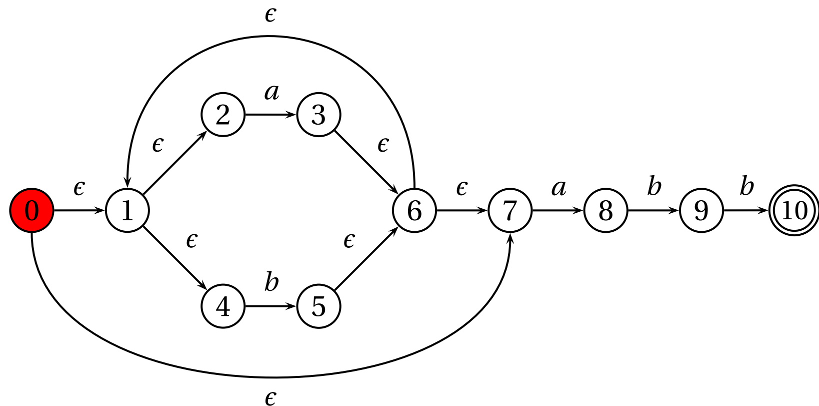
Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

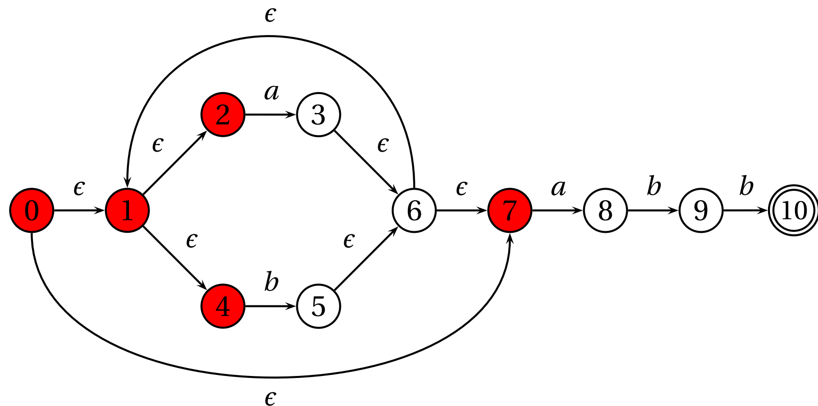
"Two-stack" NFA simulation algorithm:

1. Initial states: the ϵ-closure of the start state
2. For each character c,
   - New states: follow all transitions labeled c
   - Form the ϵ-closure of the current states
3. Accept if any final state is accepting

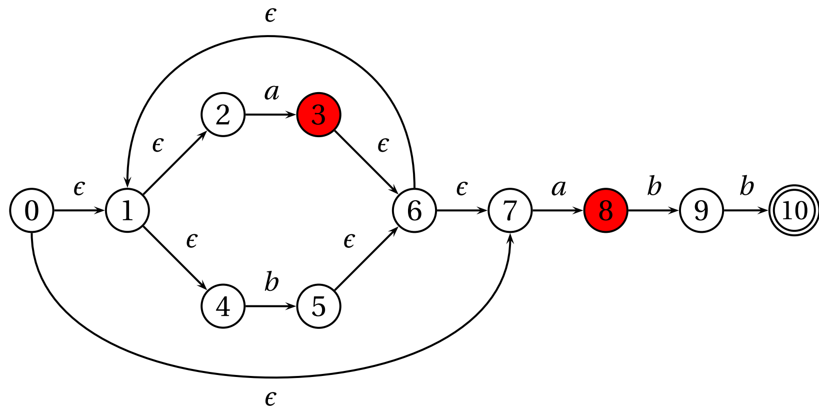
Simulating an NFA: $\cdot aabb$, Start



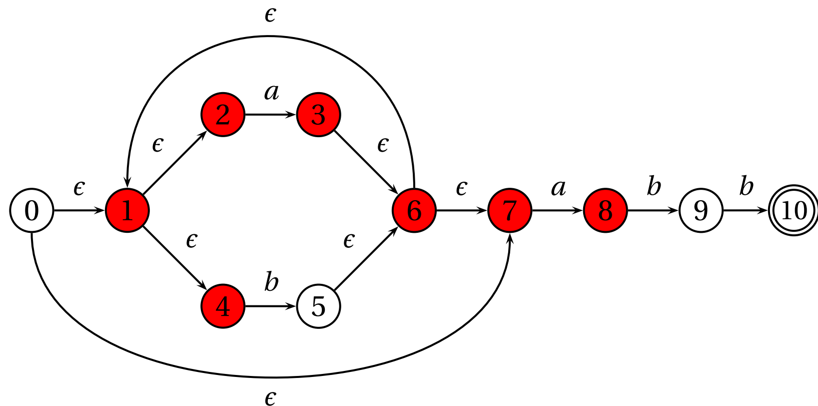
Simulating an NFA: $\cdot aabb$, ϵ-closure



