Review for the Final

COMS W4115

Prof. Stephen A. Edwards
Spring 2003
Columbia University
Department of Computer Science

The Final

Like 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 ANTLR/C/Java/Prolog/ML syntax not required

Broad knowledge of languages discussed

Topics (1)

Structure of a Compiler

Scanning and Parsing

Regular Expressions

Context-Free Grammars

Top-down Parsing

Bottom-up Parsing

ASTs

Name, Scope, and Bindings

Types

Control-flow constructs

Topics (2)

Code Generation

Logic Programming: Prolog

Concurrency: Locks and deadlocks

Functional Programming: ML and the Lambda Calculus

Scripting Languages

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

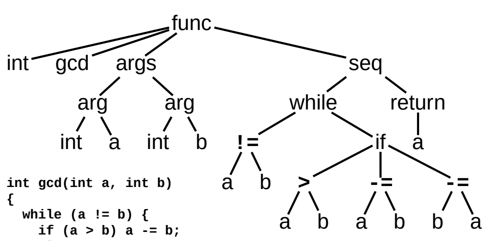
Lexical Analysis Gives Tokens

```
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.

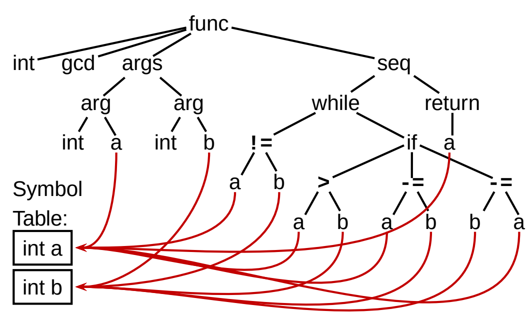
Parsing Gives an AST




```
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

Deterministic Finite Automata

A state machine with an initial state

Arcs indicate “consumed” input symbols.

States with double lines are accepting.

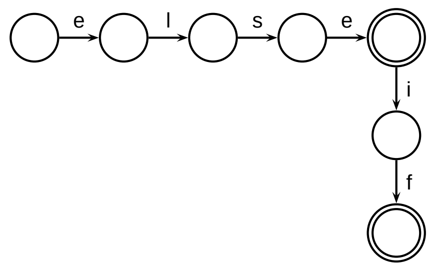
If the next token has an arc, follow the arc.

If the next token has no arc and the state is accepting, return the token.

If the next token has no arc and the state is not accepting, syntax error.

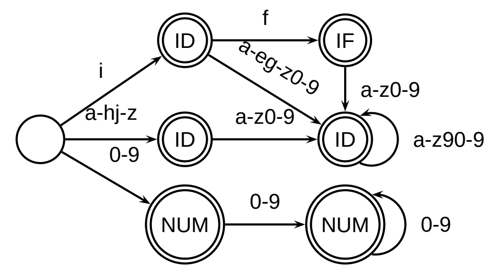
Deterministic Finite Automata

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



Deterministic Finite Automata

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



Nondeterminstic Finite Automata

DFAs with ϵ arcs.

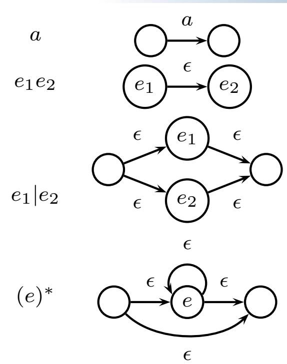
Conceptually, ϵ arcs denote state equivalence.

ϵ arcs add the ability to make nondeterministic (schizophrenic) choices.

When an NFA reaches a state with an ϵ arc, it moves into *every* destination.

NFAs can be in multiple states at once.

Translating REs into NFAs

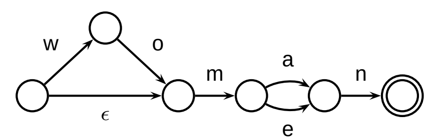


RE to NFAs

Building an NFA for the regular expression

$(wo|\epsilon)m(a|e)n$

produces

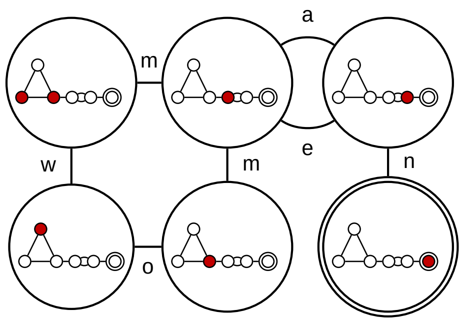


after simplification. Most ϵ arcs disappear.

