Syntax and Parsing

COMS W4115

Prof. Stephen A. Edwards
Fall 2006
Columbia University
Department of Computer Science

Lexical Analysis (Scanning)

Lexical Analysis (Scanning)

Translates a stream of characters to a stream of tokens

`f o o ␣ = ␣ a + ␣ bar(2, ␣ q);`

ID EQUALS ID PLUS ID LPAREN NUM COMMA ID LPAREN SEMI

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

Concatenation: Strings from one followed by the other

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

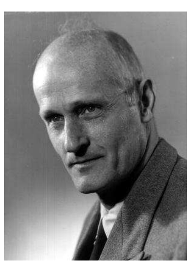
Union: All strings from each language

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

Kleene Closure: Zero or more concatenations

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

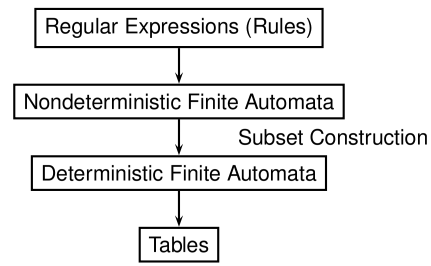
Typical choice: $\Sigma =$ ASCII characters, i.e., $\{␣, !, ", \#, \$, \ldots, 0, 1, \ldots, 9, \ldots, A, \ldots, Z, \ldots, \tilde{}\}$

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

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

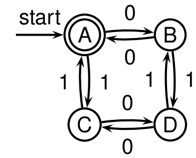
identifier: **letter** (**letter** | **digit**)*

Implementing Scanners Automatically



Nondeterministic Finite Automata

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



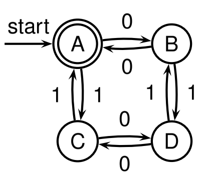
1. Set of states S: {A, B, C, D}
2. Set of input symbols Σ: $\{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \to 2^S$

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

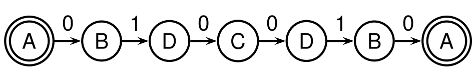
4. Start state s_0 : A
5. Set of accepting states F: {A}

The Language induced by an NFA

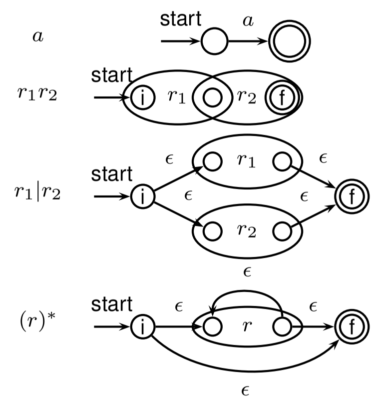
An NFA accepts an input string x iff there is a path from the start state to an accepting state that "spells out" x.



Show that the string "010010" is accepted.

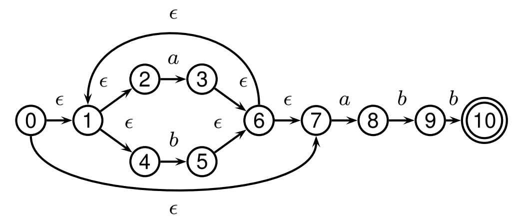


Translating REs into NFAs

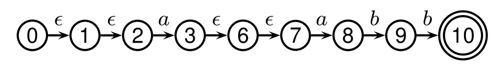


Translating REs into NFAs

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



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



Simulating NFAs

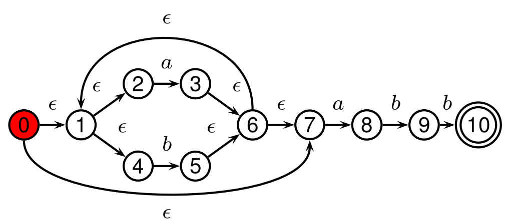
Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

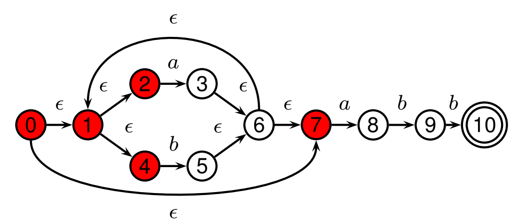
"Two-stack" NFA simulation algorithm:

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

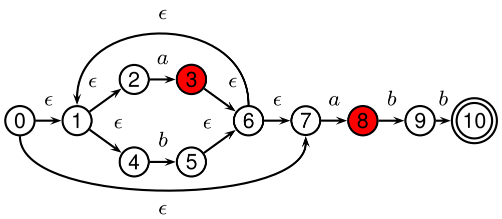
Simulating an NFA: $\cdot aabb$, Start



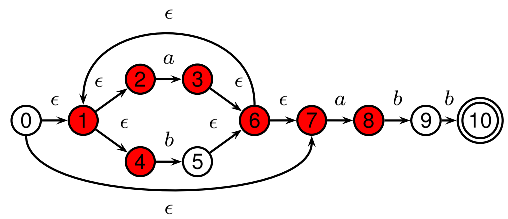
Simulating an NFA: $\cdot aabb$, ϵ-closure



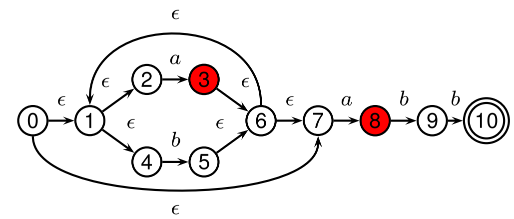
Simulating an NFA: $a{\cdot}abb$



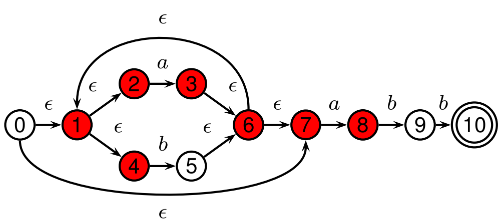
Simulating an NFA: $a{\cdot}abb$, ϵ-closure



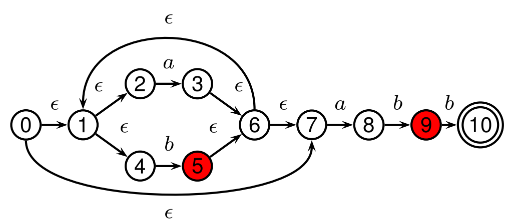
Simulating an NFA: $aa{\cdot}bb$



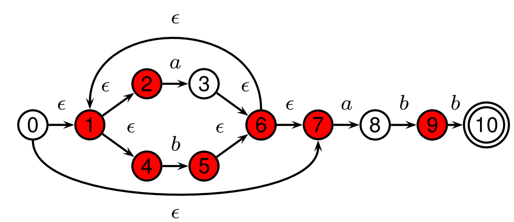
Simulating an NFA: $aa{\cdot}bb$, ϵ-closure



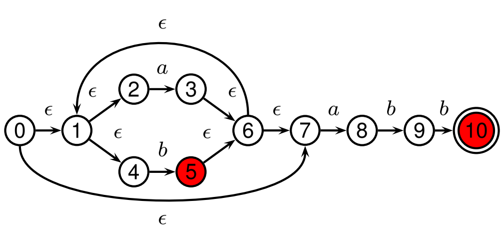
Simulating an NFA: $aab{\cdot}b$



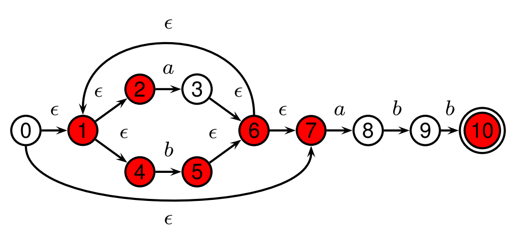
Simulating an NFA: $aab{\cdot}b$, ϵ-closure



Simulating an NFA: $aabb{\cdot}$



Simulating an NFA: $aabb{\cdot}$, Done



Deterministic Finite Automata

Restricted form of NFAs:

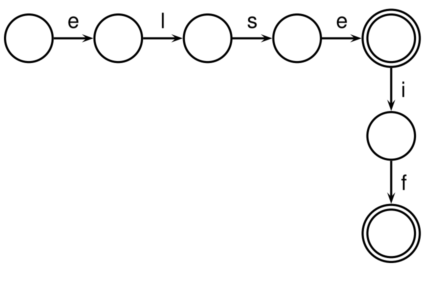
- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

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.