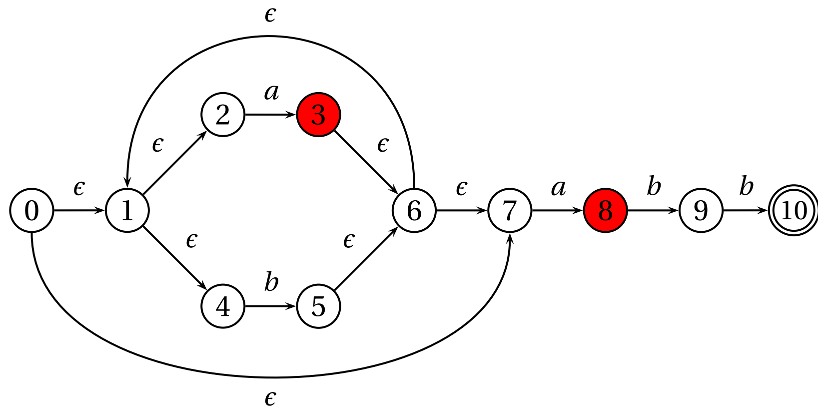
Simulating an NFA: $a \cdot abb$



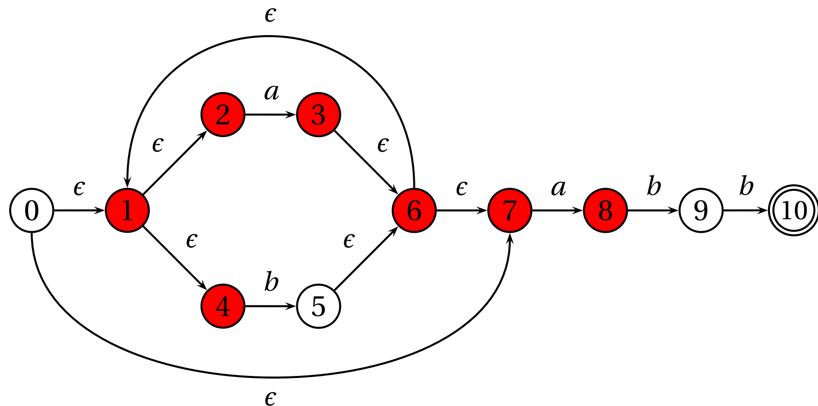
Simulating an NFA: $a{\cdot}abb$, ϵ-closure



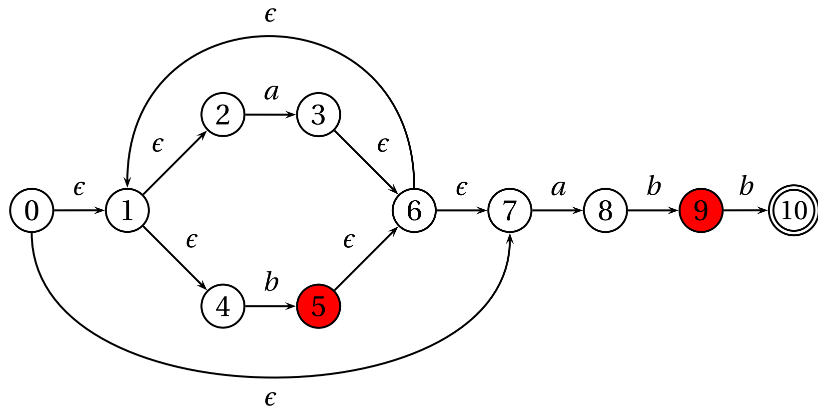
Simulating an NFA: $aa{\cdot}bb$



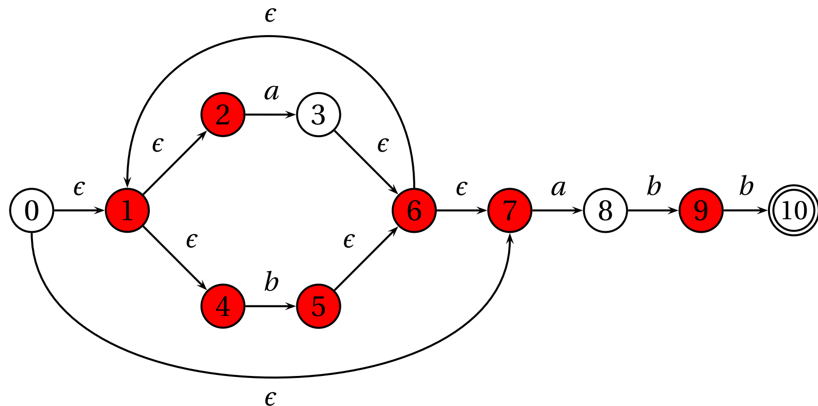
Simulating an NFA: $aa \cdot bb$, ϵ-closure



