Review for the Midterm

Stephen A. Edwards

Columbia University

Fall 2008

The Midterm

70 minutes

4–5 problems

Closed book

One sheet of notes of your own devising

Comprehensive: Anything discussed in class is fair game

Little, if any, programming.

Details of O'Caml/C/C++/Java syntax not required

Broad knowledge of languages discussed

Topics

Structure of a Compiler

Scanning and Parsing

Regular Expressions

Context-Free Grammars

Bottom-up Parsing

ASTs

Name, Scope, and Bindings

Part I

Structure of a Compiler

Compiling a Simple Program

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

What the Compiler Sees

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

```
i  n  t sp  g  c  d  (  i  n  t sp  a  , sp  i
n  t sp  b  ) nl  { nl sp sp  w  h  i  l  e sp
(  a sp  !  = sp  b  ) sp  { nl sp sp sp sp  i
f sp  (  a sp  > sp  b  ) sp  a sp  -  = sp  b
; nl sp sp sp sp  e  l  s  e sp  b sp  -  = sp
a  ; nl sp sp  } nl sp sp  r  e  t  u  r  n sp
a  ; nl  } nl
```

Text file is a sequence of characters

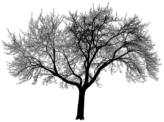
Lexical Analysis Gives Tokens

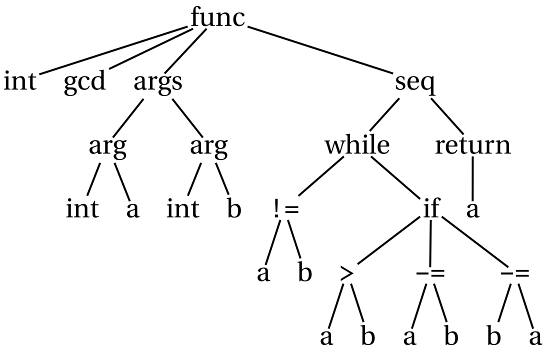
```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

`int` `gcd` `(` `int` `a` `,` `int` `b` `)` `{` `while` `(` `a`
`!=` `b` `)` `{` `if` `(` `a` `>` `b` `)` `a` `-=` `b` `;` `else`
`b` `-=` `a` `;` `}` `return` `a` `;` `}`

A stream of tokens. Whitespace, comments removed.

Parsing Gives an AST

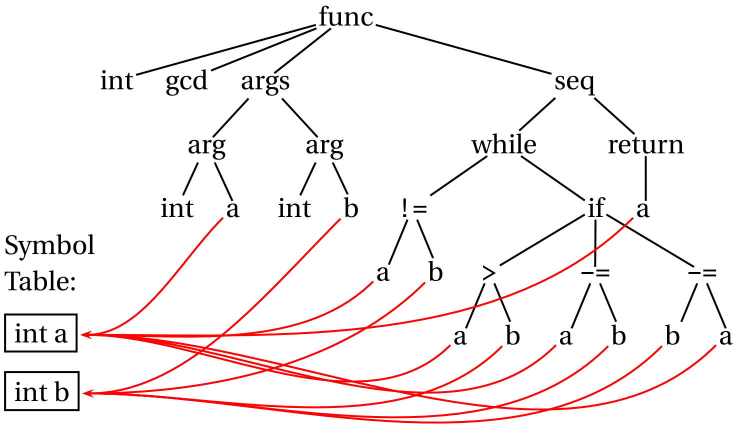





```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Abstract syntax tree built from parsing rules.

Semantic Analysis Resolves Symbols



Types checked; references to symbols resolved

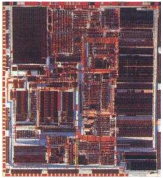
Translation into 3-Address Code

```
L0: sne   $1,  a, b
    seq   $0, $1, 0
    btrue $0, L1     % while (a != b)
    sl    $3,  b, a
    seq   $2, $3, 0
    btrue $2, L4     % if (a < b)
    sub   a,   a, b % a -= b
    jmp   L5
L4: sub   b,   b, a % b -= a
L5: jmp   L0
L1: ret    a
```

```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Idealized assembly language w/ infinite registers

Generation of 80386 Assembly



