Review for the Final

Stephen A. Edwards

Columbia University

Fall 2008

The Midterm

70 minutes

4–5 problems

Closed book

One single-sided 8.5 × 11 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

Regular Expressions

The Subset Construction Algorithm

Parsing

Bottom-up Parsing

Name, Scope, and Bindings

Static Semantic Analysis

Intermediate Representations

Separate Compilation and Linking

The Lambda Calculus

Logic Programming (Prolog)

Concurrency

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

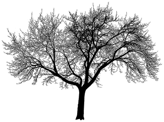
Lexical Analysis Gives Tokens

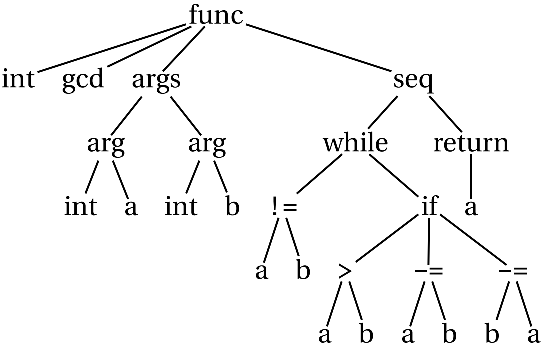
```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

`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

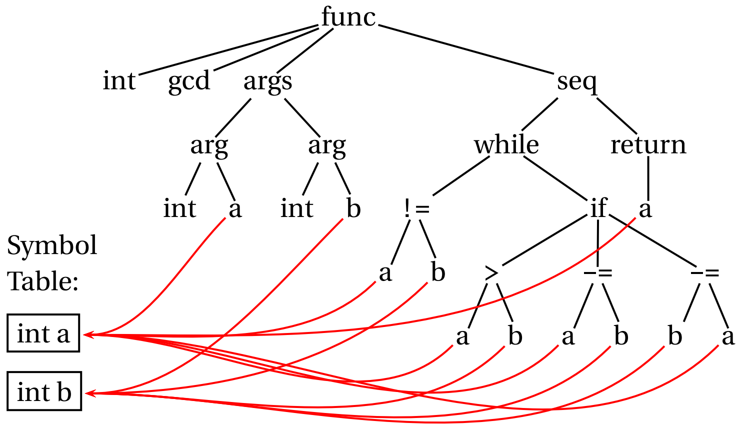





```
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

Abstract syntax tree built from parsing rules.