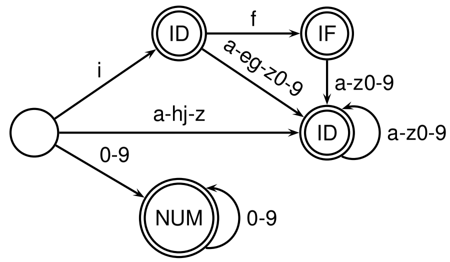
Deterministic Finite Automata

```
ELSE: "else" ;
ELSEIF: "elseif" ;
```



Deterministic Finite Automata

```
IF: "if" ;
ID: 'a'..'z' ('a'..'z' | '0'..'9')* ;
NUM: ('0'..'9')+ ;
```



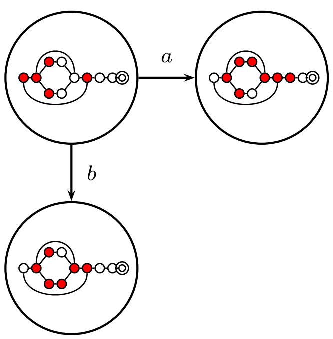
Building a DFA from an NFA

Subset construction algorithm

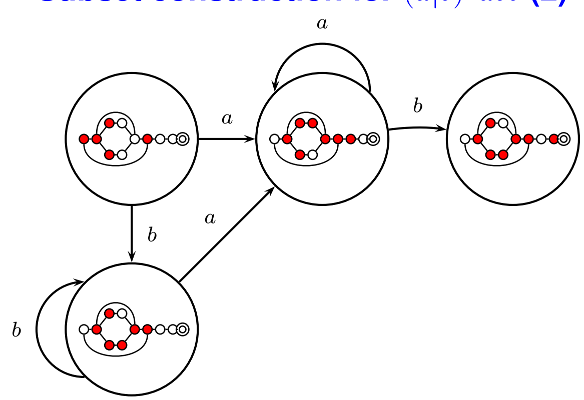
Simulate the NFA for all possible inputs and track the states that appear.

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

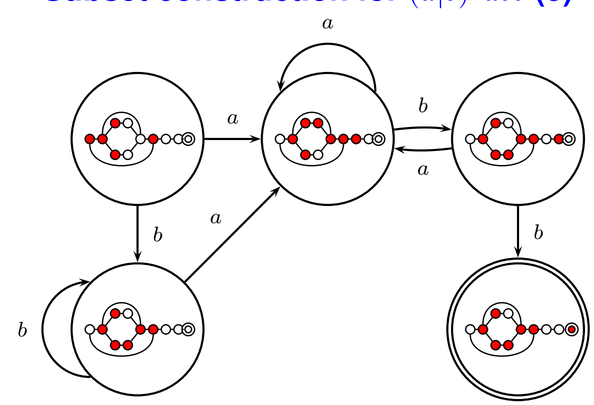
Subset construction for $(a|b)^*abb$ (1)



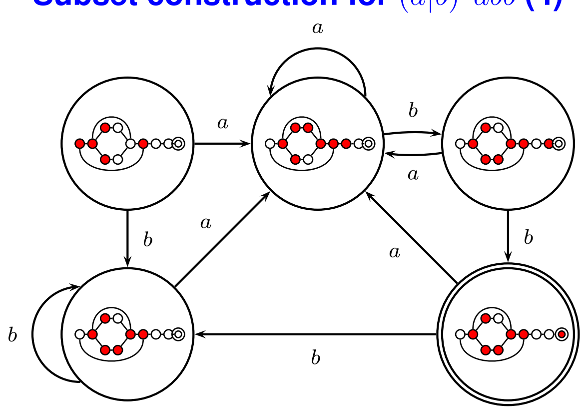
Subset construction for $(a|b)^*abb$ (2)



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.

ANTLR uses a different technique.

The ANTLR Compiler Generator

Language and compiler for writing compilers

Running ANTLR on an ANTLR file produces Java source files that can be compiled and run.

ANTLR can generate

- Scanners (lexical analyzers)
- Parsers
- Tree walkers

An ANTLR File for a Simple Scanner

```
class CalcLexer extends Lexer;

LPAREN : '(' ;          // Rules for puctuation
RPAREN : ')' ;
STAR : '*' ;
PLUS : '+' ;
SEMI : ';' ;
protected               // Can only be used as a sub-rule
DIGIT : '0'..'9' ;      // Any character between 0 and 9
INT : (DIGIT)+ ;        // One or more digits

WS : (' ' | '\t' | '\n'| '\r')      // Whitespace
     { $setType(Token.SKIP); } ;    // Action: ignore
```

ANTLR Specifications for Scanners

Rules are names starting with a capital letter.