Subset Construction

How to compute a DFA from an NFA.

Basic idea: each state of the DFA is a *marking* of the NFA



Subset Construction

An DFA can be exponentially larger than the corresponding NFA.

n states versus 2^n

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

ANTLR uses a different technique.

Grammars and Parsing

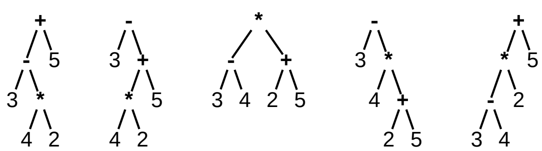
Ambiguous Grammars

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

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

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

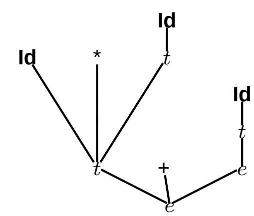
e
$t + e$
$t + t$
$t +$ **Id**
Id $* t +$ **Id**
Id $*$ **Id** $+$ **Id**

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

Handles

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

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



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

Each highlighted section is a handle.

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

Shift-reduce Parsing

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

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

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

An oracle tells what to do

LR Parsing

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

stack	input	action
$\boxed{0}$	**Id** * **Id** + **Id** \$	shift, goto 1

	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		

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

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

stack	input	action
$\boxed{0}$	**Id** * **Id** + **Id** \$	shift, goto 1
$\boxed{0}\ \boxed{^{\textbf{Id}}_{1}}$	* **Id** + **Id** \$	shift, goto 3
$\boxed{0}\ \boxed{^{\textbf{Id}}_{1}}\ \boxed{^{*}_{3}}$	**Id** + **Id** \$	shift, goto 1
$\boxed{0}\ \boxed{^{\textbf{Id}}_{1}}\ \boxed{^{*}_{3}}\ \boxed{^{\textbf{Id}}_{1}}$	+ **Id** \$	reduce w/ 4

	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		

Action is reduce with rule 4 ($t \to$ **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
$\boxed{0}\ \boxed{^{\textbf{Id}}_{1}}\ \boxed{^{*}_{3}}\ \boxed{^{t}_{5}}$	+ **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.

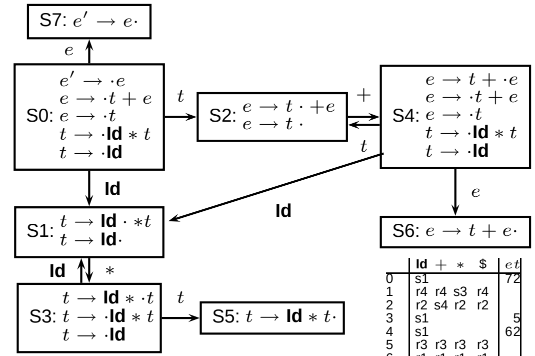
$1: e \to t + e$
$2: e \to t$
$3: t \to \textbf{Id} * t$
$4: t \to \textbf{Id}$

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.

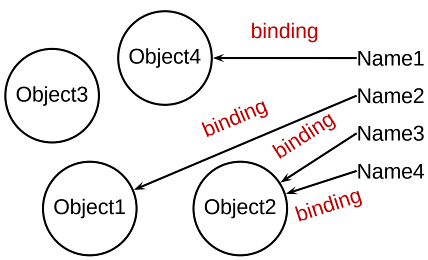
Constructing the SLR Parsing Table



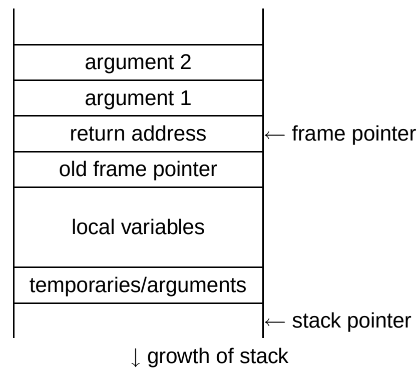
	Id	$+$	$*$	\$	e t
0	s1				72
1	r4	r4	s3	r4	
2	r2	s4	r2	r2	
3	s1				5
4	s1				62
5	r3	r3	r3	r3	
6	r1	r1	r1	r1	
7				acc	

