Review

COMS W4115

Prof. Stephen A. Edwards
Spring 2002
Columbia University
Department of Computer Science

Midterm 2 a.k.a. The Final

One single-sided 8.5×11 cheatsheet of your own devising

5 problems; 70 minutes

Covers whole class

Remember your Uni ID

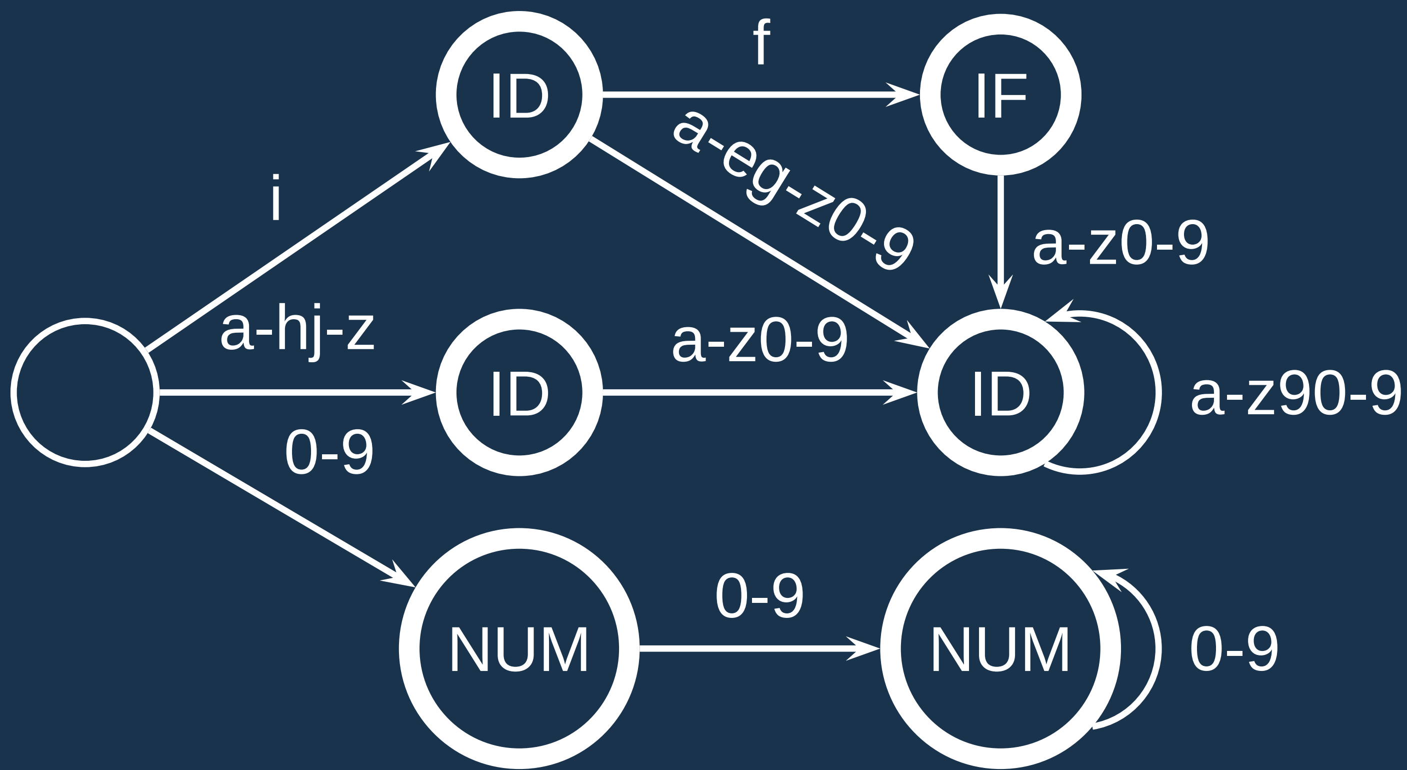
No ANTLR

Structure of a Compiler

↓	Program Text
Lexer	
↓	Token Stream
Parser	
↓	Abstract Syntax Tree
Static semantics (type checking)	
↓	Annotated AST
Translation to intermediate form	
↓	Three-address code
Code generation	
↓	Assembly Code