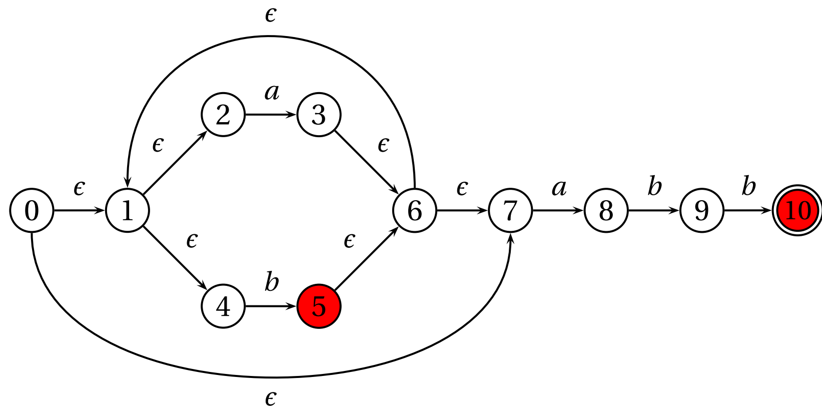
Simulating an NFA: $aab{\cdot}b$



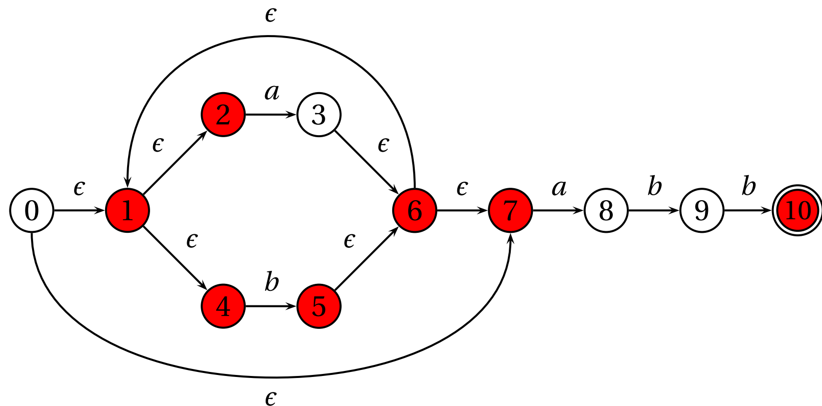
Simulating an NFA: $aab{\cdot}b$, ϵ-closure



Simulating an NFA: $aabb\cdot$



Simulating an NFA: $aabb\cdot$, Done



Deterministic Finite Automata

Restricted form of NFAs:

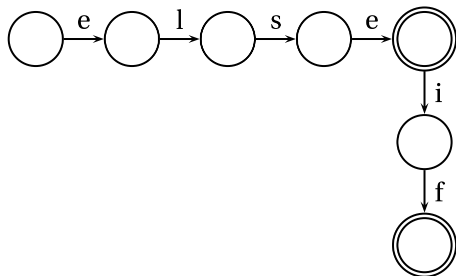
- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

Differs subtly from the definition used in COMS W3261 (Sipser, *Introduction to the Theory of Computation*)

Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.

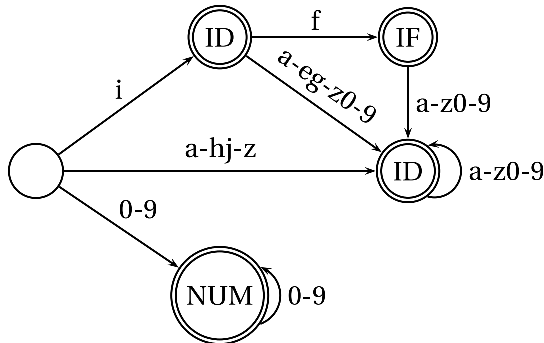
Deterministic Finite Automata

```
{
   type token = ELSE | ELSEIF
}

rule token =
  parse "else"   { ELSE }
      | "elseif" { ELSEIF }
```



Deterministic Finite Automata

```
{ type token = IF | ID of string | NUM of string }

rule token =
  parse "if"                                   { IF }
      | ['a'-'z'] ['a'-'z' '0'-'9']* as lit { ID(lit) }
      | ['0'-'9']+                      as num { NUM(num) }
```



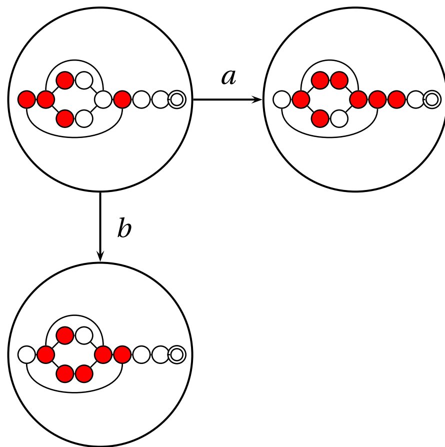
Building a DFA from an NFA

Subset construction algorithm

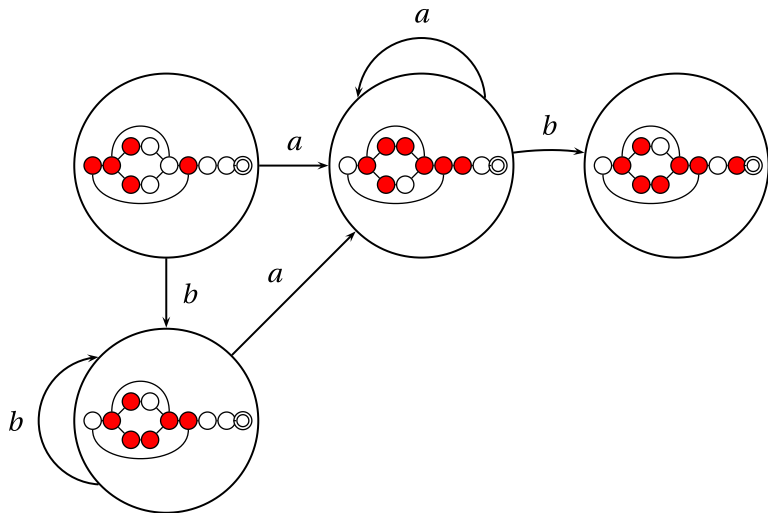
Simulate the NFA for all possible inputs and track the states that appear.