Names, Objects, and Bindings

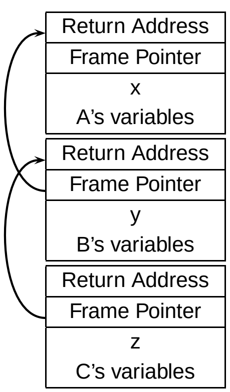
Names, Objects, and Bindings



Activation Records



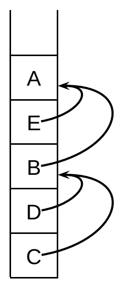
Activation Records



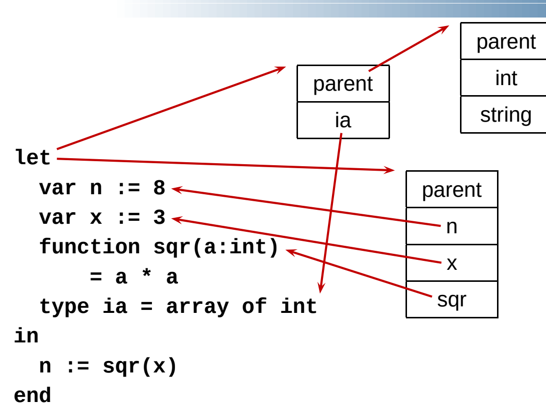

```
int A() {
  int x;
  B();
}

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
```

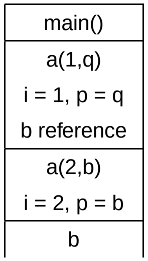
Shallow vs. Deep binding

```
typedef int (*ifunc)();
ifunc foo() {
  int a = 1;
  int bar() { return a; }
  return bar;
}
int main() {
  ifunc f = foo();
  int a = 2;
  return (*f)();
}
```

	static	dynamic
shallow	1	2
deep	1	1

Shallow vs. Deep binding

```
void a(int i, void (*p)()) {

  void b() { printf("%d", i); }

  if (i=1) a(2,b) else (*p)();
}

void q() {}

int main() {
  a(1,q);
}
```



	static
shallow	2
deep	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:

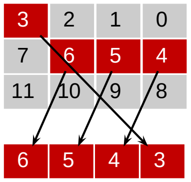
3	2	1	0
7	6	5	4
11	10	9	8

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 */
};
```

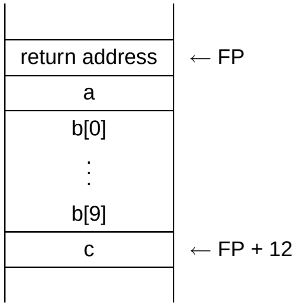
x	x	x	x
y	y		z
			w

: Added padding

Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

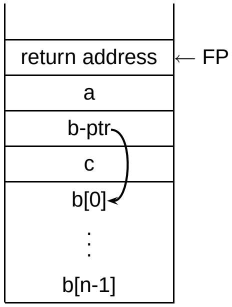
```
void foo()
{
  int a;
  int b[10];
  int c;
}
```



Allocating Variable-Sized Arrays

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.

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(-)
check(1) = int	check(1) = int
check(break) = void	check(5) = int
FAIL: int $\neq$ void	Types match, return int

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

Mid-test Loops

```
while true do begin
  readln(line);
  if all_blanks(line) then goto 100;
  consume_line(line);
end;
100:
```

```
LOOP
  line := ReadLine;
WHEN AllBlanks(line) EXIT;
  ConsumeLine(line)
END;
```

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

Prolog

Prolog

All Caltech graduates are nerds.	`nerd(X) :- techer(X).`
Stephen is a Caltech graduate.	`techer(stephen).`
Is Stephen a nerd?	`?- nerd(stephen).` `yes`

Structures and Functors

A structure consists of a functor followed by an open parenthesis, a list of comma-separated terms, and a close parenthesis:

"Functor"

paren must follow immediately

