Review for the Final

COMS W4115

Prof. Stephen A. Edwards
Spring 2007
Columbia University
Department of Computer Science

The Final

70 minutes

4–5 problems

Closed book

One single-sided 8.5 $\times$ 11 sheet of notes of your own devising

Comprehensive: Anything discussed in class is fair game

Little, if any, programming.

You should understand ANTLR/C/Java/ML/Prolog

You do not need to know details of syntax

Topics 1

Structure of a Compiler

Scripting Languages

Scanning and Parsing

Regular Expressions

Context-Free Grammars

Top-down Parsing

Bottom-up Parsing

ASTs

Name, Scope, and Bindings

Control-flow constructs

Topics 2

Types

Static Semantic Analysis

Code Generation

Functional Programming (ML, Lambda Calculus)

Logic Programming (Prolog)

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

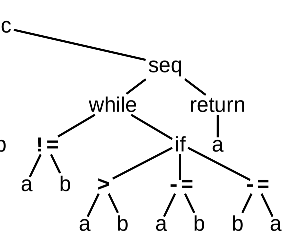
Lexical Analysis Gives Tokens

```
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.

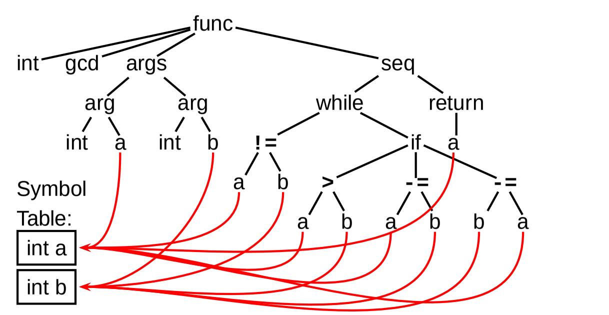
Parsing Gives an AST