Each unique state during simulation becomes a state in the DFA.

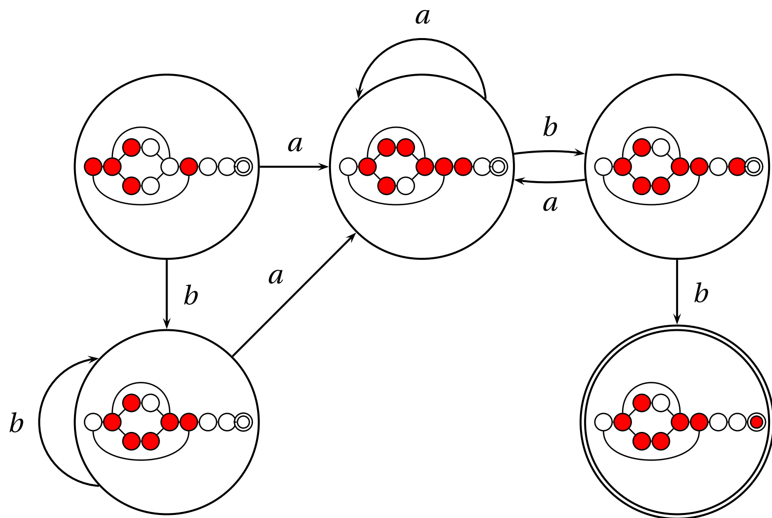
Subset construction for $(a|b)^* abb$ (1)



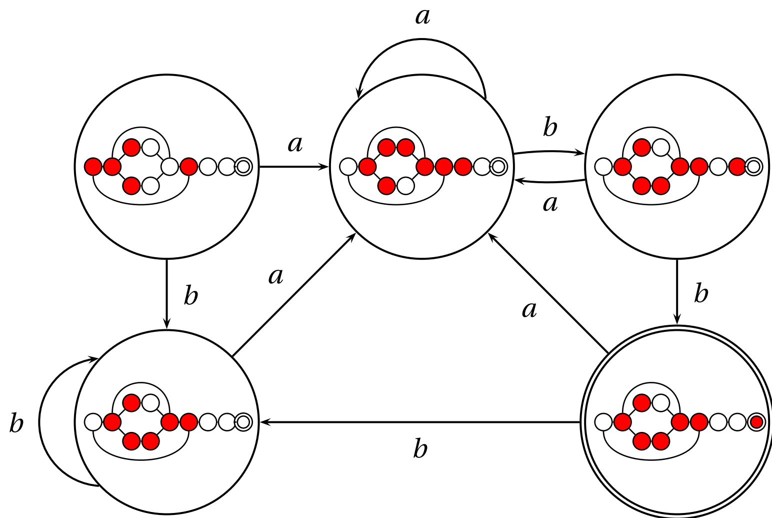
Subset construction for $(a|b)^* abb$ (2)



Subset construction for $(a|b)^* abb$ (3)



Subset construction for $(a|b)^*abb$ (4)



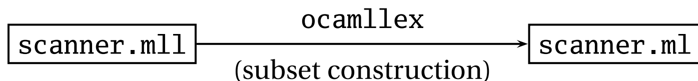
Subset Construction

An DFA can be exponentially larger than the corresponding NFA.

n states versus 2^n

Tools often try to strike a balance between the two representations.

Constructing Scanners with Ocamllex



An example:

scanner.mll

```
{ open Parser }

rule token =
  parse [' ' '\t' '\r' '\n'] { token lexbuf }
      | '+' { PLUS }
      | '-' { MINUS }
      | '*' { TIMES }
      | '/' { DIVIDE }
      | ['0'-'9']+ as lit { LITERAL(int_of_string lit) }
      | eof { EOF }
```

Ocamllex Specifications

```
{
  (* Header: verbatim OCaml code; mandatory *)
}

(* Definitions: optional *)
let ident = regexp
let ...

(* Rules: mandatory *)
rule entrypoint [arg1 ... argn] =
  parse pattern { action (* OCaml code *) }
      | ...
      | pattern { action }
and entrypoint [arg1 ... argn] =
  ...
and ...

{
  (* Trailer: verbatim OCaml code; optional *)
}
```

Patterns (In Order of Decreasing Precedence)

<table>
<tr><th>Pattern</th><th>Meaning</th></tr>
<tr><td><code>'c'</code></td><td>A single character</td></tr>
<tr><td><code>_</code></td><td>Any character (underline)</td></tr>
<tr><td><code>eof</code></td><td>The end-of-file</td></tr>
<tr><td><code>"foo"</code></td><td>A literal string</td></tr>
<tr><td><code>['1' '5' 'a'-'z']</code></td><td>“1,” “5,” or any lowercase letter</td></tr>
<tr><td><code>[^ '0'-'9']</code></td><td>Any character except a digit</td></tr>
<tr><td><code>(</code> pattern <code>)</code></td><td>Grouping</td></tr>
<tr><td>identifier</td><td>A pattern defined in the <code>let</code> section</td></tr>
<tr><td>pattern <code>*</code></td><td>Zero or more patterns</td></tr>
<tr><td>pattern <code>+</code></td><td>One or more patterns</td></tr>
<tr><td>pattern <code>?</code></td><td>Zero or one patterns</td></tr>
<tr><td>$pattern_1$ $pattern_2$</td><td>$pattern_1$ followed by $pattern_2$</td></tr>
<tr><td>$pattern_1$ <code>|</code> $pattern_2$</td><td>Either $pattern_1$ or $pattern_2$</td></tr>
<tr><td>pattern <code>as</code> id</td><td>Bind the matched pattern to variable id</td></tr>
</table>