```
bin_tree( foo, bin_tree(bar, glarch) )
```

What's a structure? Whatever you like.

A predicate **`nerd(stephen)`**
A relationship **`teaches(edwards, cs4115)`**
A data structure **`bin(+, bin(-, 1, 3), 4)`**

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**

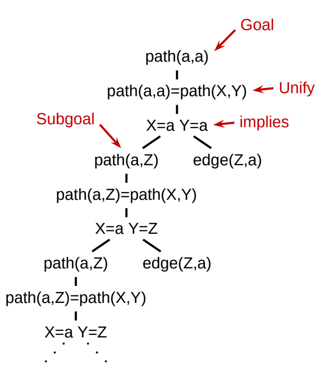
Order can cause Infinite Recursion

```
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).`

Like LL(k) grammars.



Concurrency

Coroutines

```
char c;
                          parse() {
void scan() {               char buf[10];
  c = 's';      <-------    transfer scan;
  transfer parse; ------>   buf[0] = c;
  c = 'a';      <-------    transfer scan;
  transfer parse; ------>   buf[1] = c;
  c = 'e';      <-------    transfer scan;
  transfer parse; ------>   buf[2] = c;
}                         }
```

Cooperative Multitasking

Typical MacOS < 10 or Windows < 95 program:

```
void main() {
  Event e;
  while ( (e = get_next_event()) != QUIT ) {   Magical
    switch (e) {
      case CLICK: /* ... */ break;
      case DRAG: /* ... */ break;
      case DOUBLECLICK: /* ... */ break;
      case KEYDOWN: /* ... */ break;
      /* ... */
    }
  }
}
```

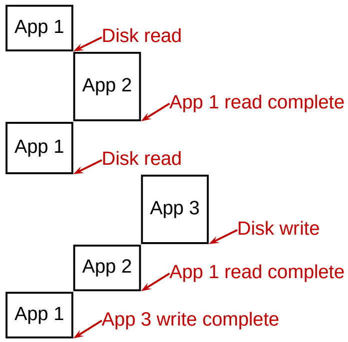
Multiprogramming

Avoids I/O busy waiting.

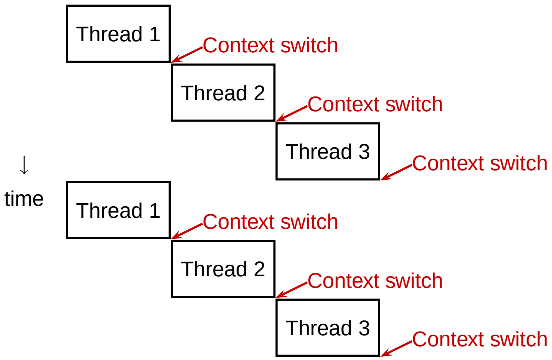
Context switch on I/O request.

I/O completion triggers interrupt.

Interrupt causes context switch.



Three Threads on a Uniprocessor



Races

In a concurrent world, always assume something else is accessing your objects.

Other threads are your adversary

Consider what can happen when two threads are simultaneously reading and writing.

Thread 1	Thread 2
`f1 = a.field1`	`a.field1 = 1`
`f2 = a.field2`	`a.field2 = 2`

Thread 1 sees old values

Thread 1 runs before Thread 2

Thread 1	Thread 2
1 `f1 = a.field1` = old value	
2 `f2 = a.field2` = old value	
	`a.field1 = 1` 3
	`a.field2 = 2` 4

Thread 1 sees new values

Thread 1 runs after Thread 2

Thread 1	Thread 2
	`a.field1 = 1` 1
	`a.field2 = 2` 2
3 `f1 = a.field1` = 1	
4 `f2 = a.field2` = 2	

Thread 1 sees inconsistent values

Execution of Thread 1 interrupts execution of Thread 2

Thread 1	Thread 2
	`a.field1 = 1` 1
2 `f1 = a.field1` = 1	
3 `f2 = a.field2` = old value	
	`a.field2 = 2` 4

Synchronized Methods

```
class AtomCount {
  int c1 = 0, c2 = 2;

  public synchronized void count() {
    c1++; c2++;
  }

  public synchronized int readcount() {
    return c1 + c2;
  }
}
```

Grab lock while method running

Object's lock acquired when a **synchronized** method is invoked.