Semantic Analysis Resolves Symbols



Types checked; references to symbols resolved

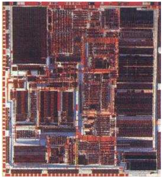
Translation into 3-Address Code

```
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: $\{\, 0, 1 \,\}$, $\{\, A, B, C, \ldots, Z \,\}$, 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

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

Part III

Regular Expressions

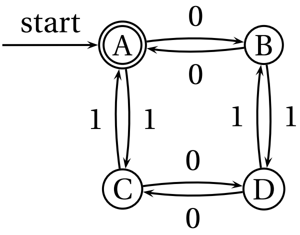
Regular Expressions over an Alphabet Σ

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"



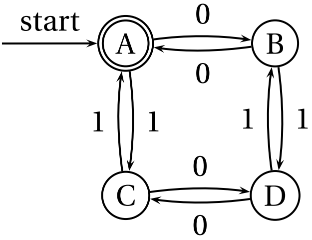
1. Set of states S: $\left\{ A, B, C, D \right\}$
2. Set of input symbols Σ: $\{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \to 2^S$

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

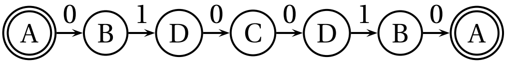
4. Start state s_0 : A
5. Set of accepting states F: $\left\{ A \right\}$

The Language induced by an NFA

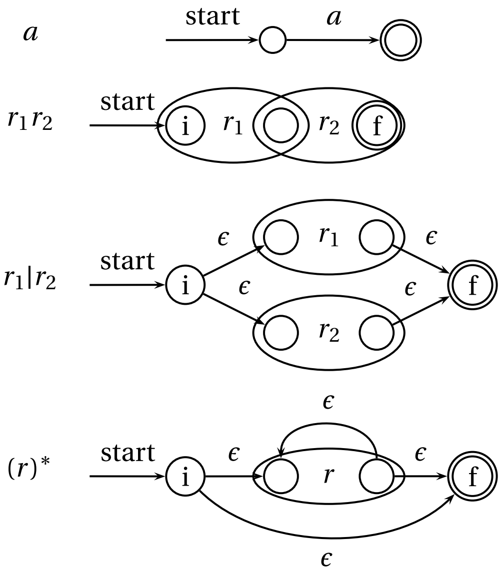
An NFA accepts an input string x iff there is a path from the start state to an accepting state that "spells out" x.



Show that the string "010010" is accepted.

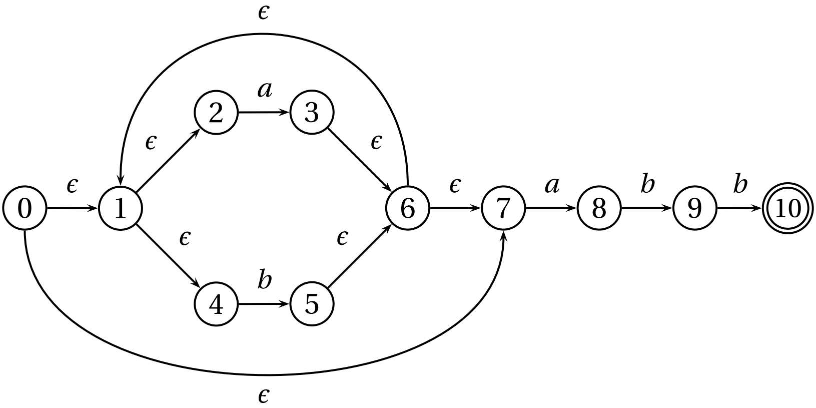


Translating REs into NFAs



Translating REs into NFAs

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



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