An Example

```
{ type token = PLUS | IF | ID of string | NUM of int }

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']

rule token =
 parse [' ' '\n' '\t'] { token lexbuf } (* Ignore whitespace *)

     | '+' { PLUS }                    (* A symbol *)

     | "if" { IF }                     (* A keyword *)
                                       (* Identifiers *)
     | letter (letter | digit | '_')* as id { ID(id) }
                                       (* Numeric literals *)
     | digit+ as lit { NUM(int_of_string lit) }

     | "/*" { comment lexbuf }         (* C-style comments *)

and comment =
  parse "*/" { token lexbuf } (* Return to normal scanning *)
      | _ { comment lexbuf }  (* Ignore other characters *)
```

Free-Format Languages

Typical style arising from scanner/parser division

Program text is a series of tokens possibly separated by whitespace and comments, which are both ignored.

- keywords (`if while`)
- punctuation (`, ( +`)
- identifiers (`foo bar`)
- numbers (`10 -3.14159e+32`)
- strings (`"A String"`)

Free-Format Languages

Java C C++ Algol Pascal

Some deviate a little (e.g., C and C++ have a separate preprocessor)

But not all languages are free-format.

FORTRAN 77

FORTRAN 77 is not free-format. 72-character lines:

```
100    IF(IN .EQ. 'Y' .OR. IN .EQ. 'y' .OR.
     $   IN .EQ. 'T' .OR. IN .EQ. 't') THEN
```


When column 6 is not a space, line is considered part of the previous.

Fixed-length line works well with a one-line buffer.

Makes sense on punch cards.

Python

The Python scripting language groups with indentation

```
i = 0
while i < 10:
    i = i + 1
    print i    # Prints 1, 2, ..., 10

i = 0
while i < 10:
    i = i + 1
print i        # Just prints 10
```

This is succinct, but can be error-prone.

How do you wrap a conditional around instructions?

Syntax and Language Design

Does syntax matter? Yes and no

More important is a language's *semantics*—its meaning.

The syntax is aesthetic, but can be a religious issue.

But aesthetics matter to people, and can be critical.

Verbosity does matter: smaller is usually better.

Too small can be problematic: APL is a succinct language with its own character set. Here is program that returns all primes ≤ R

```
PRIMES : (~R∈R∘.×R)/R←1↓⍳R
```

There are no APL programs, only puzzles.

Syntax and Language Design

Some syntax is error-prone. Classic FORTRAN example:

```
DO 5 I = 1,25  ! Loop header (for i = 1 to 25)
DO 5 I = 1.25  ! Assignment to variable DO5I
```

Trying too hard to reuse existing syntax in C++:

```
vector< vector<int> > foo;
vector<vector<int>> foo; // Syntax error
```

C distinguishes > and » as different operators.

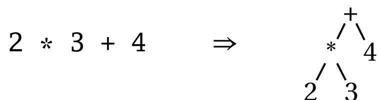
Bjarne Stroustrup tells me they have finally fixed this.

Part II

Syntactic Analysis and Ocamlyacc

Parsing

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.



Goal: discard irrelevant information to make it easier for the next stage.

Parentheses and most other forms of punctuation removed.

Grammars

Most programming languages described using a *context-free grammar*.

Compared to regular languages, context-free languages add one important thing: recursion.

Recursion allows you to count, e.g., to match pairs of nested parentheses.

Which languages do humans speak? I'd say it's regular: I do not not not not not not not not not not understand this sentence.

Languages

Regular languages (t is a terminal):

$$A \rightarrow t_1 \ldots t_n B$$
$$A \rightarrow t_1 \ldots t_n$$

Context-free languages (P is terminal or a variable):

$$A \rightarrow P_1 \ldots P_n$$

Context-sensitive languages:

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 B \alpha_2$$

"$B \rightarrow A$ only in the 'context' of $\alpha_1 \cdots \alpha_2$"

Issues

Ambiguous grammars

Precedence of operators

Left- versus right-recursive

Top-down vs. bottom-up parsers

Parse Tree vs. Abstract Syntax Tree

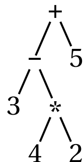
Ambiguous Grammars

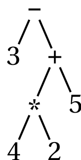
A grammar can easily be ambiguous. Consider parsing

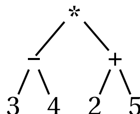
```
3 - 4 * 2 + 5
```

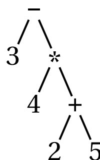
with the grammar

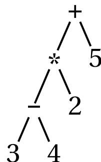
$e \to e + e \mid e - e \mid e * e \mid e / e \mid N$











Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions

Like you were taught in elementary school:

"My Dear Aunt Sally"

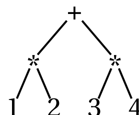
Mnemonic for multiplication and division before addition and subtraction.

Operator Precedence

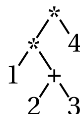
Defines how "sticky" an operator is.