```
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 frame pointer
      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
```

Scanning and Automata

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 $L = \{$ ϵ, 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, $\ldots\}$

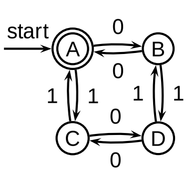
Regular Expressions over an Alphabet Σ

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 $\cup_{i=0}^{\infty} L^i$ ($L^0 = \emptyset$ and $L^i = LL^{i-1}$)

Nondeterministic Finite Automata

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



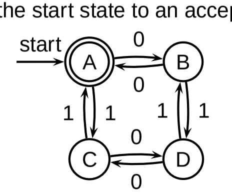
1. Set of states S: $\{$ A, B, C, D $\}$
2. Set of input symbols Σ: $\{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \to 2^S$

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

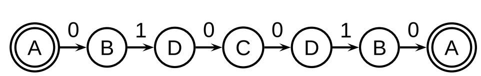
4. Start state s_0 : A
5. Set of accepting states F: $\{$ A $\}$

The Language induced by an NFA

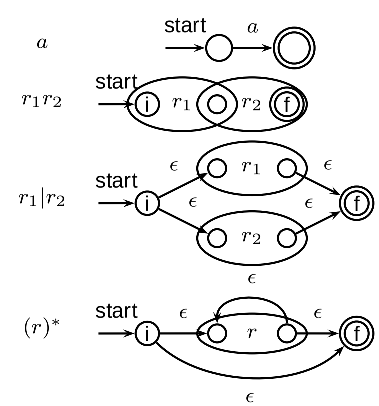
An NFA accepts an input string x iff there is a path from the start state to an accepting state that "spells out" x.



Show that the string "010010" is accepted.

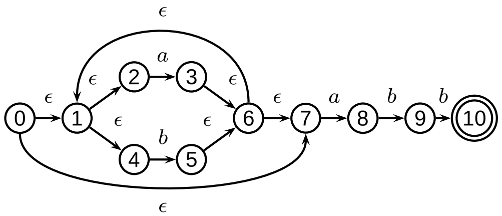


Translating REs into NFAs

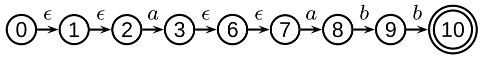


Translating REs into NFAs

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



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



Simulating NFAs

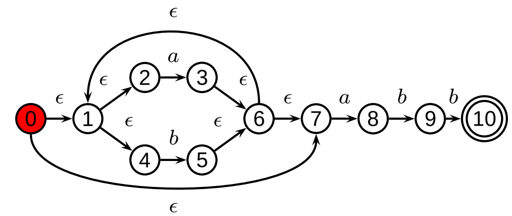
Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

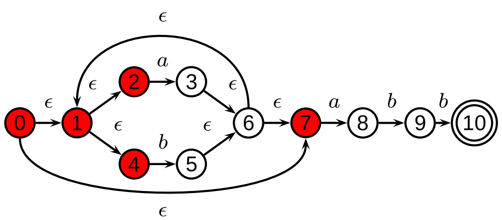
"Two-stack" NFA simulation algorithm:

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

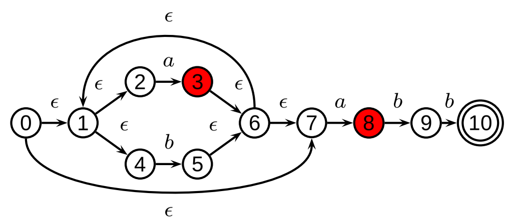
Simulating an NFA: $\cdot aabb$, Start



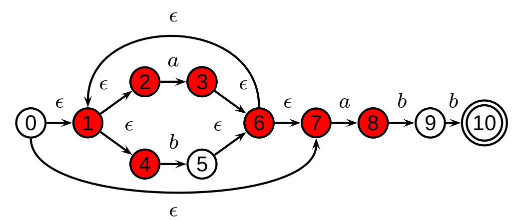
Simulating an NFA: $\cdot aabb$, ϵ-closure



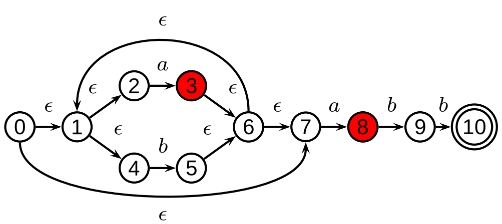
Simulating an NFA: $a{\cdot}abb$



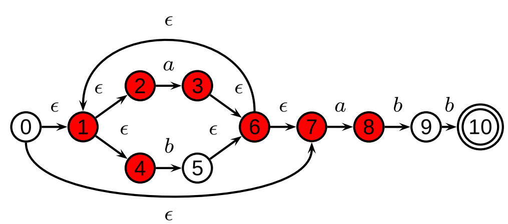