$$0 \xrightarrow{\epsilon} 1 \xrightarrow{\epsilon} 2 \xrightarrow{a} 3 \xrightarrow{\epsilon} 6 \xrightarrow{\epsilon} 7 \xrightarrow{a} 8 \xrightarrow{b} 9 \xrightarrow{b} 10$$

Simulating NFAs

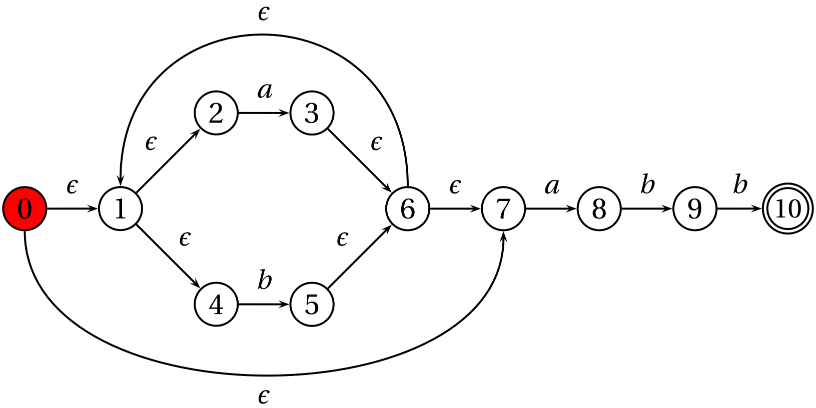
Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

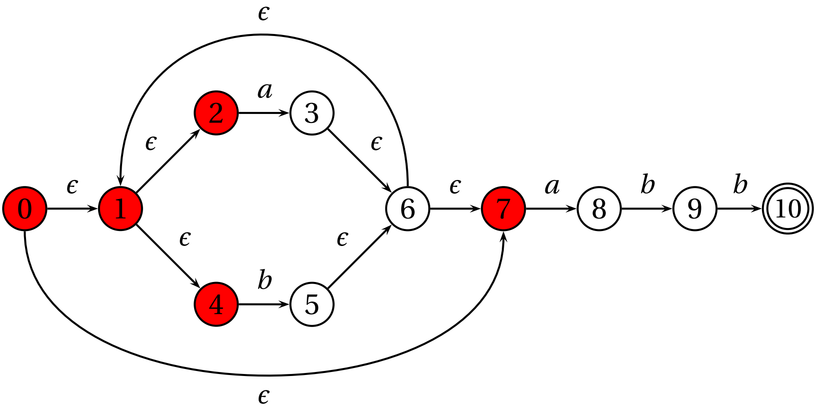
"Two-stack" NFA simulation algorithm:

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

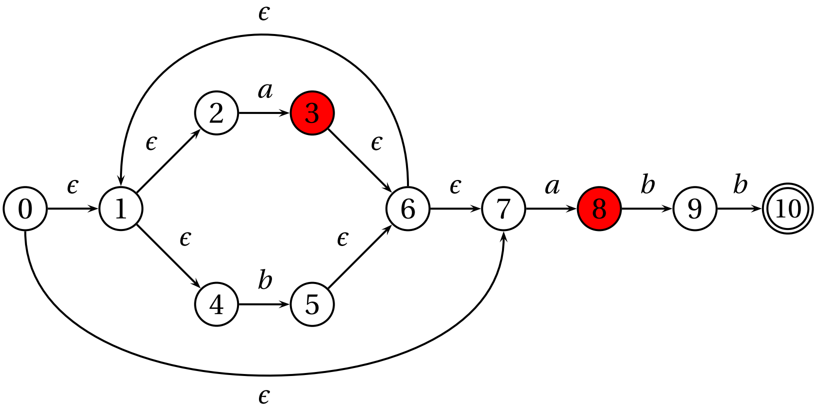
Simulating an NFA: $\cdot aabb$, Start



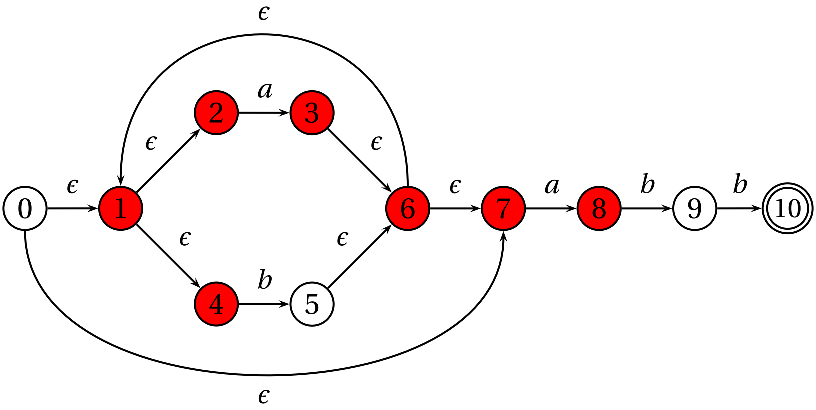
Simulating an NFA: $\cdot aabb$, ϵ-closure



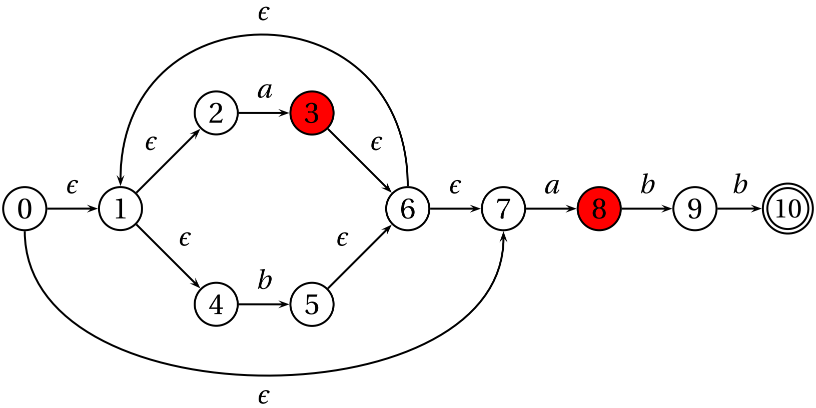
Simulating an NFA: $a{\cdot}abb$



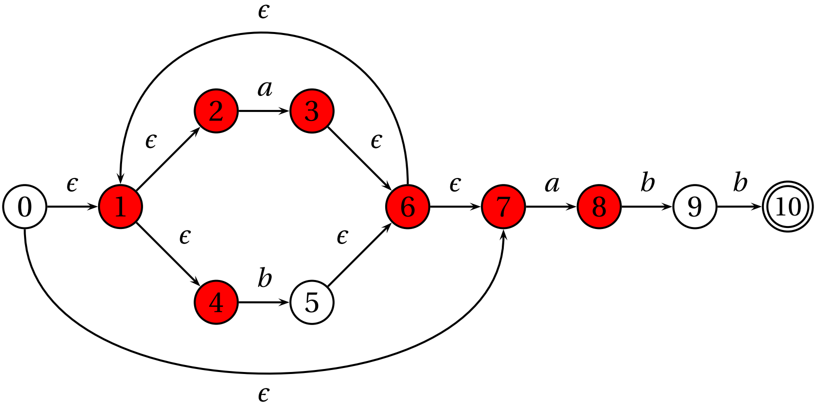
Simulating an NFA: $a \cdot abb$, ϵ-closure



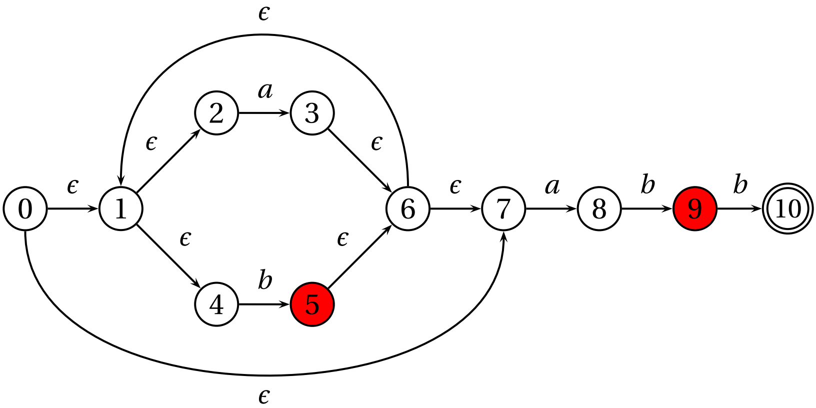
Simulating an NFA: $aa{\cdot}bb$



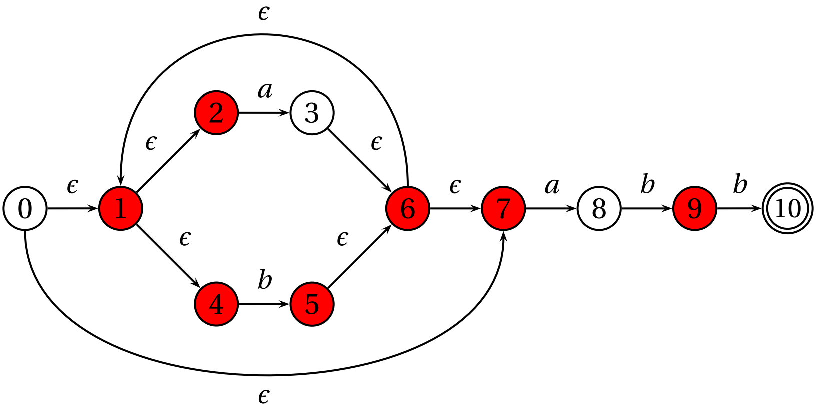
Simulating an NFA: $aa{\cdot}bb$, ϵ-closure



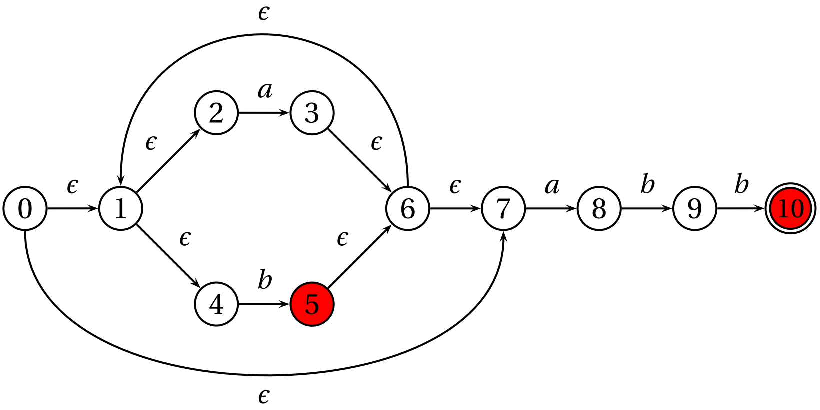
Simulating an NFA: $aab{\cdot}b$



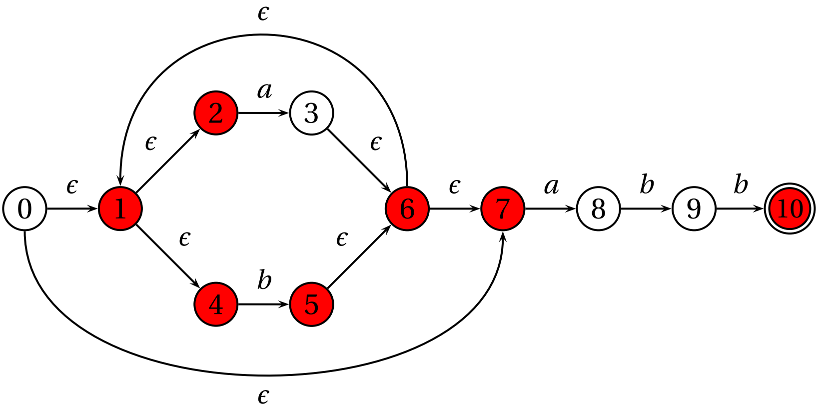
Simulating an NFA: $aab{\cdot}b$, ϵ-closure



Simulating an NFA: $aabb\cdot$



Simulating an NFA: $aabb\cdot$, Done



Deterministic Finite Automata

Restricted form of NFAs:

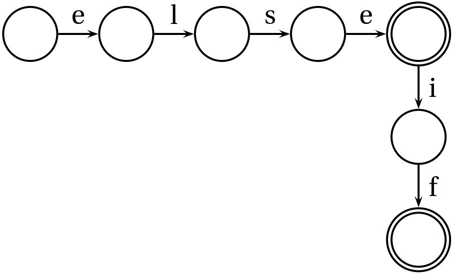
- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

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.

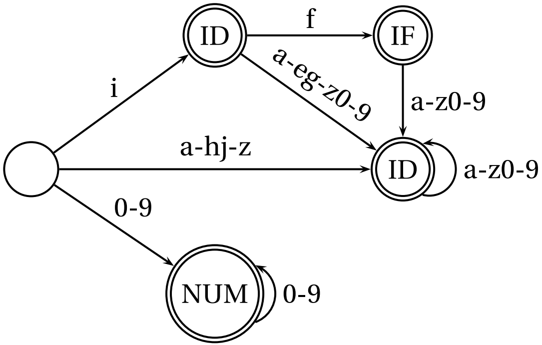
Deterministic Finite Automata