```
gcd:   pushl %ebp              % Save FP
       movl  %esp,%ebp
       movl  8(%ebp),%eax      % Load a from stack
       movl  12(%ebp),%edx     % Load b from stack
.L8:   cmpl  %edx,%eax
       je    .L3               % while (a != b)
       jle   .L5               % if (a < b)
       subl  %edx,%eax         % a -= b
       jmp   .L8
.L5:   subl  %eax,%edx         % b -= a
       jmp   .L8
.L3:   leave                   % Restore SP, BP
       ret
```

Part II

Scanning

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

Concatenation: Strings from one followed by the other

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

Union: All strings from each language

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

Kleene Closure: Zero or more concatenations

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

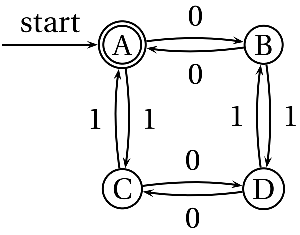
Regular Expressions over an Alphabet Σ

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 $L^i = LL^{i-1}$)

Nondeterministic Finite Automata

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



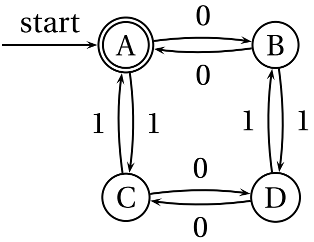
1. Set of states S: {A, B, C, D}
2. Set of input symbols Σ: $\{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \rightarrow 2^S$

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

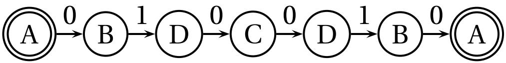
4. Start state s_0 : A
5. Set of accepting states F: {A}

The Language induced by an NFA

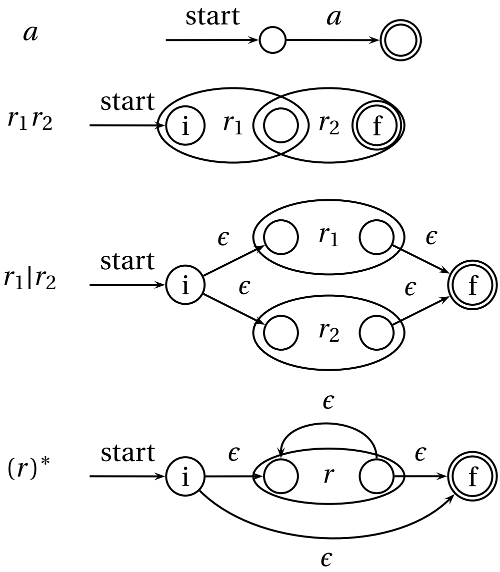
An NFA accepts an input string *x* iff there is a path from the start state to an accepting state that "spells out" *x*.