Deterministic Finite Automata

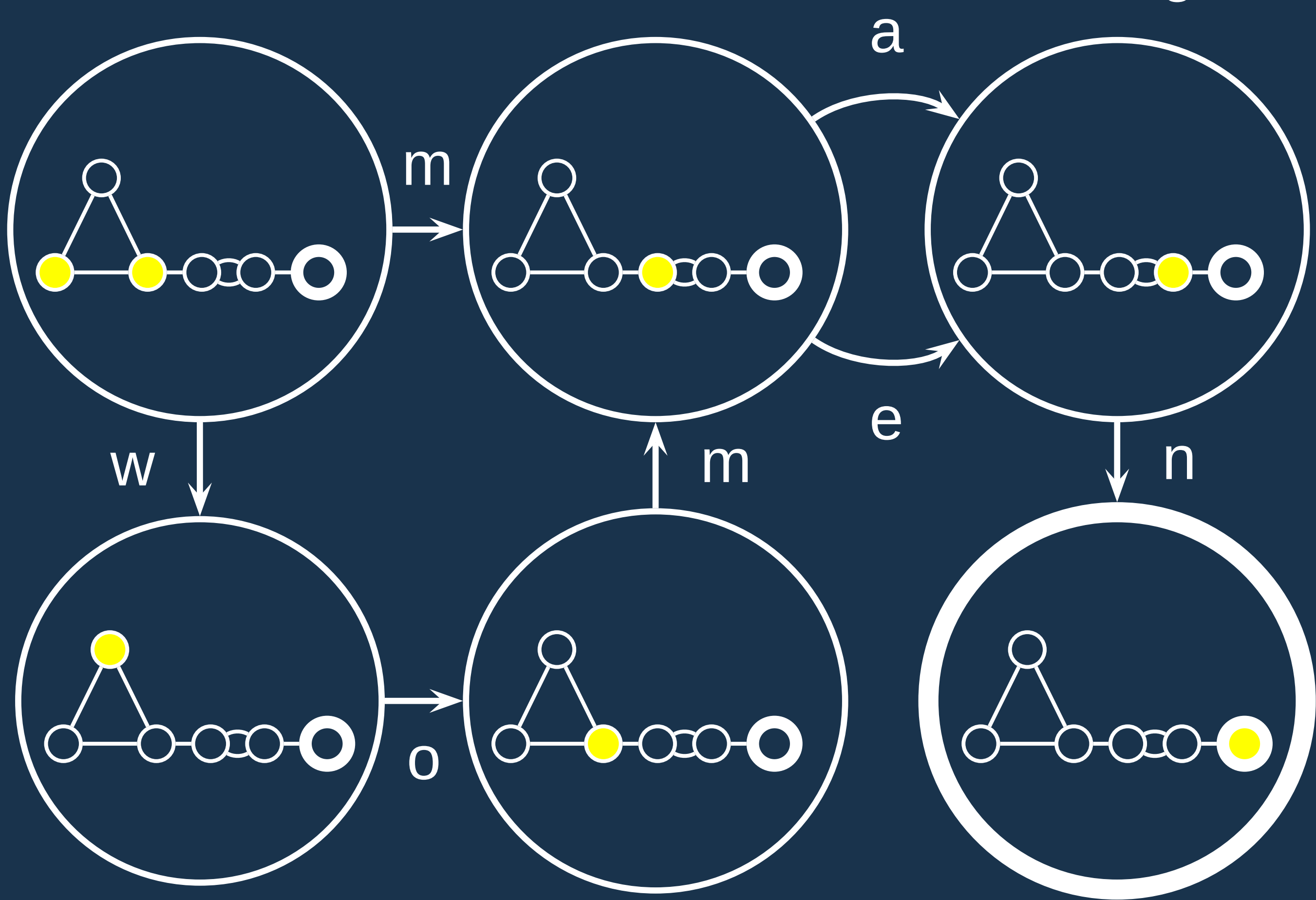
```
IF: "if" ;
ID: 'a'..'z' ('a'..'z' | '0'..'9')* ;
NUM: ('0'..'9')+ ;
```



Subset Construction

How to compute a DFA from an NFA.