```
{
   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) }
```



Part IV

The Subset Construction Algorithm

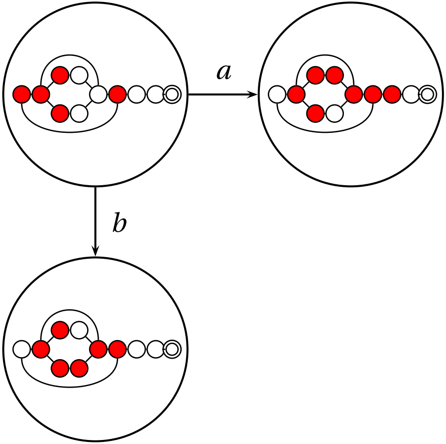
Building a DFA from an NFA

Subset construction algorithm

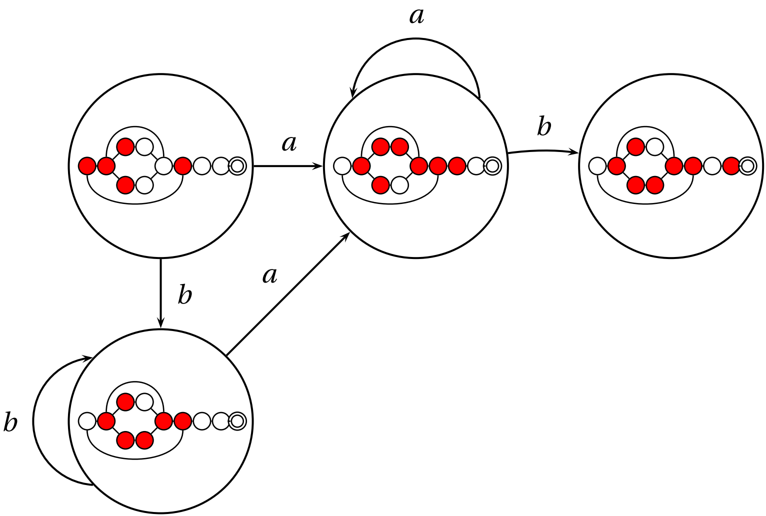
Simulate the NFA for all possible inputs and track the states that appear.

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

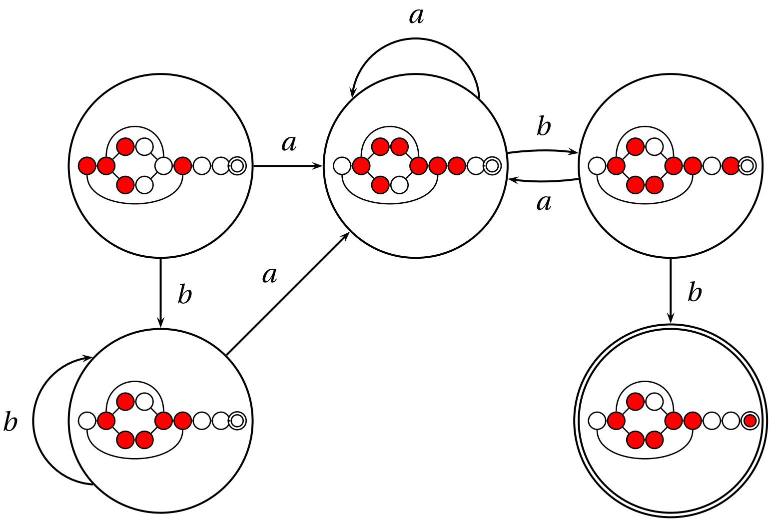
Subset construction for $(a|b)^*abb$ (1)



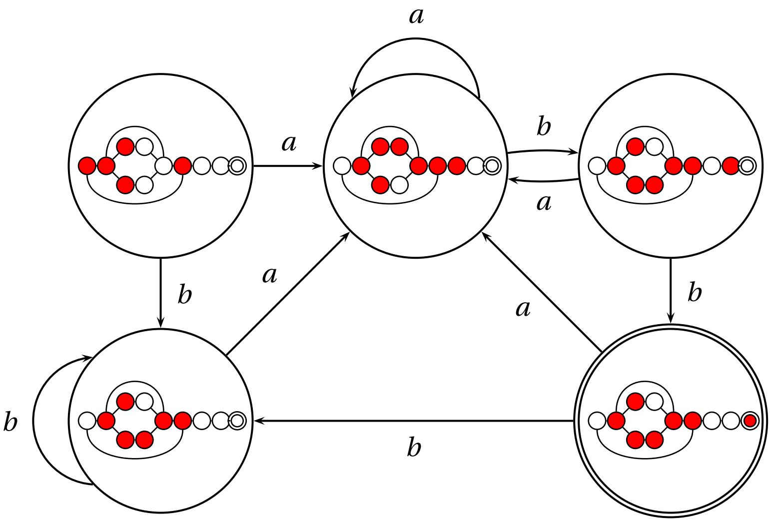
Subset construction for $(a|b)^* abb$ (2)



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



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



Part V

Parsing

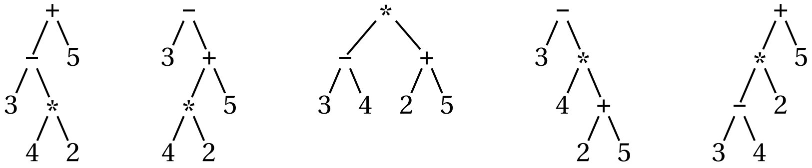
Ambiguous Grammars

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.

Part VI

Bottom-Up Parsing

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 $\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$:

$$\boxed{e}$$
$$\boxed{t + \boxed{e}}$$
$$t + \boxed{\boxed{t}}$$
$$\boxed{t} + \boxed{\mathbf{Id}}$$
$$\boxed{\mathbf{Id} * \boxed{t}} + \mathbf{Id}$$
$$\mathbf{Id} * \boxed{\mathbf{Id}} + \mathbf{Id}$$

At each step, expand the rightmost nonterminal.

nonterminal

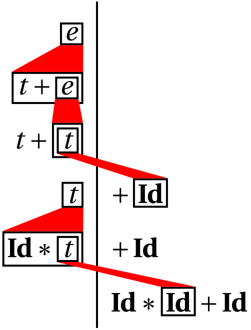
"handle": the right side of a production

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

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 $\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$:



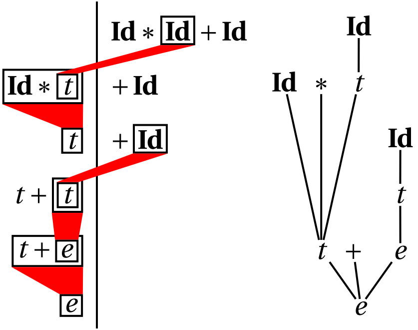
Tokens on the right are all terminals.

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

Reverse Rightmost Derivation

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

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



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 $\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$:

$$
\begin{array}{r|l}
 & \mathbf{Id} * \boxed{\mathbf{Id}} + \mathbf{Id} \\
\boxed{\mathbf{Id} * \boxed{t}} & + \mathbf{Id} \\
\boxed{t} & + \boxed{\mathbf{Id}} \\
t + \boxed{t} & \\
\boxed{t + \boxed{e}} & \\
\boxed{e} &
\end{array}
$$

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.

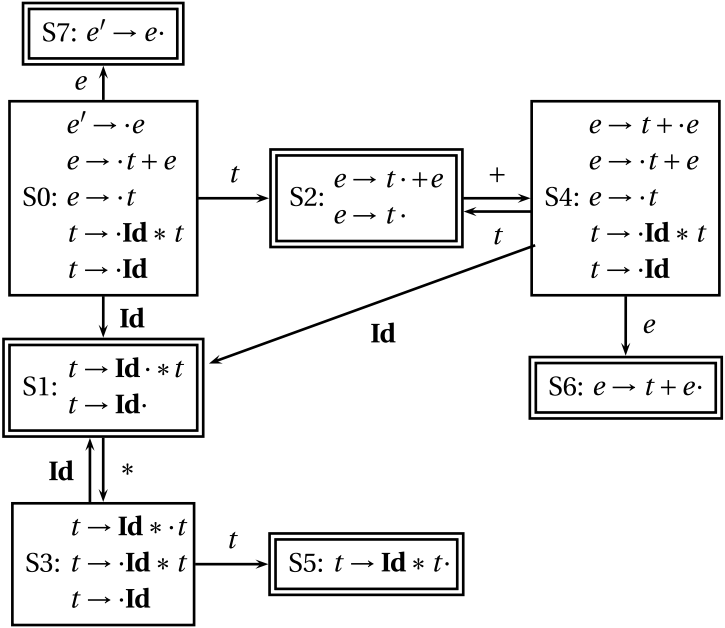
Consider the expansion of e first. This gives two possible positions:

$e \to t \cdot + e$	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

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

$2: \quad e \rightarrow t$

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

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

stack	input	action
	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id}$	$* \mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id} *$	$\mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id} * \boxed{\mathbf{Id}}$	$+ \mathbf{Id}$	reduce (4)