Lock released when method terminates.

Java's Solution: wait() and notify()

`wait()` is like `yield()`, but a waiting thread can only be reawakened by another thread.

Always in a loop; could be awakened before condition is true

```
while (!condition()) wait();
```

Thread that might affect the condition calls `notify()` to resume the thread.

Programmer's responsible for ensuring each `wait()` has a matching `notify()`.

wait() and notify()

Each object maintains a set of threads that are waiting for its lock (its wait set).

```
synchronized (obj) {   // Acquire lock on obj
  obj.wait();          // Suspend and add this thread
                       // to obj's wait set
}                      // Relinquish locks on obj
```

Other thread:

```
obj.notify();          // Awaken some waiting thread
```

wait() and notify()

Thread 1 acquires lock on obj

Thread 1 calls `wait()` on obj

Thread 1 releases lock on obj and adds itself to object's wait set.

Thread 2 calls `notify()` on obj (must have acquired lock)

Thread 1 is reawakened; it was in obj's wait set

Thread 1 reacquires lock on obj

Thread 1 continues from the `wait()`

wait() and notify()

Confusing enough?

`notify()` nondeterministically chooses one thread to reawaken (many may wait on the same object). So what happens where there's more than one?

`notifyAll()` enables all waiting threads. Much safer.

Building a Blocking Buffer

```
class OnePlace {
  El value;

  public synchronized void write(El e) { .. }
  public synchronized El read() { .. }
}
```

Only one thread may read or write the buffer at any time

Thread will block on read if no data is available

Thread will block on write if data has not been read

Building a Blocking Buffer

```
synchronized void write(El e)
   throws InterruptedException {
  while (value != null)
     wait();     // Block while full
  value = e;
  notifyAll(); // Awaken any waiting read
}

public synchronized El read()
   throws InterruptedException {
  while (value == null)
     wait();     // Block while empty
  El e = value; value = null;
  notifyAll(); // Awaken any waiting write
  return e;
}
```

Functional Programming

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
-
```

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
-
```

Recursion

ML doesn't have variables in the traditional sense, so you can't write programs with loops.

So use recursion:

```
- fun sum n =
=   if n = 0 then 0 else sum(n-1) + n;
val sum = fn : int -> int
- sum 2;
val it = 3 : int
- sum 3;
val it = 6 : int
- sum 4;
val it = 10 : int
```

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
```

But why always name functions?

```
- map( fn x => x + 5, [10,11,12]);
val it = [15,16,17] : int list
```

This is called a *lambda* expression: it's simply an unnamed function.

The **fun** operator is similar to a lambda expression:

```
- val add5 = fn x => x + 5;
val add5 = fn : int -> int
- add5 10;
val it = 15 : int
```

Pattern Matching

Functions are often defined over ranges

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise.} \end{cases}$$

Functions in ML are no different. How to cleverly avoid writing if-then:

```
fun map (f,[]) = []
  | map (f,l) = f (hd l) :: map(f,tl l);
```

Pattern matching is order-sensitive. This gives an error.

```
fun map (f,l) = f (hd l) :: map(f,tl l)
  | map (f,[]) = [];
```

Pattern Matching

More fancy binding

```
fun map (_,[]) = []
  | map (f,h :: t) = f h :: map(f,t);
```

"_" matches anything

h :: t matchs a list, binding **h** to the head and **t** to the tail.

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.

ML:

fn x => 2 * x;

The Lambda Calculus:

$\lambda x . * \ 2 \ x$

English:

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

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

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."

The Church-Rosser Theorems

If $E_1 \leftrightarrow E_2$ (are interconvertable), then there exists an E such that $E_1 \rightarrow E$ and $E_2 \rightarrow E$.

"Reduction in any way can eventually produce the same result."

If $E_1 \rightarrow E_2$, and E_2 is is normal form, then there is a normal-order reduction of E_1 to E_2.

"Normal-order reduction will always produce a normal form, if one exists."

Church-Rosser

Amazing result:

Any way you choose to evaluate a lambda expression will produce the same result.

Each program means exactly one thing: its normal form.

The lambda calculus is deterministic w.r.t. the final result.

Normal order reduction is the most general.