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



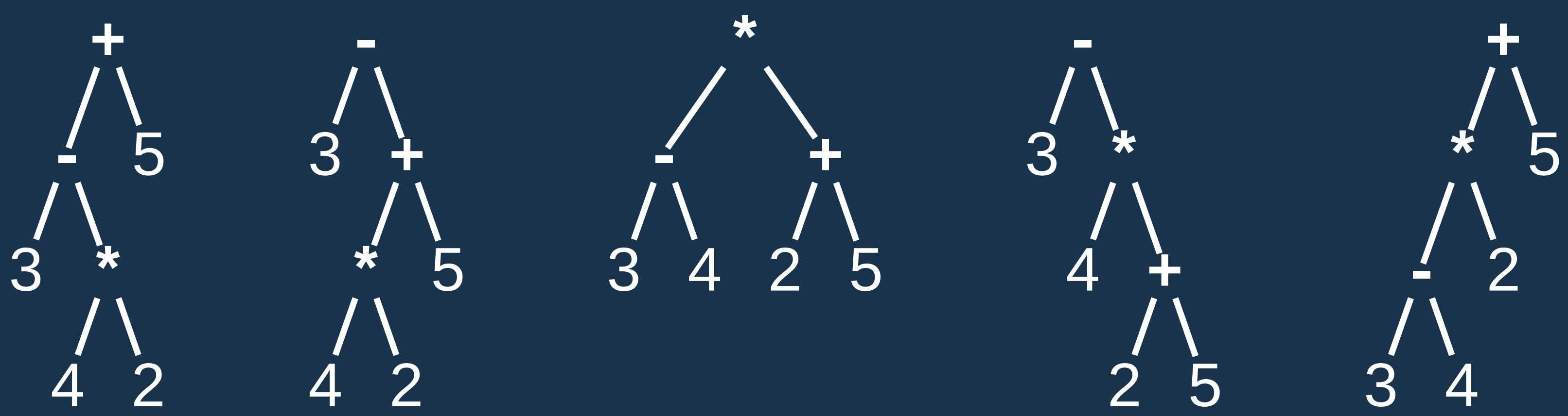
Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing

3 - 4 * 2 + 5

with the grammar

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



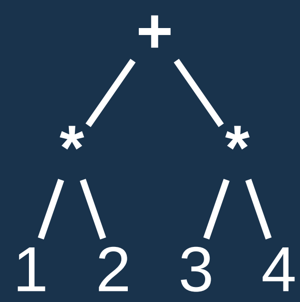
Operator Precedence

Defines how “sticky” an operator is.

`1 * 2 + 3 * 4`

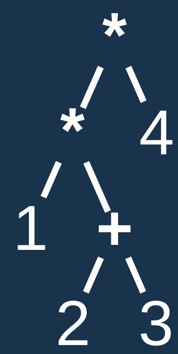
`*` at higher precedence than **`+`**:

`(1 * 2) + (3 * 4)`



`+` at higher precedence than **`*`**:

`1 * (2 + 3) * 4`

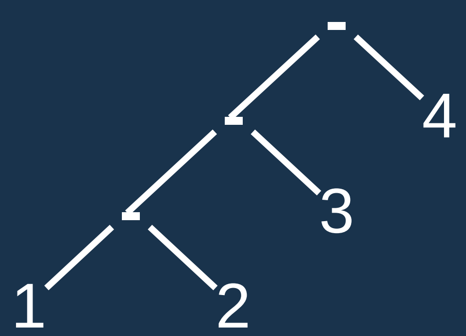


Associativity

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

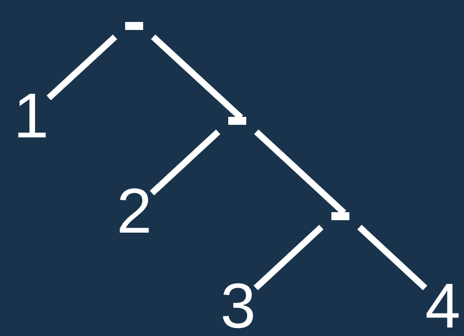
Most operators are left-associative

`1 - 2 - 3 - 4`



((1 - 2) - 3) - 4

left associative



1 - (2 - (3 - 4))