$\boxed{\mathbf{Id} * t}$	$+ \mathbf{Id}$	reduce (3)
t	$+ \mathbf{Id}$	shift
$t+$	$\mathbf{Id}$	shift
$t + \boxed{\mathbf{Id}}$		reduce (4)
$t + \boxed{t}$		reduce (2)
$\boxed{t + e}$		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
$\boxed{0}$	**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

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

	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
$\boxed{0}$	**Id** * **Id** + **Id** \$	shift, goto 1
$\boxed{0}\ \boxed{\substack{\mathbf{Id}\\1}}$	* **Id** + **Id** \$	shift, goto 3
$\boxed{0}\ \boxed{\substack{\mathbf{Id}\\1}}\ \boxed{\substack{*\\3}}$	**Id** + **Id** \$	shift, goto 1
$\boxed{0}\ \boxed{\substack{\mathbf{Id}\\1}}\ \boxed{\substack{*\\3}}\ \boxed{\substack{\mathbf{Id}\\1}}$	+ **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
$\boxed{0}\ \boxed{\substack{\mathbf{Id}\\1}}\ \boxed{\substack{*\\3}}\ \boxed{\substack{t\\5}}$	+ **Id** \$	

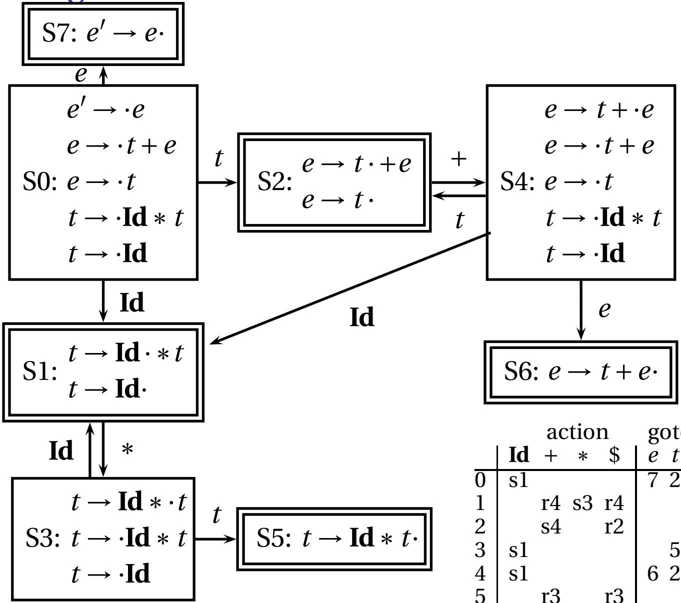
LR Parsing

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

	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
$\boxed{0}$	**Id** * **Id** + **Id** $	shift, goto 1
$\boxed{0}\ \boxed{\substack{\textbf{Id}\\1}}$	* **Id** + **Id** $	shift, goto 3
$\boxed{0}\ \boxed{\substack{\textbf{Id}\\1}}\ \boxed{\substack{*\\3}}$	**Id** + **Id** $	shift, goto 1
$\boxed{0}\ \boxed{\substack{\textbf{Id}\\1}}\ \boxed{\substack{*\\3}}\ \boxed{\substack{\textbf{Id}\\1}}$	+ **Id** $	reduce w/ 4
$\boxed{0}\ \boxed{\substack{\textbf{Id}\\1}}\ \boxed{\substack{*\\3}}\ \boxed{\substack{t\\5}}$	+ **Id** $	reduce w/ 3
$\boxed{0}\ \boxed{\substack{t\\2}}$	+ **Id** $	shift, goto 4
$\boxed{0}\ \boxed{\substack{t\\2}}\ \boxed{\substack{+\\4}}$	**Id** $	shift, goto 1
$\boxed{0}\ \boxed{\substack{t\\2}}\ \boxed{\substack{+\\4}}\ \boxed{\substack{\textbf{Id}\\1}}$	$	reduce w/ 4
$\boxed{0}\ \boxed{\substack{t\\2}}\ \boxed{\substack{+\\4}}\ \boxed{\substack{t\\2}}$	$	reduce w/ 2
$\boxed{0}\ \boxed{\substack{t\\2}}\ \boxed{\substack{+\\4}}\ \boxed{\substack{e\\6}}$	$	reduce w/ 1
$\boxed{0}\ \boxed{\substack{e\\7}}$	$	accept

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

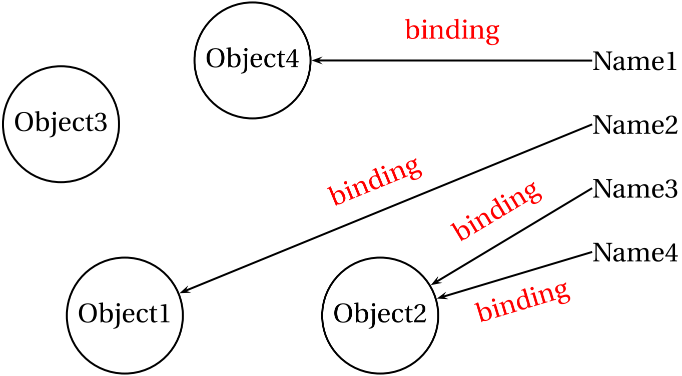


	action				goto	
	Id	$+$	$*$	\$	e	t
0	s1				7	2
1		r4	s3	r4		
2		s4		r2		
3	s1					5
4	s1				6	2
5		r3		r3		
6				r1		
7				✓		

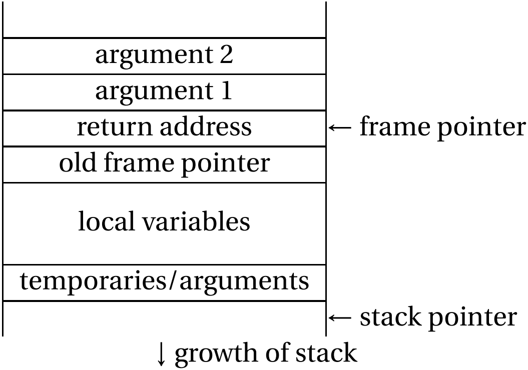
Part VII

Name, Scope, and Bindings

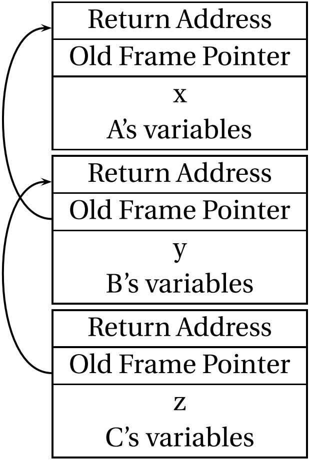
Names, Objects, and Bindings



Activation Records



Activation Records



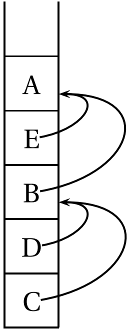