`1 * 2 + 3 * 4`

`*` at higher precedence than +:
`(1 * 2) + (3 * 4)`



+ at higher precedence than `*`:
`1 * (2 + 3) * 4`

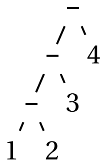


Associativity

Whether to evaluate left-to-right or right-to-left

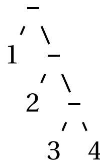
Most operators are left-associative

```
1 - 2 - 3 - 4
```



((1 - 2) - 3) - 4

left associative



1 - (2 - (3 - 4))

right associative

Fixing Ambiguous Grammars

A grammar specification:

```
expr :
     expr PLUS expr     {}
   | expr MINUS expr    {}
   | expr TIMES expr    {}
   | expr DIVIDE expr   {}
   | NUMBER             {}
;
```

Ambiguous: no precedence or associativity.

Ocamlyacc's complaint: "16 shift/reduce conflicts."

Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr PLUS expr    {}
     | expr MINUS expr   {}
     | term              {}
;

term : term TIMES term   {}
     | term DIVIDE term  {}
     | atom              {}
;

atom  : NUMBER           {}
;
```

Still ambiguous: associativity not defined

Ocamlyacc's complaint: "8 shift/reduce conflicts."

Assigning Associativity

Make one side the next level of precedence

```
expr : expr PLUS term    {}
     | expr MINUS term   {}
     | term              {}
;

term : term TIMES atom   {}
     | term DIVIDE atom  {}
     | atom              {}
;

atom  : NUMBER           {}
;
```

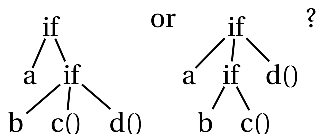
This is left-associative.

No shift/reduce conflicts.

The Dangling Else Problem

Who owns the *else*?

```
if (a) if (b) c(); else d();
```



Grammars are usually ambiguous; manuals give disambiguating rules such as C's:

> *As usual the "else" is resolved by connecting an else with the last encountered elseless if.*

The Dangling Else Problem

```
stmt : "if" expr "then" stmt iftail
     | other-statements ;

iftail
   : "else" stmt
   | /* nothing */
   ;
```

Problem comes when matching “iftail.”

Normally, an empty choice is taken if the next token is in the “follow set” of the rule. But since “else” can follow an iftail, the decision is ambiguous.

The Dangling Else Problem

Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

```
if a < b then a else b fi;
```

"fi" is "if" spelled backwards. The language also uses do–od and case–esac.

Statement separators/terminators

C uses ; as a statement terminator.

```
if (a<b) printf("a less");
else {
  printf("b"); printf(" less");
}
```

Pascal uses ; as a statement separator.