Show that the string "010010" is accepted.

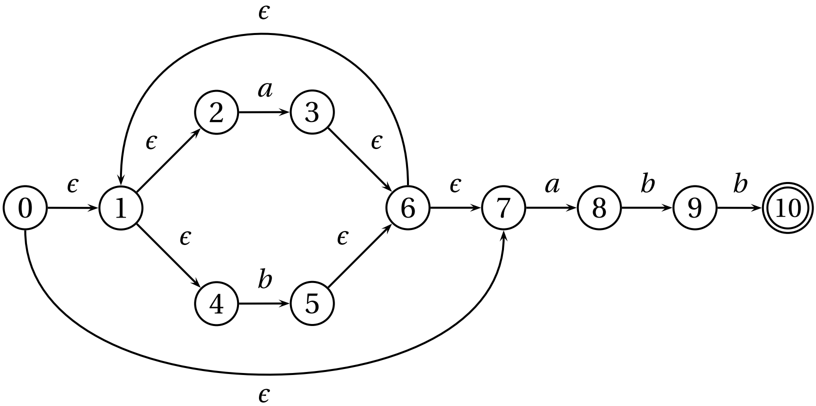


Translating REs into NFAs

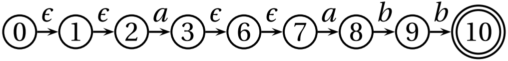


Translating REs into NFAs

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



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



Simulating NFAs

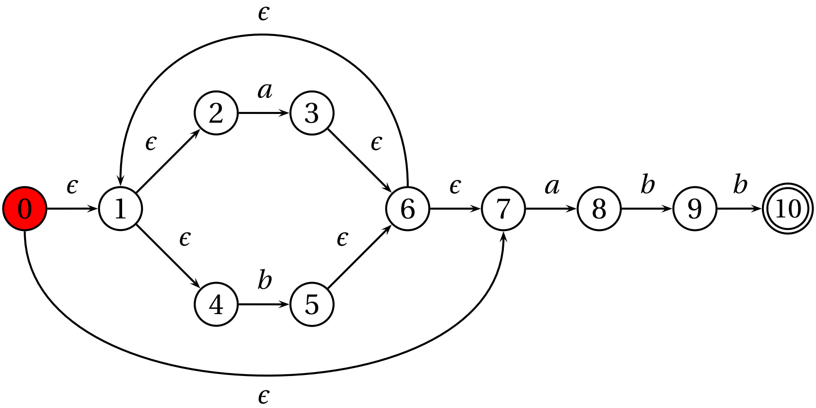
Problem: you must follow the “right” arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

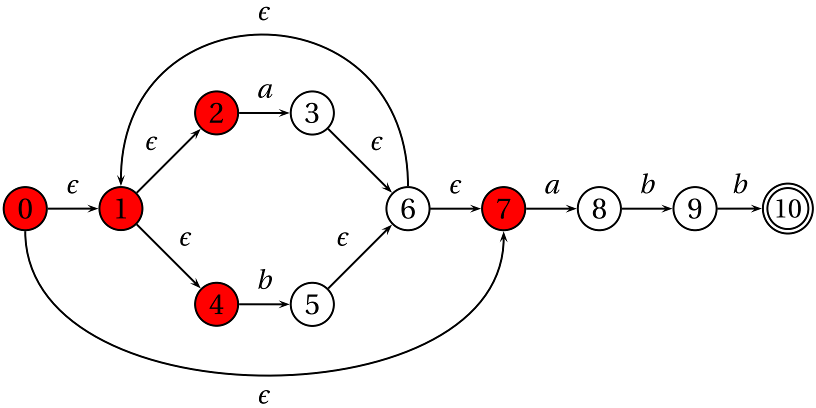
“Two-stack” NFA simulation algorithm:

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

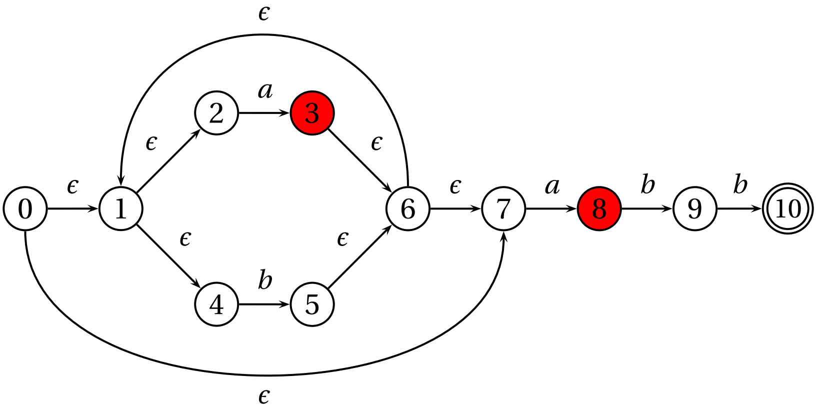
Simulating an NFA: $\cdot aabb$, Start



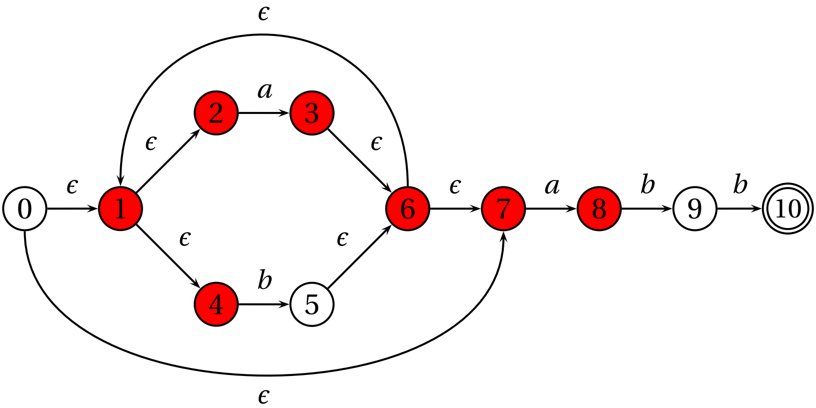
Simulating an NFA: $\cdot aabb$, ϵ-closure



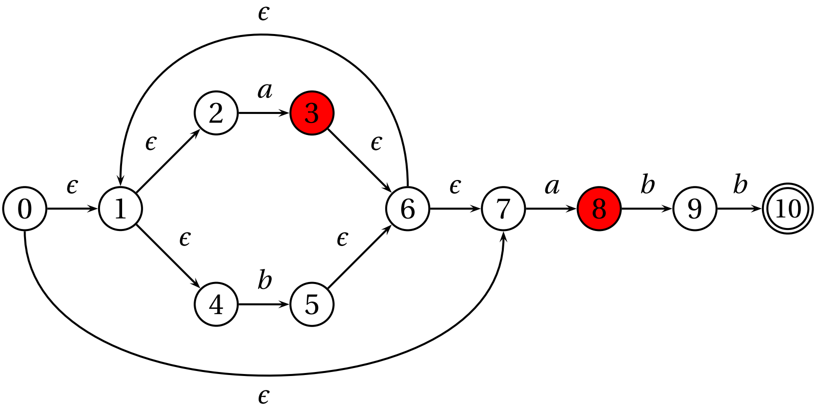