```
int A() {
  int x;
  B();
}

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

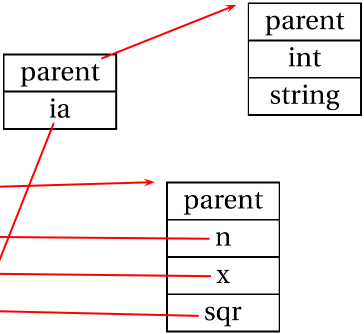
```
procedure A;
  procedure B;
    procedure C;
    begin
    ...
    end

    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
```

Part VIII

Static Semantic Analysis

Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```
if i 3 "This"                /* valid */
#a1123                       /* invalid */
```

Syntactic analysis: Makes sure tokens appear in correct order

```
for i := 1 to 5 do 1 + break /* valid */
if i 3                       /* invalid */
```

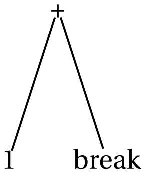
Semantic analysis: Makes sure program is consistent

```
let v := 3 in v + 8 end      /* valid */
let v := "f" in v(3) + v end /* invalid */
```

Static Semantic Analysis

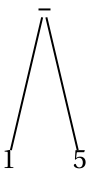
Basic paradigm: recursively check AST nodes.

```
1 + break
```




```
check(+)
  check(1) = int
  check(break) = void
  FAIL: int != void
```

```
1 - 5
```




```
check(-)
  check(1) = int
  check(5) = int
  Types match, return int
```

Ask yourself: at a particular node type, what must be true?

Implementing multi-way branches

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

Obvious way:

```
if (s == 1) { one(); }
else if (s == 2) { two(); }
else if (s == 3) { three(); }
else if (s == 4) { four(); }
```

Reasonable, but we can sometimes do better.

Implementing multi-way branches

If the cases are *dense*, a branch table is more efficient:

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

A branch table written using a GCC extension:

```
  /* Array of addresses of labels */
  static void *l[] = { &&L1, &&L2, &&L3, &&L4 };

  if (s >= 1 && s <= 4)
     goto *l[s-1];
  goto Break;
L1: one(); goto Break;
L2: two(); goto Break;
L3: three(); goto Break;
L4: four(); goto Break;
Break:
```

Applicative- and Normal-Order Evaluation

```
int p(int i) {
   printf("%d ", i);
   return i;
}

void q(int a, int b, int c)
{
  int total = a;
  printf("%d ", b);
  total += c;
}
```

What is printed by

```
q( p(1), 2, p(3) );
```

?

Applicative- and Normal-Order Evaluation

```
int p(int i) { printf("%d ", i); return i; }

void q(int a, int b, int c)
{
  int total = a;
  printf("%d ", b);
  total += c;
}

q( p(1), 2, p(3) );
```

Applicative: arguments evaluated before function is called.

Result: 1 3 2

Normal: arguments evaluated when used.

Result: 1 2 3

Applicative- vs. and Normal-Order

Most languages use applicative order.

Macro-like languages often use normal order.

```
#define p(x) (printf("%d ",x), x)

#define q(a,b,c) total = (a), \
   printf("%d ", (b)), \
   total += (c)

q( p(1), 2, p(3) );
```

Prints 1 2 3.

Some functional languages also use normal order evaluation to avoid doing work. “Lazy Evaluation”

Nondeterminism

Nondeterminism is not the same as random:

Compiler usually chooses an order when generating code.

Optimization, exact expressions, or run-time values may affect behavior.

Bottom line: don't know what code will do, but often know set of possibilities.

```
int p(int i) { printf("%d ", i); return i; }
int q(int a, int b, int c) {}
q( p(1), p(2), p(3) );
```

Will *not* print 5 6 7. It will print one of

1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1

Layout of Records and Unions

Modern memory systems read data in 32-, 64-, or 128-bit chunks:

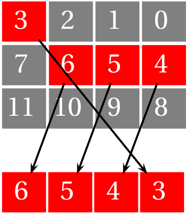
3	2	1	0
7	6	5	4
11	10	9	8

Reading an aligned 32-bit value is fast: a single operation.

3	2	1	0
7	6	5	4
11	10	9	8

Layout of Records and Unions

Slower to read an unaligned value: two reads plus shift.



SPARC prohibits unaligned accesses.

MIPS has special unaligned load/store instructions.

x86, 68k run more slowly with unaligned accesses.

Layout of Records and Unions

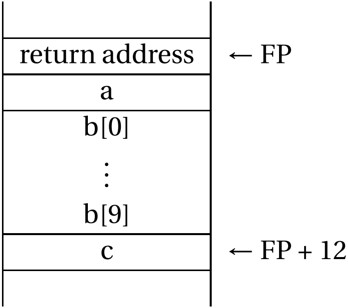
Most languages "pad" the layout of records to ensure alignment restrictions.

```
struct padded {
  int x;    /* 4 bytes */
  char z;   /* 1 byte */
  short y;  /* 2 bytes */
  char w;   /* 1 byte */
};
```

Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

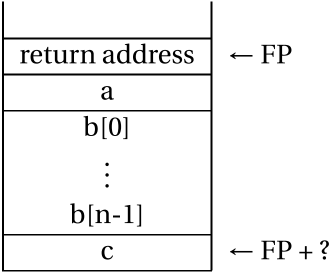
```
void foo()
{
  int a;
  int b[10];
  int c;
}
```



Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

```
void foo(int n)
{
  int a;
  int b[n];
  int c;
}
```

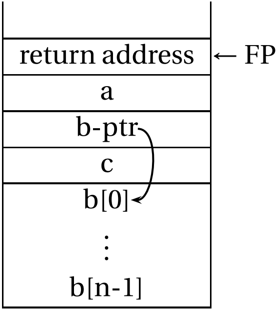


Doesn't work: generated code expects a fixed offset for c. Even worse for multi-dimensional arrays.

Allocating Variable-Sized Arrays

As always:
add a level of indirection

```
void foo(int n)
{
  int a;
  int b[n];
  int c;
}
```



Variables remain constant offset from frame pointer.

Part IX

Intermediate Representations/Formats

Stack-Based IR: Java Bytecode

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

```
# javap -c Gcd

Method int gcd(int, int)
   0 goto 19

   3 iload_1          // Push a
   4 iload_2          // Push b
   5 if_icmple 15     // if a <= b goto 15

   8 iload_1          // Push a
   9 iload_2          // Push b
  10 isub             // a - b
  11 istore_1         // Store new a
  12 goto 19

  15 iload_2          // Push b
  16 iload_1          // Push a
  17 isub             // b - a
  18 istore_2         // Store new b

  19 iload_1          // Push a
  20 iload_2          // Push b
  21 if_icmpne 3      // if a != b goto 3