right associative

Actions

Simple languages can be interpreted with parser actions.

```
class CalcParser extends Parser;

expr returns [int r] { int a; r=0; }
 : r=mexpr ("+" a=mexpr { r += a; } )* EOF ;

mexpr returns [int r] { int a; r=0; }
 : r=atom ("*" a=atom { r *= a; } )* ;

atom returns [int r] { r=0; }
 : i:INT
 { r = Integer.parseInt(i.getText()); } ;
```

Object Lifetimes

The objects considered here are regions in memory.

Three principal storage allocation mechanisms:

1. Static

   Objects created when program is compliled, persists throughout run

2. Stack

   Objects created/destroyed in last-in, first-out order. Usually associated with function calls.

3. Heap

   Objects created/deleted in any order, possibly with automatic garbage collection.

Static vs. Dynamic Scope

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

  procedure seta;    begin a := 1 end

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

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

Function Name Overloading

C++ and Java allow functions/methods to be overloaded.

```
int foo();
int foo(int a);   // OK: different # of args
float foo();      // Error: only return type
int foo(float a); // OK: different arg types
```

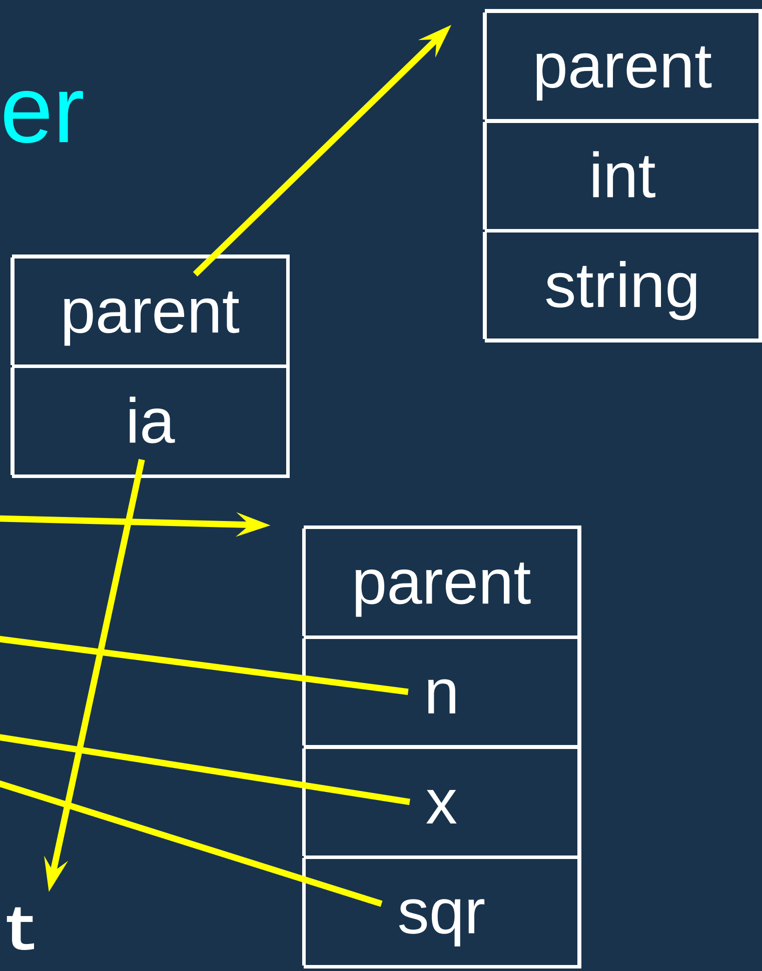
Useful when doing the same thing many different ways:

```
int add(int a, int b);
float add(float a, float b);

void print(int a);
void print(float a);
void print(char *s);
```