Simulating an NFA: $a{\cdot}abb$



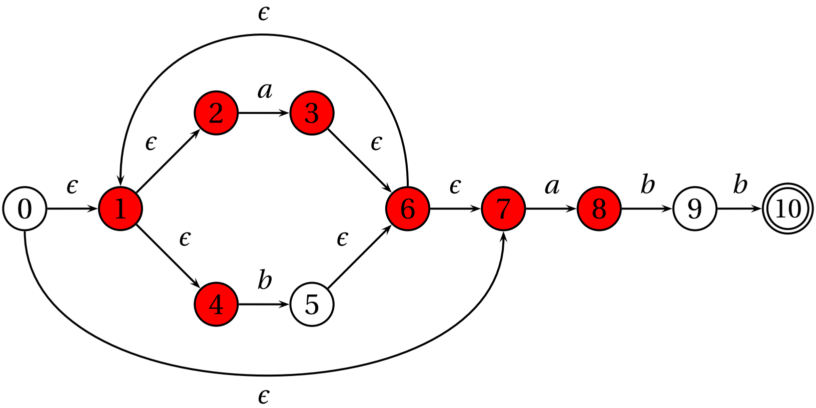
Simulating an NFA: $a \cdot abb$, ϵ-closure



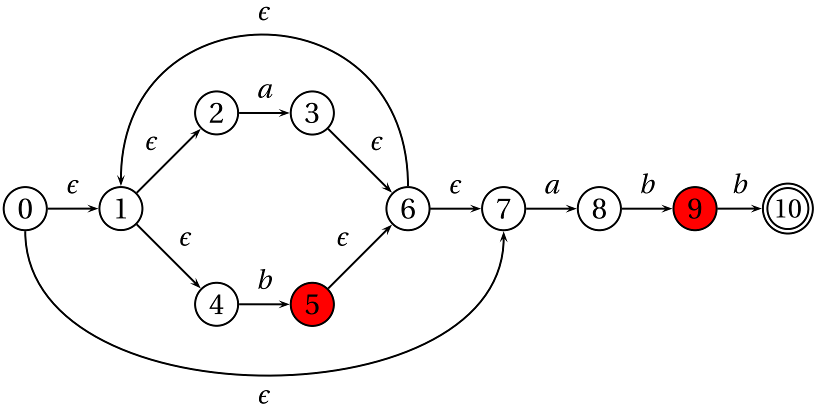
Simulating an NFA: $aa{\cdot}bb$



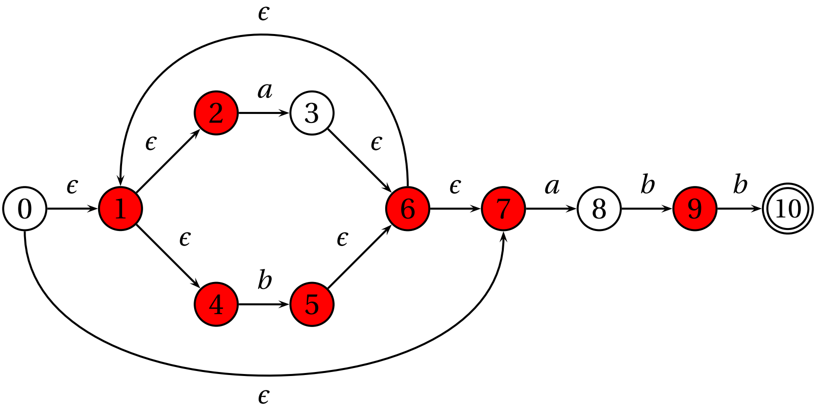
Simulating an NFA: $aa{\cdot}bb$, ϵ-closure



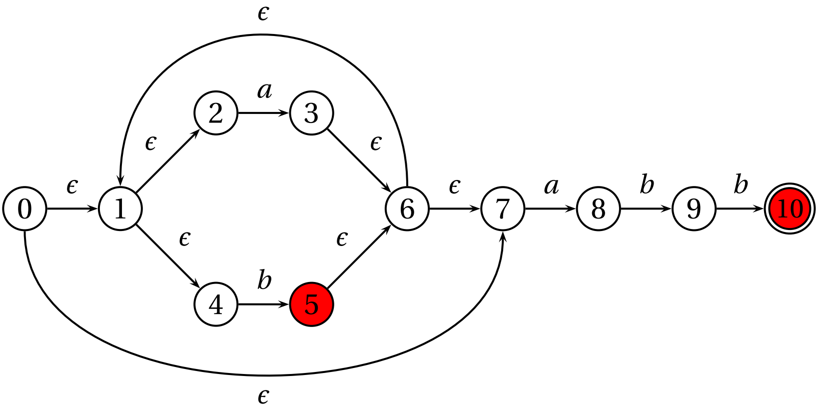
Simulating an NFA: *aab·b*



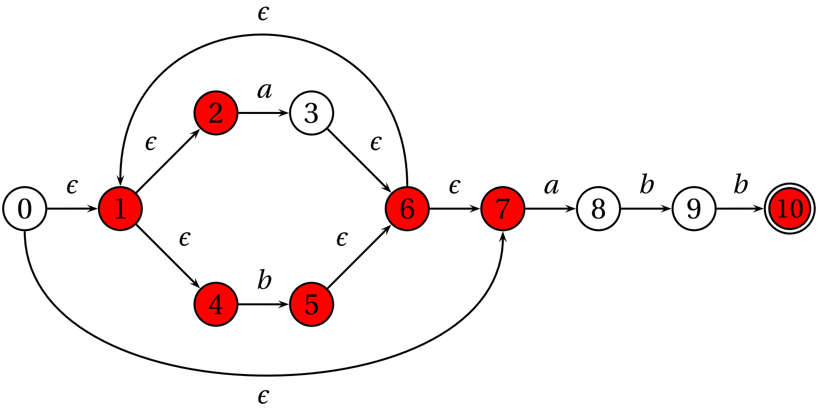
Simulating an NFA: $aab \cdot b$, ϵ-closure



Simulating an NFA: $aabb\cdot$



Simulating an NFA: $aabb\cdot$, Done



Deterministic Finite Automata

Restricted form of NFAs:

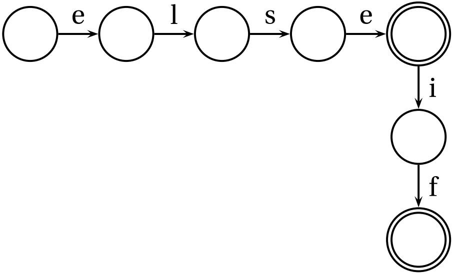
- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

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.

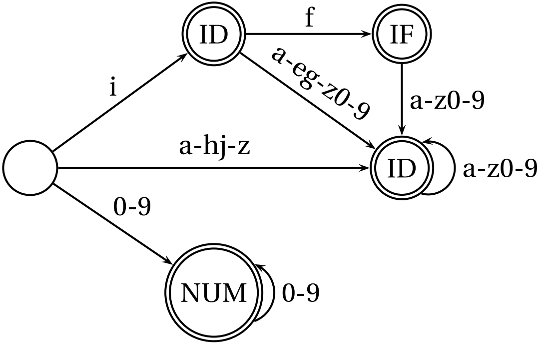
Deterministic Finite Automata