Simulating an NFA: $a{\cdot}abb$, ϵ-closure



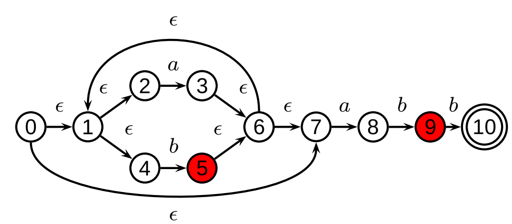
Simulating an NFA: $aa{\cdot}bb$



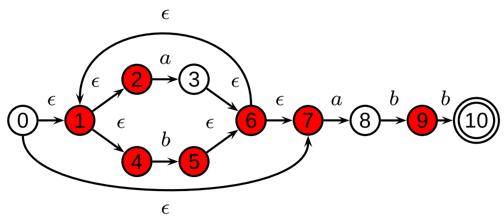
Simulating an NFA: $aa{\cdot}bb$, ϵ-closure



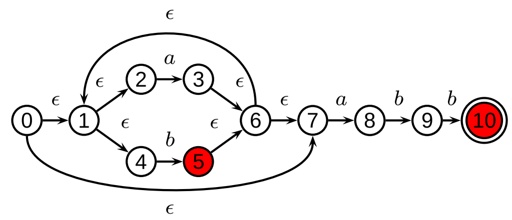
Simulating an NFA: $aab{\cdot}b$



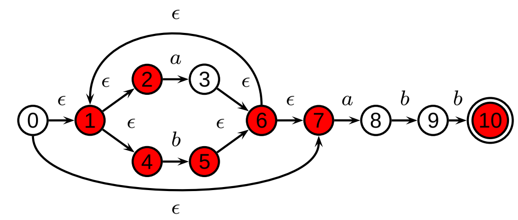
Simulating an NFA: $aab{\cdot}b$, ϵ-closure



Simulating an NFA: $aabb{\cdot}$



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



Deterministic Finite Automata

Restricted form of NFAs:

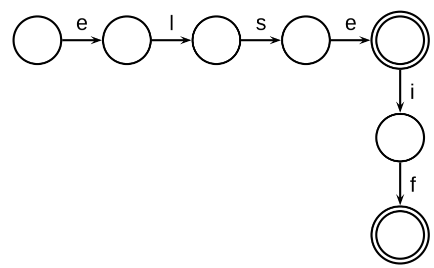
- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

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.

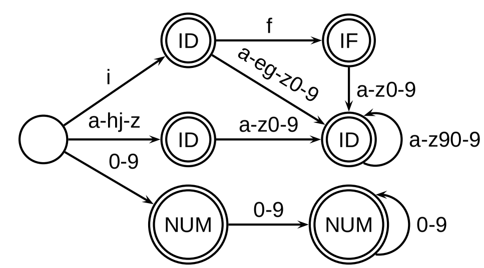
Deterministic Finite Automata

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



Deterministic Finite Automata

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



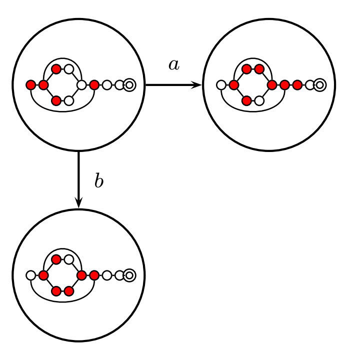
Building a DFA from an NFA

Subset construction algorithm

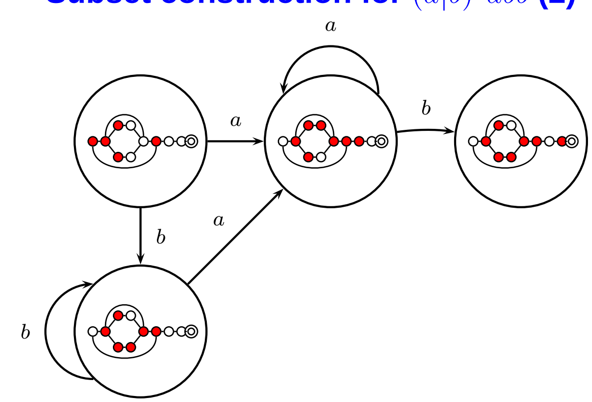
Simulate the NFA for all possible inputs and track the states that appear.

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

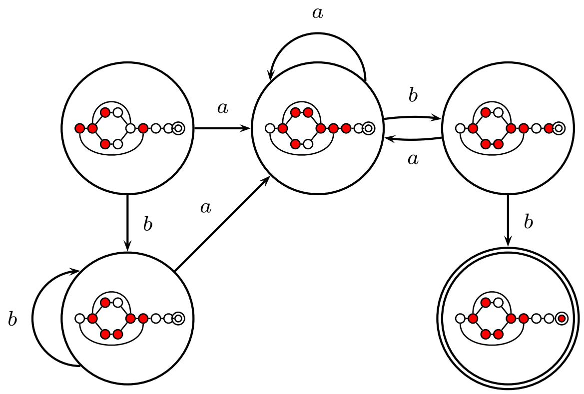
Subset construction for $(a|b)^*abb$ (1)



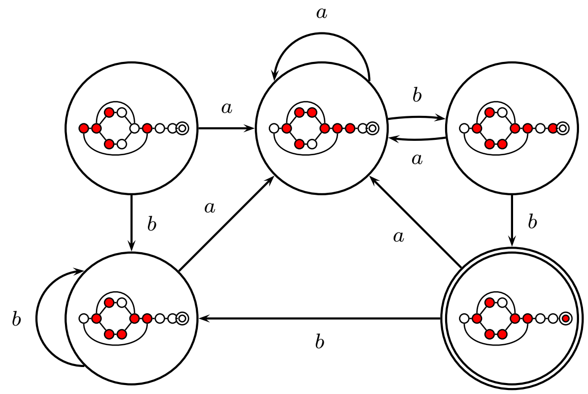
Subset construction for $(a|b)^*abb$ (2)



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



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



Grammars and Parsing

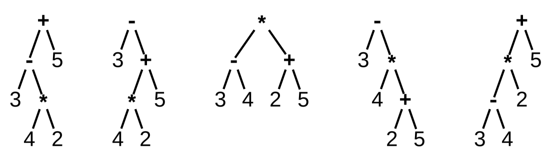
Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

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

with the grammar

$e \rightarrow e + e \,|\, e - e \,|\, e * e \,|\, e \,/\, e$



Fixing Ambiguous Grammars

Original ANTLR grammar specification

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

Ambiguous: no precedence or associativity.

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

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 ;
```