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
```



C’s Type System: Structs and Unions

Structures: each field has own storage

```
struct box {
  int x, y, h, w;
  char *name;
};
```

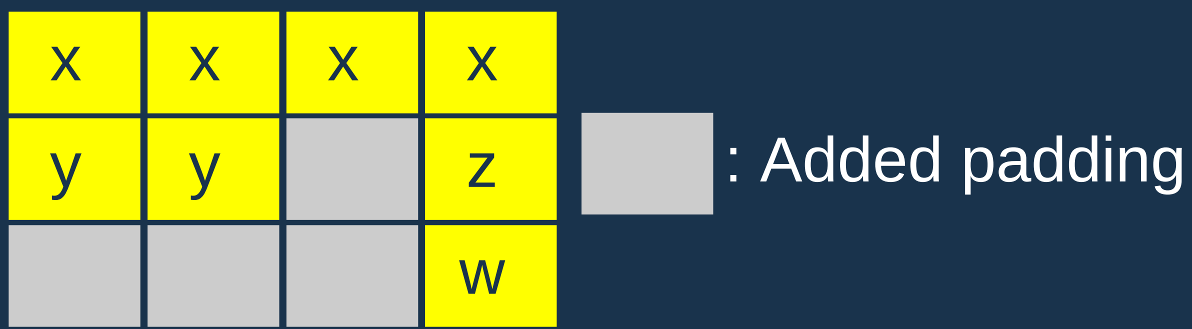
Unions: fields share same memory

```
union token {
  int i;
  double d;
  char *s;
};
```

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 */
};
```



Polymorphism: C++ Templates

```
template <class T> void sort(T a[], int n)
{
  int i, j;
  for ( i = 0 ; i < n-1 ; i++ )
    for ( j = i + 1 ; j < n ; j++ )
      if (a[j] < a[i]) {
        T tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
}
int a[10];
sort<int>(a, 10);
```

Name vs. Structural Equivalence

```
let
  type a = { x: int, y: int }
  type b = { x: int, y: int }
  var i : a := a { x = 1, y = 2 }
  var j : b := b { x = 0, y = 0 }
in
  i := j
end
```

Not legal because **a** and **b** are considered distinct types.

Three Attributes of OO Langauges

1. Encapsulation

   Hides data and procedures from other parts of the program.

2. Inheritance

   Creates new components by refining existing ones.

3. Dynamic Method Dispatch

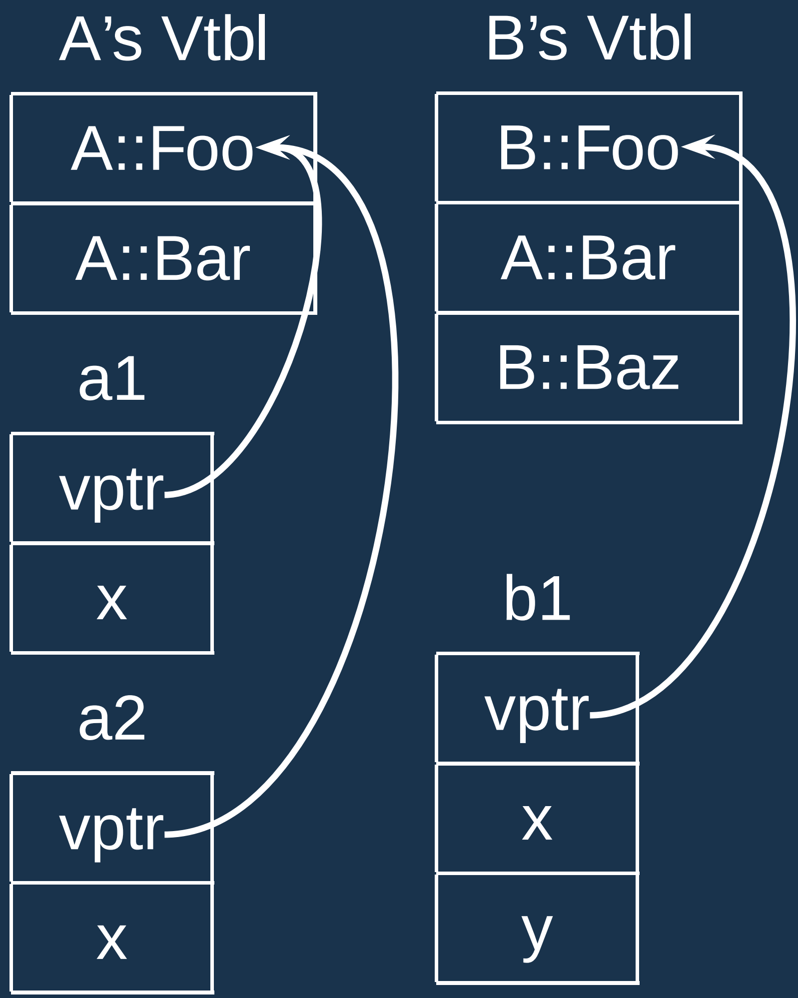
   The ability for a newly-refined object to display new behavior in an existing context.