```
{
   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) }
```



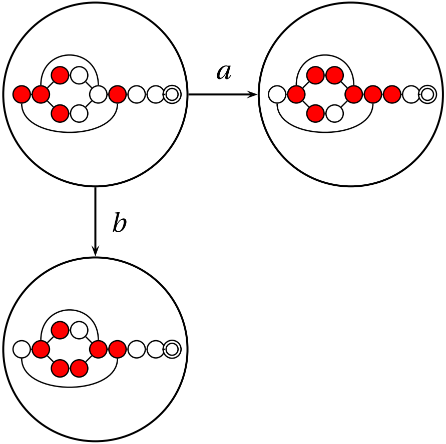
Building a DFA from an NFA

Subset construction algorithm

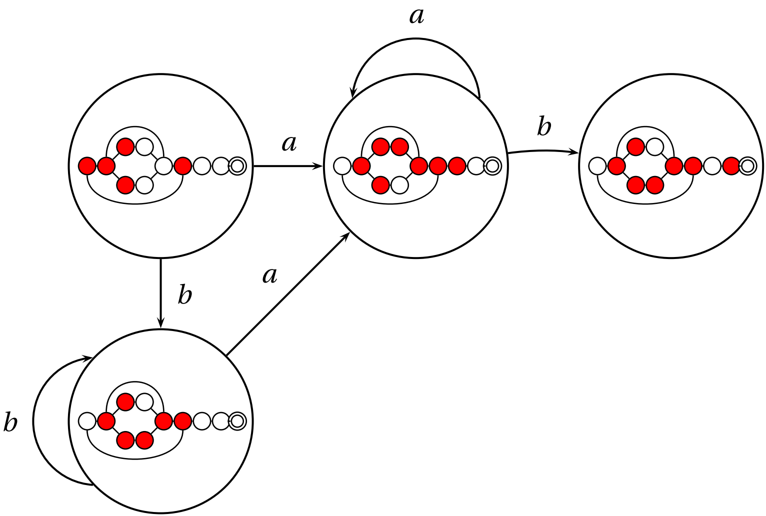
Simulate the NFA for all possible inputs and track the states that appear.

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

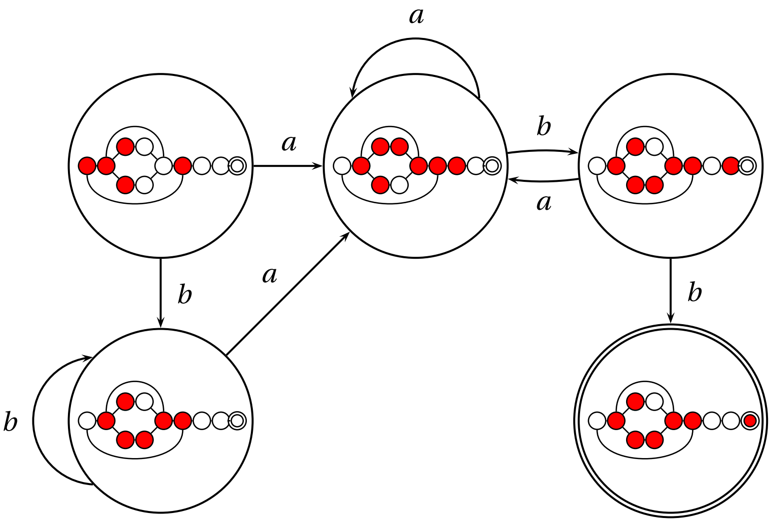
Subset construction for $(a|b)^* abb$ (1)



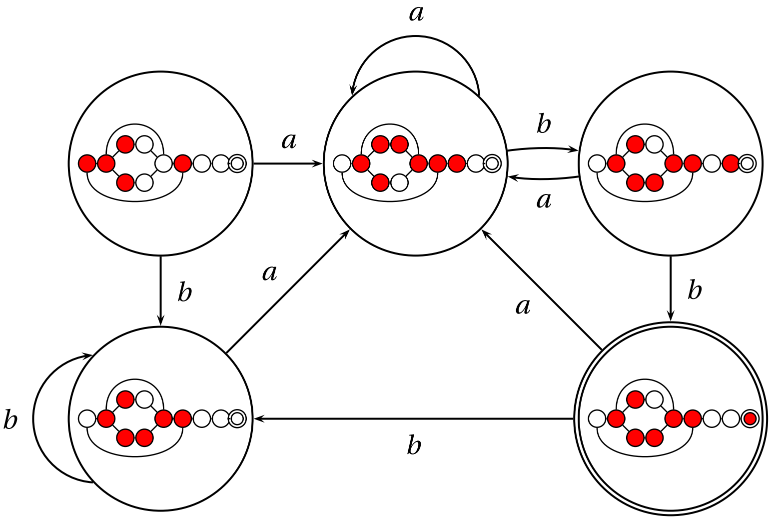
Subset construction for $(a|b)^* abb$ (2)



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



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



Part III

Parsing

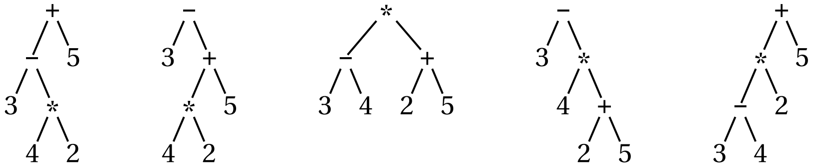
Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

`3 - 4 * 2 + 5`

with the grammar

$$e \rightarrow e + e \mid e - e \mid e * e \mid e \,/\, e \mid N$$



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          {}
;
```