  24 iload_1          // Push a
  25 ireturn          // Return a
```

Register-Based IR: Mach SUIF

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

```
gcd:
gcd._gcdTmp0:
  sne   $vr1.s32 <- gcd.a,gcd.b
  seq   $vr0.s32 <- $vr1.s32,0
  btrue $vr0.s32,gcd._gcdTmp1    // if !(a != b) goto Tmp1

  sl    $vr3.s32 <- gcd.b,gcd.a
  seq   $vr2.s32 <- $vr3.s32,0
  btrue $vr2.s32,gcd._gcdTmp4    // if !(a < b) goto Tmp4

  mrk   2, 4     // Line number 4
  sub   $vr4.s32 <- gcd.a,gcd.b
  mov   gcd._gcdTmp2 <- $vr4.s32
  mov   gcd.a <- gcd._gcdTmp2    // a = a - b
  jmp   gcd._gcdTmp5
gcd._gcdTmp4:
  mrk   2, 6
  sub   $vr5.s32 <- gcd.b,gcd.a
  mov   gcd._gcdTmp3 <- $vr5.s32
  mov   gcd.b <- gcd._gcdTmp3    // b = b - a
gcd._gcdTmp5:
  jmp   gcd._gcdTmp0

gcd._gcdTmp1:
  mrk   2, 8
  ret   gcd.a    // Return a
```

Basic Blocks


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

lower →

```
A: sne t, a, b
   bz  E, t
   slt t, a, b
   bnz B, t
   sub b, b, a
   jmp C
B: sub a, a, b
C: jmp A
E: ret a
```

split →

```
A: sne t, a, b
   bz  E, t

   slt t, a, b
   bnz B, t

   sub b, b, a
   jmp C

B: sub a, a, b

C: jmp A

E: ret a
```

The statements in a basic block all run if the first one does.

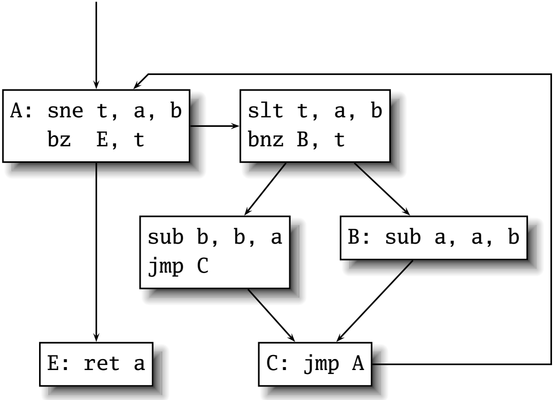
Starts with a statement following a conditional branch or is a branch target.

Usually ends with a control-transfer statement.

Control-Flow Graphs

A CFG illustrates the flow of control among basic blocks.

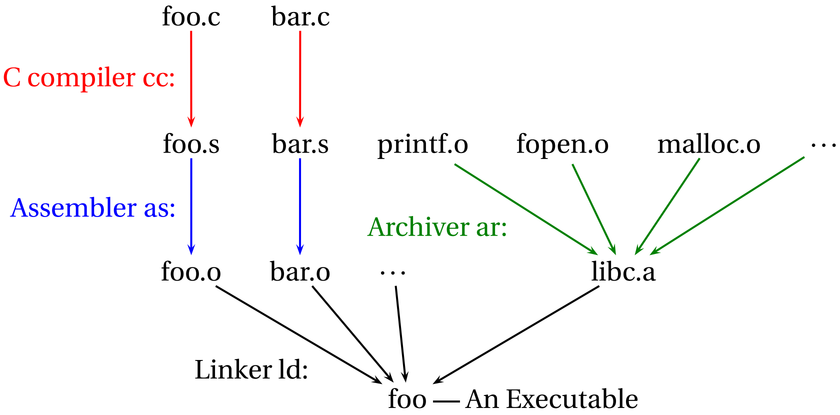
```
A: sne t, a, b
   bz  E, t

   slt t, a, b
   bnz B, t

   sub b, b, a
   jmp C

B: sub a, a, b

C: jmp A

E: ret a
```



Part X

Separate Compilation and Linking

Separate Compilation



Part XI

The Lambda Calculus

The Lambda Calculus

Fancy name for rules about how to represent and evaluate expressions with unnamed functions.

Theoretical underpinning of functional languages. Side-effect free.

Very different from the Turing model of a store with evolving state.

O'Caml:

```
fun x -> 2 * x
```

The Lambda Calculus:

$\boxed{\lambda x. * \; 2 \; x}$

English:

The function of x that returns the product of two and x

Grammar of Lambda Expressions

$$
\begin{array}{lcl}
\mathit{expr} & \rightarrow & \mathit{constant} \\
 & | & \mathit{variable\text{-}name} \\
 & | & \mathit{expr}\ \mathit{expr} \\
 & | & (\mathit{expr}) \\
 & | & \lambda\ \mathit{variable\text{-}name}\,.\ \mathit{expr}
\end{array}
$$

Constants are numbers; variable names are identifiers and operators.

Somebody asked, “does a language needs to have a large syntax to be powerful?”

Bound and Unbound Variables

In $\lambda x. * \ 2 \ x$, x is a *bound variable*. Think of it as a formal parameter to a function.

"$* \ 2 \ x$" is the *body*.

The body can be any valid lambda expression, including another unnnamed function.

$$\boxed{\lambda x.\lambda y. * \ (+ \ x \ y) \ 2}$$

"The function of x that returns the function of y that returns the product of the sum of x and y and 2."

Currying

$$\boxed{\lambda x.\lambda y. * \ (+\ x\ y)\ 2}$$

is equivalent to the O'Caml