Virtual Functions

The Trick: Add a "virtual table" pointer to each object.

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};

A a1, a2; B b1;
```



Ordering Within Expressions

What code does a compiler generate for

```
a = b + c + d;
```

Most likely something like

```
tmp = b + c;
a = tmp + d;
```

(Assumes left-to-right evaluation of expressions.)

Misbehaving Floating-Point Numbers

1e20 + 1e-20 = 1e20

1e-20 $\ll$ 1e20

(1 + 9e-7) + 9e-7 $\neq$ 1 + (9e-7 + 9e-7)

9e-7 $\ll$ 1, so it is discarded, however, 1.8e-6 is large enough

$1.00001(1.000001 - 1) \neq 1.00001 \cdot 1.000001 - 1.00001 \cdot 1$

$1.00001 \cdot 1.000001 = 1.00001100001$ requires too much intermediate precision.

Gotos vs. Structured Programming

A typical use of a goto is building a loop. In BASIC:

```
10 print I
20 I = I + 1
30 IF I < 10 GOTO 10
```

A cleaner version in C using structured control flow:

```
do {
  printf("%d\n", i);
  i = i + 1;
} while ( i < 10 )
```

An even better version

```
for (i = 0 ; i < 10 ; i++) printf("%d\n", i);
```

Changing Loop Indices

Most languages prohibit changing the index within a loop.

(Algol 68, Pascal, Ada, FORTRAN 77 and 90, Modula-3)

But C, C++, and Java allow it.

Why would a language bother to restrict this?

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:
```

Tail-Recursion and Iteration

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

Can be rewritten into:

```
int gcd(int a, int b) {
start:
  if ( a==b ) return a;
  else if ( a > b ) a = a-b; goto start;
  else b = b-a; goto start;
}
```

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

setjmp/longjmp Behavior and Usage

```c
#include <setjmp.h>
jmp_buf closure;  /* address, stack */
void top(void) {
  switch (setjmp(closure)) {
    case 0: child(); break;
    case 1: /* longjmp called */ break;
} }
void child() { child2(); }
void child2() { longjmp(closure, 1); }
```

Java's Finally

```
class E extends Exception {}

class Foo {
  public static void main(String[] args)
  { p(1); foo(args[0]); p(5); }

  static void foo(String s) {
    try {
      if (s.equals("a")) throw new E();
      if (s.equals("b")) return;
      p(2);
    } catch (E e) { p(3); }
    finally { p(4); } // Always executed
  }

  static void p(int v) { System.out.println(v); }
}
```

a	b	c
1	1	1
		2
3		
4	4	4
5	5	5

Call-By-Value

The default in C

```
void foo(int x) {
  x = x + 10; // Does not change y
  printf("%d ", x);
}

void main() {
  int y = 0;
  foo();
  printf("%d ", y);
}
```