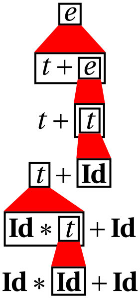
This is left-associative.

No shift/reduce conflicts.

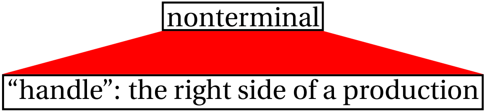
Rightmost Derivation

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

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



At each step, expand the rightmost nonterminal.

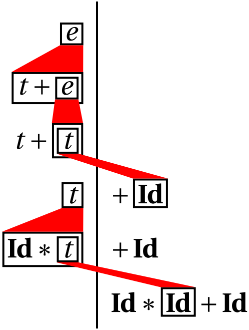


Fun and interesting fact: there is exactly one rightmost expansion if the grammar is unambigious.

Rightmost Derivation

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

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

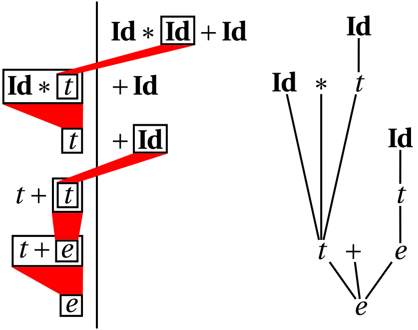
$1: \quad e \rightarrow t + e$

$2: \quad e \rightarrow t$

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

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

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



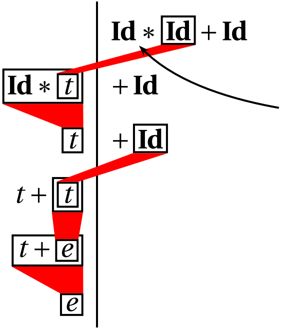
Beginning to look like a parsing algorithm: start with terminals and reduce them to the starting nonterminal.

Reductions build the parse tree starting at the leaves.

Reverse Rightmost Derivation

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

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.

$$
\begin{aligned}
e &\to \cdot t + e \\
e &\to \cdot t \\
t &\to \cdot \mathbf{Id} * t \\
t &\to \cdot \mathbf{Id}
\end{aligned}
$$

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:

$e \to t \cdot + e$ when e was expanded to $t + e$

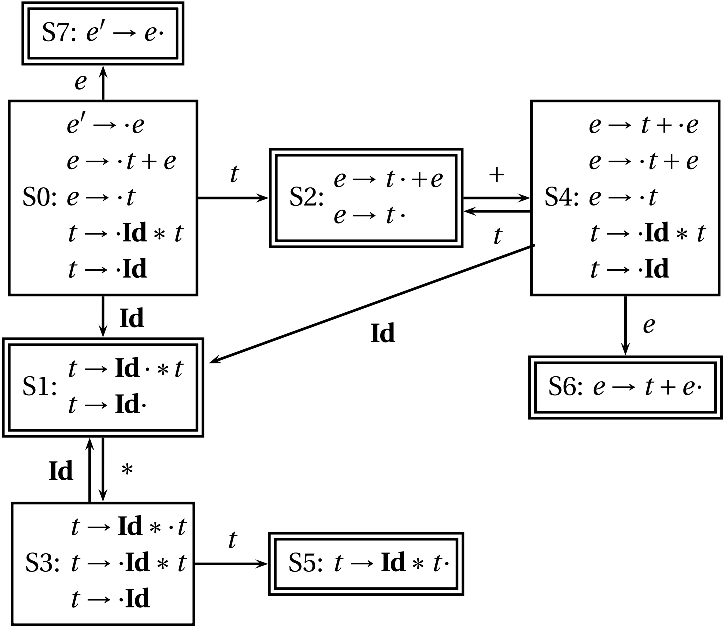
$e \to t \cdot$ when e was expanded to just t; t is a handle

The expanded-t case also gives two possible positions:

$t \to \mathbf{Id} \cdot * t$ when t was expanded to $\mathbf{Id} + t$

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

Constructing the LR(0) Automaton



Shift-reduce Parsing

1 : $e \rightarrow t + e$

2 : $e \rightarrow t$

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

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

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

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

$2: \quad e \rightarrow t$

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

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

	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

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 : $t \rightarrow \mathbf{Id} \, * \, t$

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

	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
$0 \; \mathbf{Id}_1$	* **Id** + **Id** \$	shift, goto 3
$0 \; \mathbf{Id}_1 \; *_3$	**Id** + **Id** \$	shift, goto 1
$0 \; \mathbf{Id}_1 \; *_3 \; \mathbf{Id}_1$	+ **Id** \$	reduce w/ 4

Action is "reduce with rule 4 ($t \rightarrow \mathbf{Id}$)." 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
$0 \; \mathbf{Id}_1 \; *_3 \; t_5$	+ **Id** \$	