A Top-Down Parser

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

expr : NUMBER | '(' expr ')' ;
```

```
AST stmt() {
 switch (next-token) {
 case "if" : match("if"); expr(); match("then"); expr();
 case "while" : match("while"); expr(); match("do"); expr();
 case NUMBER or "(" : expr(); match(":="); expr();
 }
}
```

Writing LL(k) Grammars

Cannot have left-recursion

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

becomes

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

Writing LL(1) Grammars

Cannot have common prefixes

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

becomes

AST expr() **{**
  switch (next-token) **{**
  case ID : match(ID); match('('); expr(); match(')');
  case ID : match(ID); match('='); expr();

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 */
      ;
```

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**:

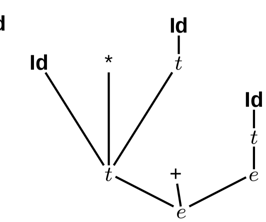
$\boxed{e}$
$t + \boxed{e}$
$t + \boxed{t}$
$\boxed{t} + \textbf{Id}$
$\textbf{Id} * \boxed{t} + \textbf{Id}$
$\textbf{Id} * \textbf{Id} + \textbf{Id}$

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

Handles

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

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

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

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

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

An oracle tells 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	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

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

	action				goto	
	Id	$+$	$*$	$	e	t
0	s1				7	2
1	r4	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

$1: \; e \to t + e$

$2: \; e \to t$

$3: \; t \to \textbf{Id} * t$

$4: \; t \to \textbf{Id}$

	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.

$1: \; e \to t + e$

$2: \; e \to t$

$3: \; t \to \textbf{Id} * t$

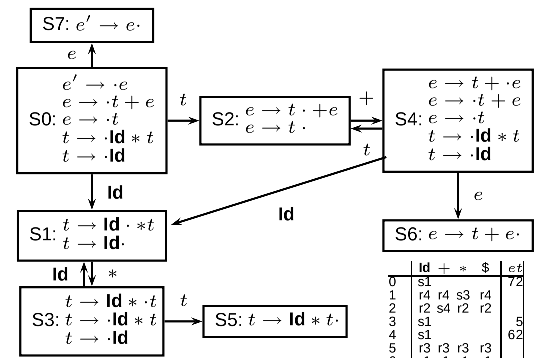
$4: \; t \to \textbf{Id}$

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

$e' \to \cdot e$
$e \to \cdot t + e$
$e \to \cdot t$
$t \to \cdot\textbf{Id} * t$
$t \to \cdot\textbf{Id}$

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.

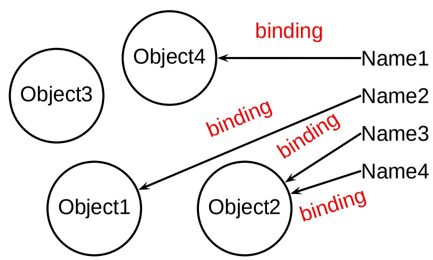
Constructing the SLR Parsing Table



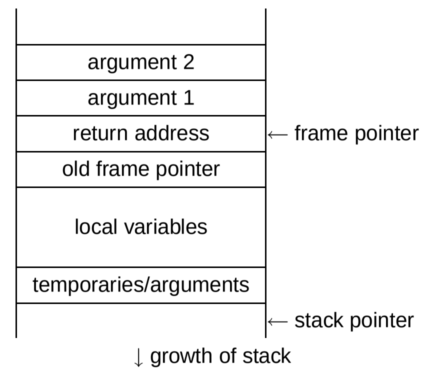
	Id	+	*	$	e	t
0	s1				7	2
1	r4	r4	s3	r4		
2	r2	s4	r2	r2		
3	s1					5
4	s1				6	2
5	r3	r3	r3	r3		
6	r1	r1	r1	r1		
7				acc		