Prints “10 0”

Call-By-Reference

C++ references simplify the syntax

```
void swap(int &x, int &y) {
  int tmp = x;
  x = y;
  y = tmp;
}

void main() {
  int x = 2, y = 3;
  swap(x, y);      // Works
}
```

Pass by Value/Result

Ada has copy in/copy out semantics.

```
procedure foo(a : in integer,
              b : out integer,
              c : in out integer) in
begin
  c = c + a;
  b = a + 2;
  a = a + 1;
end foo;

x, y, z : integer;

x := 1; z := 5;
foo(x,y,z);
-- x = 1    unchanged
-- y = 3    copied from x
-- z = 6    copied then out
```

Pass-by-name

Caller passes a “thunk” that evaluates the parameter each time it’s referenced.

```
int x = 0;

void foo() { return x; }

int bar(int a) {
  x++;
  return a;
}
```

For **`bar(foo())`**, pass-by-name returns 1. Pass-by-value returns 0.

A Long K's Journey into Byte†

Compiler front end	Source code ↓ Parser/Semantic Analysis AST
Compiler back end	↓ Intermediate code generation IR ↓ Optimization Assembly Code
Assembler	↓ Assemble Relocatable Object Code
Linker	↓ Link Executable
Loader	↓ Relocate In-memory image

†Apologies to O'Neill

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
```

Typical Optimizations

Folding constant expressions

$1+3 \longrightarrow 4$

Removing dead code

if (0) – . . . ″ $\longrightarrow$ nothing

Moving variables from memory to registers

```
ld   [%fp+68], %i1
sub  %i0, %i1, %i0   → sub   %o1, %o0, %o1
st   %i0, [%fp+72]
```

Removing unnecessary data movement

Filling branch delay slots (Pipelined RISC processors)

Common subexpression elimination;

Assembly Code

Most compilers produce assembly code: easier to debug than binary files.