```
if a < b then writeln('a less')
else begin
  write('a'); writeln(' less')
end
```

Pascal later made a final ; optional.

Parsing Context-Free Grammars

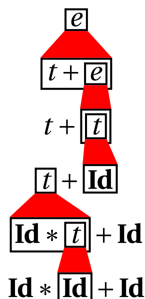
There are $O(n^3)$ algorithms for parsing arbitrary CFGs, but most compilers demand $O(n)$ algorithms.

Fortunately, the LL and LR subclasses of CFGs have $O(n)$ parsing algorithms. People use these in practice.

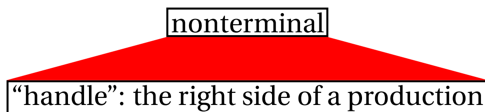
Rightmost Derivation

$$\begin{aligned} 1: &\quad e \rightarrow t + e \\ 2: &\quad e \rightarrow t \\ 3: &\quad t \rightarrow \mathbf{Id} * t \\ 4: &\quad t \rightarrow \mathbf{Id} \end{aligned}$$

The rightmost derivation for $\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$:



At each step, expand the rightmost nonterminal.

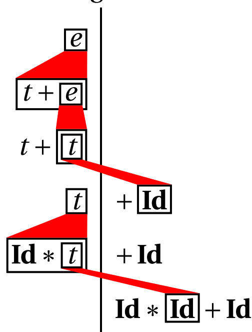


Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambigious.

Rightmost Derivation

1 : $e \rightarrow t + e$
2 : $e \rightarrow t$
3 : $t \rightarrow \textbf{Id} * t$
4 : $t \rightarrow \textbf{Id}$

The rightmost derivation for **Id** $*$ **Id** $+$ **Id**:



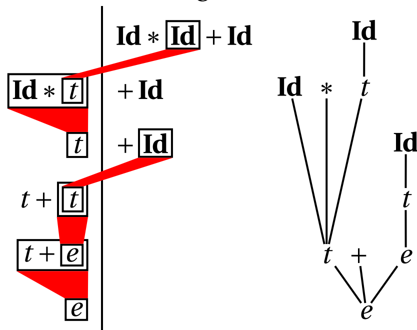
Tokens on the right are all terminals.

In each step, nonterminal just to the left is expanded.

Reverse Rightmost Derivation

$$
\begin{aligned}
1:&\quad e \rightarrow t + e\\
2:&\quad e \rightarrow t\\
3:&\quad t \rightarrow \textbf{Id} * t\\
4:&\quad t \rightarrow \textbf{Id}
\end{aligned}
$$

The reverse rightmost derivation for **Id** $*$ **Id** $+$ **Id**:



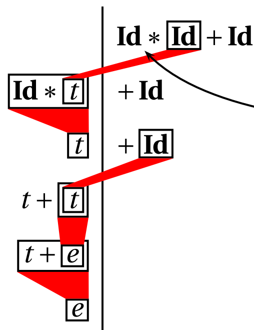
Beginning to look like a parsing algorithm: start with terminals and reduce them to the starting nonterminal.

Reductions build the parse tree starting at the leaves.

Reverse Rightmost Derivation

$$
\begin{aligned}
1: &\quad e \rightarrow t + e \\
2: &\quad e \rightarrow t \\
3: &\quad t \rightarrow \textbf{Id} * t \\
4: &\quad t \rightarrow \textbf{Id}
\end{aligned}
$$

The reverse rightmost derivation for **Id** $*$ **Id** $+$ **Id**:



Big question: where are the handles?

A handle is the right side of a production, but not always vice-versa.

A handle is the result of an expansion in the rightmost derivation.

Every step in a rightmost derivation is a right sentential form.

Handle Hunting

The basic trick, due to Knuth: build an automaton that tells us where the handle is in right-sentential forms.

Represent where we could be with a dot.

$$e \rightarrow \cdot t + e$$
$$e \rightarrow \cdot t$$
$$t \rightarrow \cdot \mathbf{Id} * t$$
$$t \rightarrow \cdot \mathbf{Id}$$

The first two come from expanding e. The second two come from expanding t.

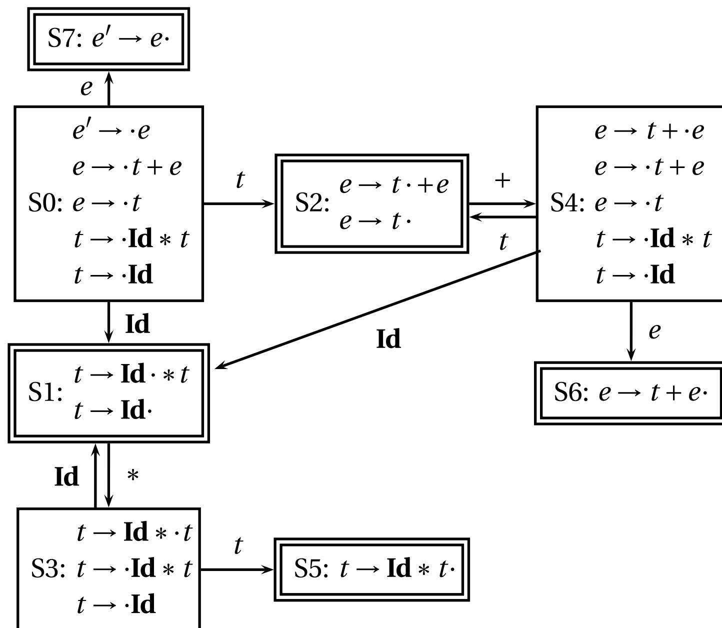
Consider the expansion of e first. This gives two possible positions:

$e \rightarrow t \cdot +e$	when e was expanded to $t + e$
$e \rightarrow t\cdot$	when e was expanded to just t; t is a handle

The expanded-t case also gives two possible positions:

$t \rightarrow \mathbf{Id} \cdot * t$	when t was expanded to $\mathbf{Id} + t$
$t \rightarrow \mathbf{Id}\cdot$	when y was expanded to just **Id**; **Id** is a handle

Constructing the LR(0) Automaton



Shift-reduce Parsing

$1: \quad e \rightarrow t + e$

$2: \quad e \rightarrow t$

$3: \quad t \rightarrow \mathbf{Id} * t$

$4: \quad t \rightarrow \mathbf{Id}$

stack	input	action
	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id}$	$* \mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id} *$	$\mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id} * \mathbf{Id}$	$+ \mathbf{Id}$	reduce (4)
$\mathbf{Id} * t$	$+ \mathbf{Id}$	reduce (3)
t	$+ \mathbf{Id}$	shift
$t +$	$\mathbf{Id}$	shift
$t + \mathbf{Id}$		reduce (4)
$t + t$		reduce (2)
$t + e$		reduce (1)
e		accept

Scan input left-to-right, looking for handles.

An oracle says what to do

LR Parsing

$$1: \quad e \rightarrow t + e$$
$$2: \quad e \rightarrow t$$
$$3: \quad t \rightarrow \textbf{Id} * t$$
$$4: \quad t \rightarrow \textbf{Id}$$

	action				goto	
	Id	+	*	$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

stack	input	action
0	**Id** * **Id** + **Id** $	shift, goto 1

1. Look at state on top of stack
2. and the next input token
3. to find the next action
4. In this case, shift the token onto the stack and go to state 1.

LR Parsing

$1: \quad e \rightarrow t + e$

$2: \quad e \rightarrow t$

$3: \quad t \rightarrow \textbf{Id} * t$

$4: \quad t \rightarrow \textbf{Id}$

	action				goto	
	Id	+	*	$	*e*	*t*
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

stack	input	action
[0]	**Id** * **Id** + **Id** $	shift, goto 1
[0] [Id 1]	* **Id** + **Id** $	shift, goto 3
[0] [Id 1] [* 3]	**Id** + **Id** $	shift, goto 1
[0] [Id 1] [* 3] [Id 1]	+ **Id** $	reduce w/ 4

Action is "reduce with rule 4 ($t \rightarrow \textbf{Id}$)." The right side is removed from the stack to reveal state 3. The goto table in state 3 tells us to go to state 5 when we reduce a t:

stack	input	action
[0] [Id 1] [* 3] [*t* 5]	+ **Id** $	

LR Parsing

$1: \quad e \rightarrow t + e$

$2: \quad e \rightarrow t$

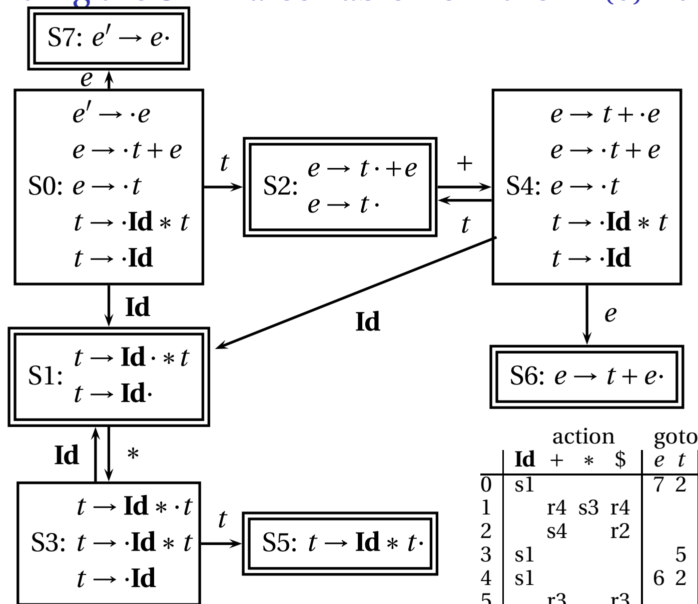
$3: \quad t \rightarrow \mathbf{Id} * t$

$4: \quad t \rightarrow \mathbf{Id}$

	action				goto	
	Id	+	*	$	*e*	*t*
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

stack	input	action
0	**Id** * **Id** + **Id** $	shift, goto 1
0 Id/1	* **Id** + **Id** $	shift, goto 3
0 Id/1 */3	**Id** + **Id** $	shift, goto 1
0 Id/1 */3 Id/1	+ **Id** $	reduce w/ 4
0 Id/1 */3 *t*/5	+ **Id** $	reduce w/ 3
0 t/2	+ **Id** $	shift, goto 4
0 t/2 +/4	**Id** $	shift, goto 1
0 t/2 +/4 Id/1	$	reduce w/ 4
0 t/2 +/4 *t*/2	$	reduce w/ 2
0 t/2 +/4 *e*/6	$	reduce w/ 1
0 e/7	$	accept

Building the SLR Parse Table from the LR(0) Automaton



	action				goto	
	Id	+	*	$	*e*	*t*
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

The Punchline

This is a tricky, but mechanical procedure. The Ocamlyacc parser generator uses a modified version of this technique to generate fast bottom-up parsers.

You need to understand it to comprehend error messages:

Shift/reduce conflicts are caused by a state like

$t \rightarrow \cdot\, \textbf{Else}\; s$
$t \rightarrow \cdot$

If the next token is **Else**, do you reduce it since **Else** may follow a t, or shift it?

Reduce/reduce conflicts are caused by a state like

$t \rightarrow \textbf{Id} * t\, \cdot$
$e \rightarrow t + e\, \cdot$

Do you reduce by "$t \rightarrow \textbf{Id} * t$" or by "$e \rightarrow t + e$"?

Ocamlyacc Specifications