Names, Objects, and Bindings

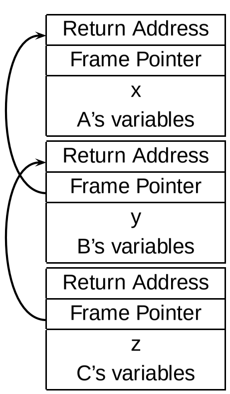
Names, Objects, and Bindings



Activation Records



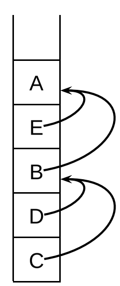
Activation Records




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

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

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

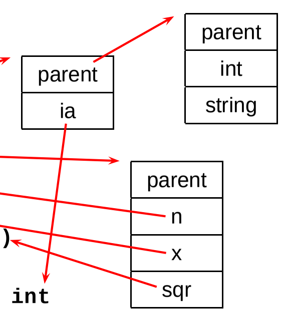
Nested Subroutines in Pascal

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

    procedure D;
    begin C end
  begin D end

  procedure E;
  begin B end
begin E 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
```



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            1 - 5

   +                  -
  / \                / \
 1   break          1   5
```

check(+)
  check(1) = int
  check(break) = void
  FAIL: int $\neq$ void

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;
}
```

```
labels l[] = { L1, L2, L3, L4 }; /* Array of labels */
if (s>=1 && s<=4) goto l[s-1];   /* not legal C */
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 processors have byte-addressable memory.

0
1
2
3
4

Many data types (integers, addresses, floating-point numbers) are wider than a byte.

16-bit integer:			1	0
32-bit integer:	3	2	1	0

Layout of Records and Unions

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

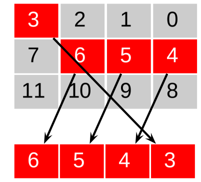
3	2	1	0
7	6	5	4
11	10	9	8

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 */
};
```

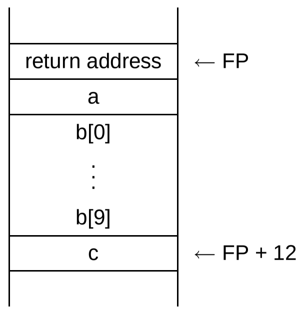
x	x	x	x
y	y		z
			w

: Added padding

Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

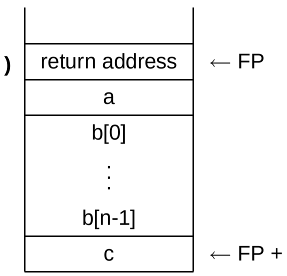
```
void foo()
{
  int a;
  int b[10];
  int c;
}
```



Allocating Variable-Sized Arrays

Variable-sized local arrays aren’t as easy.

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

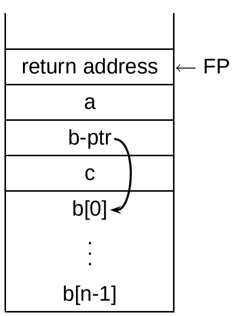


Doesn’t work: generated code expects a fixed offset for c. Even worse for multi-dimensional arrays.

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.

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.

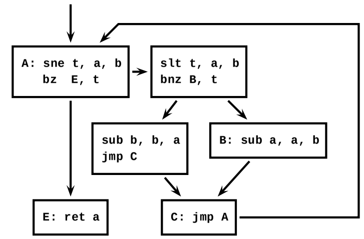
Starts with a statement following a conditional branch or is a branch target.

Usually ends with a control-transfer statement.

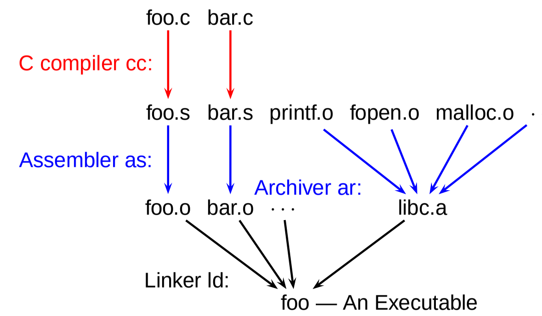
Control-Flow Graphs

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
```



Separate Compilation



Simple functional programming in ML

A function that squares numbers:

```
% sml
Standard ML of New Jersey, Version 110.0.7
- fun square x = x * x;
val square = fn : int -> int
- square 5;
val it = 25 : int
-
```

A more complex function

```
- fun max a b =
=     if a > b then a else b;
val max = fn : int -> int -> int
- max 10 5;
val it = 10 : int
- max 5 10;
val it = 10 : int
-
```

Notice the odd type:

```
int -> int -> int
```

This is a function that takes an integer and returns a function that takes a function and returns an integer.