```
! gcd on the SPARC
gcd:
     cmp    %o0, %o1
     be     .LL8
     nop
.LL9:
     ble,a .LL2
     sub    %o1, %o0, %o1
     sub    %o0, %o1, %o0
.LL2:
     cmp    %o0, %o1
     bne    .LL9
     nop
.LL8:
     retl
     nop
```

Comment

Operand (a register)

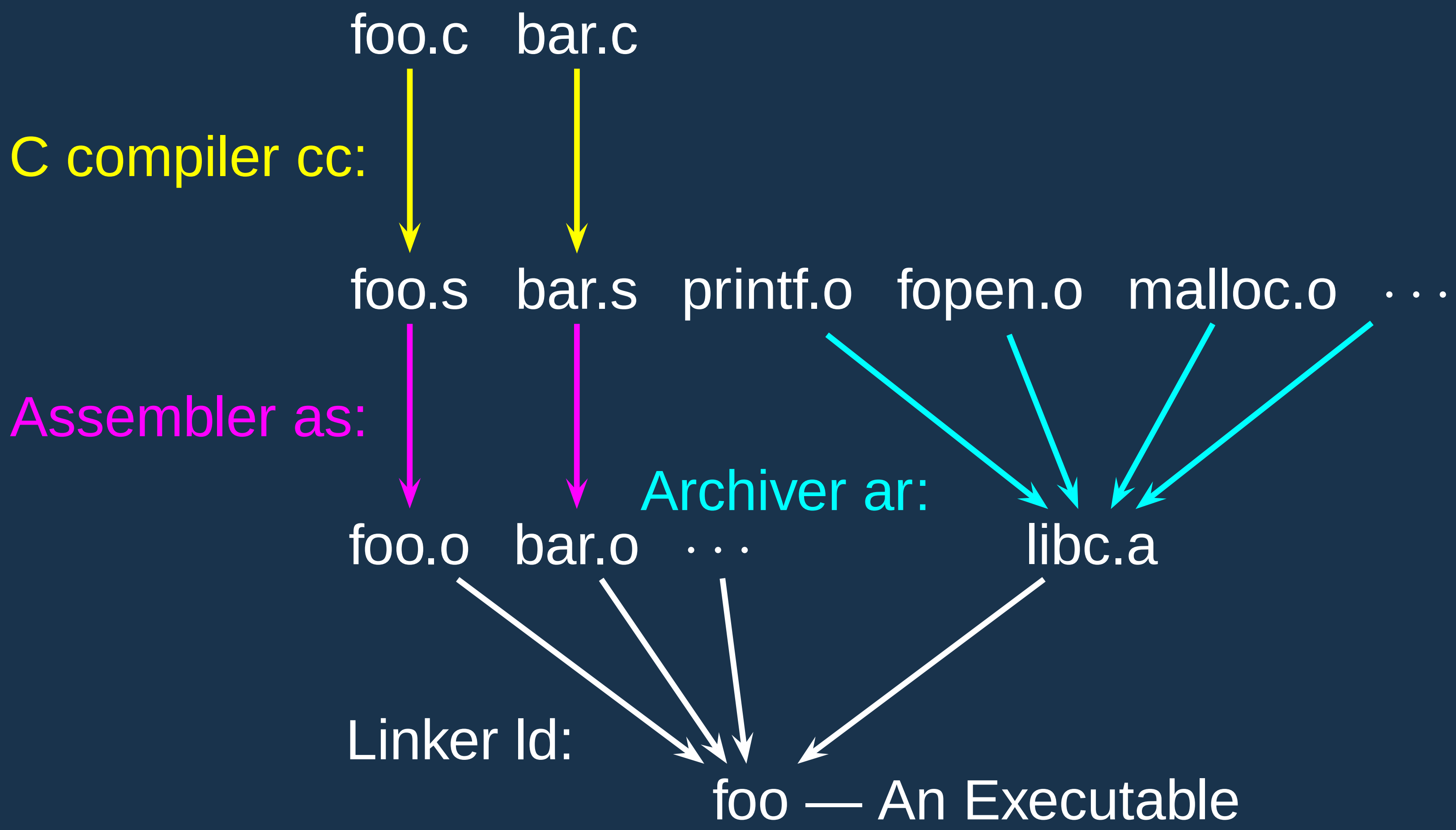
Opcode

Label

Conditional branch to a label

No operation

Separate Compilation



Shared Libraries

Problem fundamentally is that each program may need to see different libraries each at a different address.

Prolog Searching

```
> ~/tmp/beta-prolog/bp
Beta-Prolog Version 1.2 (C) 1990-1994.
| ?- [user].
|:techer(stephen).
|:techer(todd).
|:nerd(X) :- techer(X).
|:^D
yes
| ?- nerd(X).
X = stephen?;
X = todd?;
no
| ?-
```

Prolog: 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

Prolog Search Algorithm: Order Matters

search(goal g, variables e) In the order they appear

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

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

if successful,

In the order they appear

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

$e =$ search(t_k, e)

if all successful, return e

return **no**

Functional Programming

Referential Transparency

- No side-effects; no global data
- Every expression denotes a single value

Recursion

```
dec sum : num -> num ;
--- sum(n) <= if n = 0 then 0
              else sum(n - 1) + n;
```

The Lambda Calculus

Church's alternative to Alan Turning's tape machines.

$(\lambda x.x + y)3 = 3 + y$

Church-Rosser Theorem:

> If an expression can be reduced in two different ways to two normal forms, these forms are the same.

"All roads lead to Rome"

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;
}                     }
```

Concurrency Schemes Compared

	Scheduler	Fair	Cost
Coroutines	Program	No	Low
Cooperative Multitasking	Program/OS	No	Medium
Multiprogramming	OS	No	Medium
Preemptive Multitasking	OS	Yes	High

Java’s Thread Basics

Creating a thread:

```
class MyThread extends Thread {
  public void run() {
    /* thread body */
  }
}

MyThread mt = new MyThread(); // Create the thread
mt.start();   // Invoke run, return immediately
```

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`

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.

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;
}
```