A character in single quotes matches that character.

```
LPAREN : '(' ;
```

A string in double quotes matches the string

```
IF : "if" ;
```

A vertical bar indicates a choice:

```
OP : '+' | '-' | '*' | '/' ;
```

ANTLR Specifications

Question mark makes a clause optional.

```
PERSON : ("wo")? 'm' ('a'|'e') 'n' ;
```

(Matches man, men, woman, and women.)

Double dots indicate a range of characters:

```
DIGIT : '0'..'9';
```

Asterisk and plus match “zero or more,” “one or more.”

```
ID : LETTER (LETTER | DIGIT)* ;
NUMBER : (DIGIT)+ ;
```

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
```

1 ··· 5 — Statement label; 6 — Continuation; 7 ··· 72 — Normal

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 Langauge 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 a problem: APL is a compact, cryptic language with its own character set (!)

```
E←A TEST B;L
L←0.5
E←((A×A)+B×B)*L
```

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
```

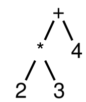
C distinguishes > and >> as different operators.

Parsing

Parsing

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

`2 * 3 + 4` ⇒



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 understand this sentence.

Languages

Regular languages (t is a terminal):

$A \to t_1 \ldots t_n B$

$A \to t_1 \ldots t_n$

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

$A \to P_1 \ldots P_n$

Context-sensitive languages:

$\alpha_1 A \alpha_2 \to \alpha_1 B \alpha_2$

"$B \to A$ 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

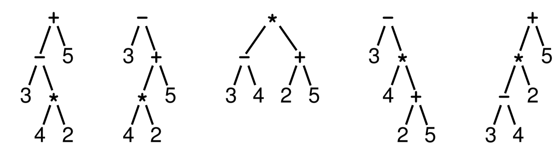
Ambiguous Grammars

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"

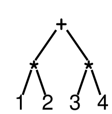
Mnemonic for multiplication and division before addition and subtraction.

Operator Precedence

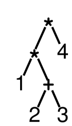
Defines how "sticky" an operator is.

`1 * 2 + 3 * 4`

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



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

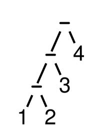


Associativity

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

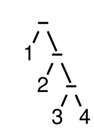
Most operators are left-associative

`1 - 2 - 3 - 4`



((1 - 2) - 3) - 4

left associative



1 - (2 - (3 - 4))

right associative

Fixing Ambiguous Grammars

Original ANTLR grammar specification

```
expr
  : expr '+' expr
  | expr '-' expr
  | expr '*' expr
  | expr '/' expr
  | NUMBER
  ;
```

Ambiguous: no precedence or associativity.

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.

Writing LL(k) Grammars

Cannot have left-recursion

```
expr : expr '+' term | term ;
```

becomes

```
AST expr() {
  switch (next-token) {
  case NUMBER : expr(); /* Infinite Recursion */
```

Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr '+' expr
     | expr '-' expr
     | term ;

term : term '*' term
     | term '/' term
     | atom ;

atom : NUMBER ;
```

Still ambiguous: associativity not defined

Parsing LL(k) Grammars

LL: Left-to-right, Left-most derivation

k: number of tokens to look ahead

Parsed by top-down, predictive, recursive parsers

Basic idea: look at the next token to predict which production to use

ANTLR builds recursive LL(k) parsers

Almost a direct translation from the grammar.

Writing LL(1) Grammars

Cannot have common prefixes

```
expr : ID '(' expr ')'
     | ID '=' expr
```

becomes

```
expr() {
  switch (next-token) {
  case ID:
    match(ID); match(LPAR); expr(); match(RPAR);  break;
  case ID:
    match(ID); match(EQUALS); expr();              break;
```

Assigning Associativity

Make one side or the other the next level of precedence

```
expr : expr '+' term
     | expr '-' term
     | term ;

term : term '*' atom
     | term '/' atom
     | atom ;

atom : NUMBER ;
```

Implementing a Top-Down Parser

```
stmt : 'if' expr 'then' expr
     | 'while' expr 'do' expr
     | expr ':=' expr ;
expr : NUMBER | '(' expr ')' ;
```