Currying

Functions are first-class objects that can be manipulated with abandon and treated just like numbers.

```
- fun max a b = if a > b then a else b;
val max = fn : int -> int -> int
- val max5 = max 5;
val max5 = fn : int -> int
- max5 4;
val it = 5 : int
- max5 6;
val it = 6 : int
-
```

Fun with recursion

```
- fun addto (l,v) =
=   if null l then nil
=   else hd l + v :: addto(tl l, v);
val addto = fn : int list * int -> int list

- addto([1,2,3],2);
val it = [3,4,5] : int list
```

More recursive fun

```
- fun map (f, l) =
=   if null l then nil
=   else f (hd l) :: map(f, tl l);
val map = fn : ('a -> 'b) * 'a list -> 'b list

- fun add5 x = x + 5;
val add5 = fn : int -> int

- map(add5, [10,11,12]);
val it = [15,16,17] : int list
```

Reduce

Another popular functional language construct:

```
fun reduce (f, z, nil) = z
  | reduce (f, z, h::t) = f(h, reduce(f, z, t));
```

If **f** is "$-$", **reduce(f,z,a::b::c)** is $a-(b-(c-z))$

```
- reduce( fn (x,y) => x - y, 0, [1,5]);
val it = ~4 : int
- reduce( fn (x,y) => x - y, 2, [10,2,1]);
val it = 7 : int
```

Another Example

Consider

```
- fun find1(a,b) =
=   if b then true else (a = 1);
val find1 = fn : int * bool -> bool

- reduce(find1, false, [3,3,3]);
val it = false : bool

- reduce(find1, false, [5,1,2]);
val it = true : bool
```

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.

ML:	The Lambda Calculus:
`fn x => 2 * x;`	$\lambda x. * \; 2 \; x$

English:

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

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.

$$\lambda x.\lambda y. * \; (+ \; x \; y) \; 2$$

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

Arguments

$\lambda x.\lambda y. * \; (+ \; x \; y) \; 2$

is equivalent to the ML

```
fn x => fn y => (x + y) * 2;
```

All lambda calculus functions have a single argument.

As in ML, multiple-argument functions can be built through such "currying."

In this context, currying has nothing to do with Indian cooking. It is due to 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

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

This means 8.

Curried functions need more parentheses:

$(\lambda x.(\lambda y. * \; (+ \; x \; y) \; 2) \; 4) \; 5$

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

Grammar of Lambda Expressions

Utterly trivial:

expr	$\rightarrow$	*constant*
	|	*variable*
	|	*expr expr*
	|	(*expr*)
	|	λ *variable* . *expr*

Somebody asked whether a language needs to have a large syntax to be powerful. Clearly, the answer is a resounding "no."

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:

$(+ \; (* \; 5 \; 6) \; (* \; 8 \; 3))$
$(+ \; 30 \; (* \; 8 \; 3))$
$(+ \; 30 \; 24)$
54

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

$(\lambda x.x\ y\ (\lambda y.+\ y\ ))\ x$

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.

$\lambda x.(\lambda x.x)\ (+\ 1\ x) \leftrightarrow_\alpha \lambda x.(\lambda y.y)\ (+\ 1\ x)$

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.

$(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )$

We could choose to reduce one of two things, either

$(\lambda z.z\ z)\ (\lambda z.z\ z)$

or the whole thing

$(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )$

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:

$(\lambda z.z\ z)\ (\lambda z.z\ z)$

So always reducing it does not terminate.

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

$(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )$

$\lambda y.y$

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 $(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )$,

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

$(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )$ 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 $(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )$, applicative order reduction never terminated but 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,

$\lambda y.y$

is the normal form of

$(\lambda x.\lambda y.y)\ (\ (\lambda z.z\ z)\ (\lambda z.z\ z)\ )$

Normal Form

Not everything has a normal form

$(\lambda z.z\ z)\ (\lambda z.z\ z)$

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

"Infinite loop."

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

search(goal g, variables e)

  for each clause h **:-** $t_1, \ldots, t_n$ in the database

    $e =$ unify(g, h, e)

    if successful,

      for each term $t_1, \ldots, t_n$,

        $e =$ search(t_k, e)

      if all successful, return e

  return **no**

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).
```

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

Consider the query

?- path(a, a).

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).
```

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)
⋮

Consider the query

?- path(a, a).

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