LR Parsing

1 : $e \rightarrow t + e$

2 : $e \rightarrow t$

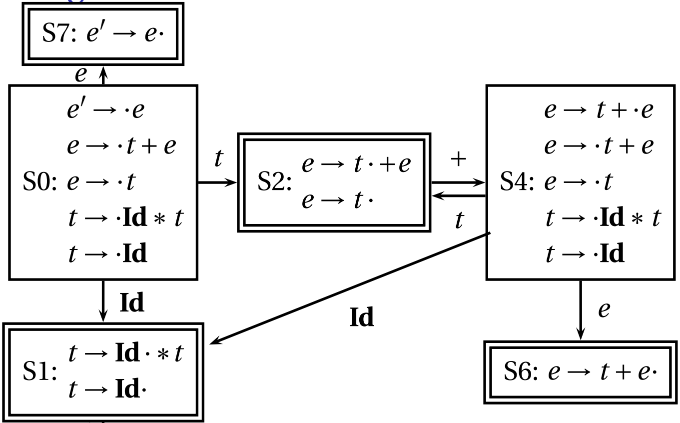
3 : $t \rightarrow \mathbf{Id} * t$

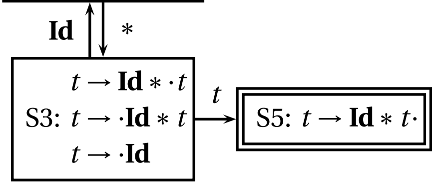
4 : $t \rightarrow \mathbf{Id}$

	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



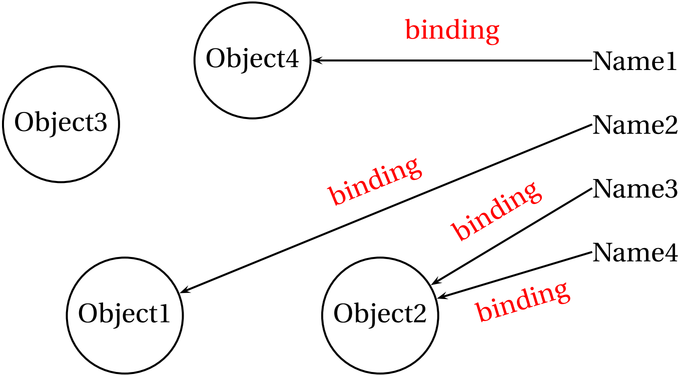


	action				goto	
	Id	+	*	$	*e*	*t*
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

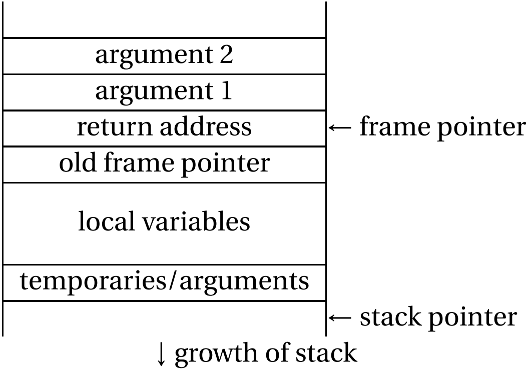
Part IV

Name, Scope, and Bindings

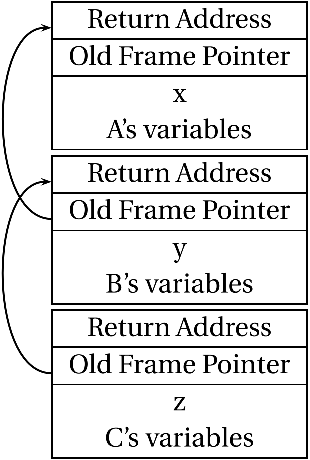
Names, Objects, and Bindings



Activation Records



Activation Records



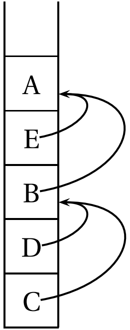

```
int A() {
  int x;
  B();
}

int B() {
  int y;
  C();
}

int C() {
  int z;
}
```

Nested Subroutines in Pascal

```
procedure mergesort;
var N : integer;

  procedure split;
  var I : integer;
  begin
  ...
  end

  procedure merge;
  var J : integer;
  begin
  ...
  end

begin
...
end
```

Nested Subroutines in Pascal

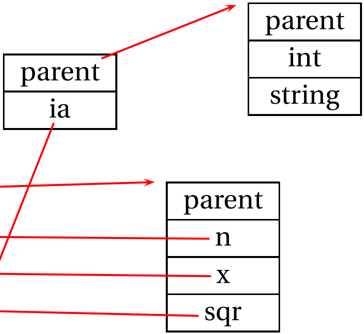
```
procedure A;
  procedure B;
    procedure C;
    begin
    ...
    end

    procedure D;
    begin
    C
    end

  begin
  D
  end

  procedure E;
  begin
  B
  end

begin
E
end
```



Static vs. Dynamic Scope

```
program example;
var a : integer; (* Outer a *)

  procedure seta;
  begin
    a := 1  (* Which a does this change? *)
  end

  procedure locala;
  var a : integer; (* Inner a *)
  begin
    seta
  end

begin
  a := 2;
  if (readln() = 'b')
    locala
  else
    seta;
  writeln(a)
end
```

Symbol Tables in Tiger




```
let
  var n := 8
  var x := 3
  function sqr(a:int)
      = a * a
  type ia = array of int
in
  n := sqr(x)
end
```