```
stmt() {
  switch (next-token) {
  case IF:
    match(IF); expr(); match(THEN); expr();          break;
  case WHILE:
    match(WHILE); expr(); match(DO); expr();         break;
  case NUMBER or LPAREN:
    expr(); match(COLEQ); expr();                    break;
}}
```

Eliminating Common Prefixes

Consolidate common prefixes:

```
expr
  : expr '+' term
  | expr '-' term
  | term
  ;
```

becomes

```
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

Eliminating Left Recursion

Understand the recursion and add tail rules

```
expr
  : expr ('+' term | '-' term )
  | term
  ;
```

becomes

```
expr : term exprt ;
exprt : '+' term exprt
      | '-' term exprt
      | /* nothing */
      ;
```

Using ANTLR's EBNF

ANTLR makes this easier since it supports * and –:

```
expr : expr '+' term
     | expr '-' term
     | term ;
```

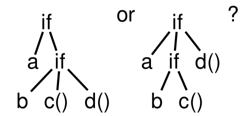
becomes

```
expr : term ('+' term | '-' term)* ;
```

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

ANTLR can resolve this problem by making certain rules "greedy." If a conditional is marked as greedy, it will take that option even if the "nothing" option would also match:

```
stmt
  : "if" expr "then" stmt
     ( options {greedy = true;}
     : "else" stmt
     )?
  | other-statements
  ;
```

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.

Bottom-up Parsing

Rightmost Derivation

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

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

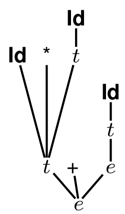
e
$t + e$
$t + t$
$t + \textbf{Id}$
$\textbf{Id} * t + \textbf{Id}$
$\textbf{Id} * \textbf{Id} + \textbf{Id}$

Basic idea of bottom-up parsing: construct this rightmost derivation backward.

Handles

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

Id $*$ **Id** $+$ **Id**
Id $* t$ $+$ **Id**
$t +$ **Id**
$t + t$
$t + e$
e



This is a reverse rightmost derivation for **Id** $*$ **Id** $+$ **Id**.

Each highlighted section is a handle.

Taken in order, the handles build the tree from the leaves to the root.

Shift-reduce Parsing

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

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

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

An oracle tells what to do

LR Parsing

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

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

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 : $e \rightarrow t + e$
2 : $e \rightarrow t$
3 : $t \rightarrow$ **Id** $* t$
4 : $t \rightarrow$ **Id**

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

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 ($t \rightarrow$ **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 : $e \rightarrow t + e$
2 : $e \rightarrow t$
3 : $t \rightarrow$ **Id** $* t$
4 : $t \rightarrow$ **Id**

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

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

Constructing the SLR Parse Table

The states are places we could be in a reverse-rightmost derivation. Let's represent such a place with a dot.

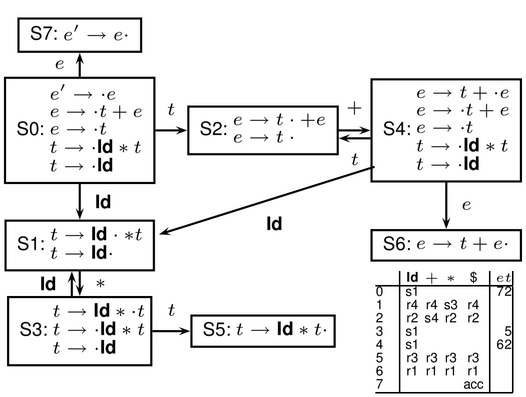
1 : $e \rightarrow t + e$
2 : $e \rightarrow t$
3 : $t \rightarrow$ **Id** $* t$
4 : $t \rightarrow$ **Id**

Say we were at the beginning ($\cdot e$). This corresponds to

$e' \rightarrow \cdot e$
$e \rightarrow \cdot t + e$
$e \rightarrow \cdot t$
$t \rightarrow \cdot$**Id** $* t$
$t \rightarrow \cdot$**Id**

The first is a placeholder. The second are the two possibilities when we're just before e. The last two are the two possibilities when we're just before t.

Constructing the SLR Parsing Table



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

The Punchline

This is a tricky, but mechanical procedure. The parser generators YACC, Bison, Cup, and others (but not ANTLR) use 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

$t \rightarrow$ **Id** $\cdot * t$
$t \rightarrow$ **Id** $* t \cdot$

Reduce/reduce conflicts are caused by a state like

$t \rightarrow$ **Id** $* t \cdot$
$e \rightarrow t + e \cdot$