```
fun x -> fun y -> (x + y) * 2
```

All lambda calculus functions have a single argument.

As in O'Caml, multiple-argument functions can be built through such "currying."

Currying is named after Haskell Brooks Curry (1900–1982), who contributed to the theory of functional programming. The Haskell functional language is named after him.

Calling Lambda Functions

To invoke a Lambda function, we place it in parentheses before its argument.

Thus, calling $\lambda x. * \ 2 \ x$ with 4 is written

$$\boxed{(\lambda x. * \ 2 \ x) \ 4}$$

This means 8.

Curried functions need more parentheses:

$$\boxed{(\lambda x.(\lambda y. * \ (+ \ x \ y) \ 2) \ 4) \ 5}$$

This binds 4 to y, 5 to x, and means 18.

Evaluating Lambda Expressions

Pure lambda calculus has no built-in functions; we'll be impure.

To evaluate $(+ \ (* \ 5 \ 6) \ (* \ 8 \ 3))$, we can't start with $+$ because it only operates on numbers.

There are two *reducible expressions*: $(* \ 5 \ 6)$ and $(* \ 8 \ 3)$. We can reduce either one first. For example:

$$
\begin{array}{l}
(+ \ (* \ 5 \ 6) \ (* \ 8 \ 3)) \\
(+ \ 30 \ (* \ 8 \ 3)) \\
(+ \ 30 \ 24) \\
54
\end{array}
$$

Looks like deriving a sentence from a grammar.

Evaluating Lambda Expressions

We need a reduction rule to handle λs:

$$(\lambda x. * \; 2 \; x) \; 4$$
$$(* \; 2 \; 4)$$
$$8$$

This is called β-reduction.

The formal parameter may be used several times:

$$(\lambda x. + \; x \; x) \; 4$$
$$(+ \; 4 \; 4)$$
$$8$$

Beta-reduction

May have to be repeated:

$$((\lambda x.(\lambda y.-\ x\ y))\ 5)\ 4$$
$$(\lambda y.-\ 5\ y)\ 4$$
$$(-\ 5\ 4)$$
$$1$$

Functions may be arguments:

$$(\lambda f.f\ 3)(\lambda x.+\ x\ 1)$$
$$(\lambda x.+\ x\ 1)3$$
$$(+\ 3\ 1)$$
$$4$$

More Beta-reduction

Repeated names can be tricky:

$$(\lambda x.(\lambda x. + \ (-\ x\ 1))\ x\ 3)\ 9$$
$$(\lambda x. + \ (-\ x\ 1))\ 9\ 3$$
$$+\ (-\ 9\ 1)\ 3$$
$$+\ 8\ 3$$
$$11$$

In the first line, the inner x belongs to the inner λ, the outer x belongs to the outer one.

Free and Bound Variables

In an expression, each appearance of a variable is either "free" (unconnected to a λ) or bound (an argument of a λ).

β-reduction of $(\lambda x.E)\ y$ replaces every x that *occurs free in E* with y.

Free or bound is a function of the position of each variable and its context.

Free variables

$$\boxed{(\lambda x.x\ y\ (\lambda y. +\ y))\ x}$$

Bound variables

Alpha conversion

One way to confuse yourself less is to do α-conversion.

This is renaming a λ argument and its bound variables.

Formal parameters are only names: they are correct if they are consistent.

$$\boxed{\lambda x.(\lambda x.x)\ (+\ 1\ x) \longleftrightarrow_{\alpha} \lambda x.(\lambda y.y)\ (+\ 1\ x)}$$

Alpha Conversion

An easier way to attack the earlier example:

$$(\lambda x.(\lambda x. + \ (- \ x \ 1)) \ x \ 3) \ 9$$
$$(\lambda x.(\lambda y. + \ (- \ y \ 1)) \ x \ 3) \ 9$$
$$(\lambda y. + \ (- \ y \ 1)) \ 9 \ 3$$
$$+ \ (- \ 9 \ 1) \ 3$$
$$+ \ 8 \ 3$$
$$11$$

Reduction Order

The order in which you reduce things can matter.

$$\boxed{(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )}$$

We could choose to reduce one of two things, either

$$\boxed{(\lambda z.z\ z)\ (\lambda z.z\ z)}$$

or the whole thing

$$\boxed{(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )}$$

Reduction Order

Reducing $(\lambda z.z\ z)\ (\lambda z.z\ z)$ effectively does nothing because $(\lambda z.z\ z)$ is the function that calls its first argument on its first argument. The expression reduces to itself:

$$\boxed{(\lambda z.z\ z)\ (\lambda z.z\ z)}$$

So always reducing it does not terminate.

However, reducing the outermost function does terminate because it ignores its (nasty) argument:

$$\boxed{\begin{array}{l}(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )\\ \lambda y.y\end{array}}$$

Reduction Order

The *redex* is a sub-expression that can be reduced.

The *leftmost* redex is the one whose λ is to the left of all other redexes. You can guess which is the *rightmost*.

The *outermost* redex is not contained in any other.

The *innermost* redex does not contain any other.

For $\boxed{(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ ),}$

$\boxed{(\lambda z.z\ z)\ (\lambda z.z\ z)}$ is the leftmost innermost and

$\boxed{(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )}$ is the leftmost outermost.

Applicative vs. Normal Order

Applicative order reduction: Always reduce the leftmost innermost redex.

Normative order reduction: Always reduce the leftmost outermost redex.

For

$$\boxed{(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )}$$

applicative order reduction never terminated; normative order did.

Applicative vs. Normal Order

Applicative:
reduce leftmost innermost

"evaluate arguments before the function itself"

eager evaluation, call-by-value, usually more efficient

Normative:
reduce leftmost outermost

"evaluate the function before its arguments"

lazy evaluation, call-by-name, more costly to implement, accepts a larger class of programs

Normal Form

A lambda expression that cannot be reduced further is in *normal form*.

Thus,

$$\boxed{\lambda y.y}$$

is the normal form of

$$\boxed{(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )}$$

Normal Form

Not everything has a normal form. E.g.,

$$\boxed{(\lambda z.z\ z)\ (\lambda z.z\ z)}$$

can only be reduced to itself, so it never produces an non-reducible expression.

“Infinite loop.”

Part XII

Logic Programming

Unification

Part of the search procedure that matches patterns.

The search attempts to match a goal with a rule in the database by unifying them.

Recursive rules:

- A constant only unifies with itself
- Two structures unify if they have the same functor, the same number of arguments, and the corresponding arguments unify
- A variable unifies with anything but forces an equivalence

Unification Examples

The = operator checks whether two structures unify:

```
| ?- a = a.
yes                              % Constant unifies with itself
| ?- a = b.
no                               % Mismatched constants
| ?- 5.3 = a.
no                               % Mismatched constants
| ?- 5.3 = X.
X = 5.3?;                        % Variables unify
no
| ?- foo(a,X) = foo(X,b).
no                               % X=a required, but inconsistent
| ?- foo(a,X) = foo(X,a).
X = a?;                          % X=a is consistent
no
| ?- foo(X,b) = foo(a,Y).
X = a
Y = b?;                          % X=a, then b=Y
no
| ?- foo(X,a,X) = foo(b,a,c).
no                               % X=b required, but inconsistent
```

The Searching Algorithm

in the order they appear

search(goal g, variables e)
  for each clause h :- $t_1, \ldots, t_n$ in the database
    $e = \text{unify}(g, h, e)$ in the order they appear
    if successful,
      for each term $t_1, \ldots, t_n$,
        $e = \text{search}(t_k, e)$
    if all successful, return e
  return no

Note: This pseudo-code ignores one very important part of the searching process!

Order Affects Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :-
    edge(X, Z), path(Z, Y).
```

Consider the query

```
?- path(a, a).
```

path(a,a)

path(a,a)=path(X,X)

X=a

yes

Good programming practice: Put the easily-satisfied clauses first.

Order Affect Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, Y) :-
    edge(X, Z), path(Z, Y).
path(X, X).
```

Consider the query

```
?- path(a, a).
```

path(a,a)

path(a,a)=path(X,Y)

X=a Y=a

edge(a,Z)

edge(a,Z)=edge(a,b)

Z=b

path(b,a)

⋮

Will eventually produce the right answer, but will spend much more time doing so.

Order can cause Infinite Recursion

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, Y) :-
   path(X, Z), edge(Z, Y).
path(X, X).
```

Consider the query

```
?- path(a, a).
```