```
%{
  (* Header: verbatim OCaml; optional *)
%}

  /* Declarations: tokens, precedence, etc. */

%%

  /* Rules: context-free rules */

%%

  (* Trailer: verbatim OCaml; optional *)
```

Declarations

- `%token` *symbol* ...
  Define symbol names (exported to .mli file)
- `%token` < *type* > *symbol* ...
  Define symbols with attached attribute (also exported)
- `%start` *symbol* ...
  Define start symbols (entry points)
- `%type` < *type* > *symbol* ...
  Define the type for a symbol (mandatory for start)
- `%left` *symbol* ...
- `%right` *symbol* ...
- `%nonassoc` *symbol* ...
  Define predecence and associtivity for the given symbols, listed in order from lowest to highest precedence

Rules

```
nonterminal :
    symbol ... symbol { semantic-action }
  | ...
  | symbol ... symbol { semantic-action }
;
```

- *nonterminal* is the name of a rule, e.g., "program," "expr"
- *symbol* is either a terminal (token) or another rule
- *semantic-action* is OCaml code evaluated when the rule is matched
- In a *semantic-action*, `$1`, `$2`, ...returns the value of the first, second, ...symbol matched
- A rule may include "`%prec` *symbol*" to override its default precedence

An Example .mly File

```
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token EOL
%left PLUS MINUS /* lowest precedence */
%left TIMES DIV  /* medium precedence */
%nonassoc UMINUS /* highest precedence */
%start main      /* the entry point */
%type <int> main

%%

main:
    expr EOL                { $1 }
;
expr:
    INT                     { $1 }
  | LPAREN expr RPAREN      { $2 }
  | expr PLUS expr          { $1 + $3 }
  | expr MINUS expr         { $1 - $3 }
  | expr TIMES expr         { $1 * $3 }
  | expr DIV expr           { $1 / $3 }
  | MINUS expr %prec UMINUS { - $2 }
;
```