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

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

What the Compiler Sees

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

Lexical Analysis Gives Tokens

```
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

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

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  
    jmp L5  
    L4: sub b, b, a  
    jmp .L8  
    L5: jmp L0  
    L1: ret 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  
        jle .L5  
        % while (a >= b)  
        subl %edx, %eax  
        jmp .L8  
    .L5: subl %eax, %edx  
        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.

**Nondeterministic 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**

Building an NFA for the regular expression

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

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.

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.

Ambiguous Grammars

A grammar can easily be ambiguous. Consider parsing $3 - 4 * 2 + 5$ with the grammar:

$$e ightarrow e + e | e - e | e * e | e / e$$

Fixing Ambiguous Grammars

Original ANTLR grammar specification:

```plaintext
expr : 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:

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

term : term '*' atom |
     term '/' atom |
     atom ;

atom : NUMBER ;
```

Still ambiguous: associativity not defined

Assigning Associativity

Make one side or the other the next level of precedence:

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

term : term '*' atom |
     term '/' atom |
     atom ;

atom : NUMBER ;
```

A Top-Down Parser

```plaintext
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();
        default : exit(1); // Infinite Recursion
    }
}
```

Writing LL(k) Grammars

Cannot have left-recursion:

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

becomes:

```plaintext
expr : expr '+' term |
     expr | term ;
```
**Writing LL(1) Grammars**

Cannot have common prefixes

```plaintext
expr : ID ' ' expr ' '
    | ID '=' expr
```

becomes

AST `expr()` –

```plaintext
switch (next-token) –
case ID : match(ID); match(' '); expr(); match(' ');
case ID : match(ID); match('='); expr();
```

**Eliminating Common Prefixes**

Consolidate common prefixes:

```plaintext
expr : expr '+' term
    | expr '-' term
    | term
```

becomes

```plaintext
expr : expr ('+' term | '-' term )
    | term
```

**Eliminating Left Recursion**

Understand the recursion and add tail rules

```plaintext
expr : expr ('+' term | '-' term )
    | term
```

becomes

```plaintext
expr: term exprt ;
exprt : '+' term exprt
    | '-' term exprt
    | /* nothing */
```

**Bottom-up Parsing**

**Rightmost Derivation**

```
1: e -> t + e
2: e -> t
3: t -> Id * t
4: t -> Id

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

```
e
  t + e
  t + t
  Id + Id
  Id + Id
```

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

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: e -> t + e
2: e -> t
3: t -> Id * t
4: t -> Id

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

```
e
  t + e
  t + t
  Id + Id
  Id + Id
```

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

An oracle tells what to do

**LR Parsing**

```
1: e -> t + e
2: e -> t
3: t -> Id * t
4: t -> Id

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

```
e
  t + e
  t + t
  Id + Id
  Id + Id
```

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

An oracle tells what to do
LR Parsing

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 → t + e
2: e → t
3: t → Id * t
4: t → Id

action goto

id + * $ 1 2
1 r4 r4 s3 r4
2 r2 s4 r2 r2
3 s1 5 2
4 s1 6 2
5 r3 r3 r3 r3
6 r1 r1 r1 r1
7 acc

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

e' → e

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

Names, Objects, and Bindings

Activation Records

Nested Subroutines in Pascal

Symbol Tables in Tiger

int A() {
  int x;
  B();
}
int B() {
  int y;
  C();
}
int C() {
  int z;
}

procedure A;
  procedure B;
    procedure C;
    begin .. end
  procedure D;
    begin C end
    begin D end
  procedure E;
    begin B end
    begin E end

int 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; } // static, dynamic
    return bar;
}
int main() {
    shallow 1 2
    deep 1 1
    ifunc f = foo();
    int a = 2;
    return (*f)();
}
void a(int i, void (*p)()) {
    void b() { printf("%d", i); }
    if (i=1) a(2,b) else (*p)();
}
void q() {}
int main() {
    static
    a(1,q);
    b
    i = 2, p = b
}

Layout of Records and Unions

Modern processors have byte-addressable memory.

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:

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

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

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

Layout of Records and Unions

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

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

SPARC prohibits unaligned accesses.
MIPS has special unaligned load/store instructions.
x86, 68k run more slowly with unaligned accesses.

Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

void foo() {
    return address ← FP
    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;
}

Static Semantic Analysis

Variables remain constant offset from frame pointer.

x86, 68k run more slowly with unaligned accesses.
Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```
if i 3 "This" /* valid */
#all123 /* 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 */
```

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.

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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
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;
  printf("%d ", c);
  total *= c;
}
```

Normal: arguments evaluated when used.

```
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;
  printf("%d ", c);
  total *= c;
}
```

Result: 1 2 3

Applicative- vs. and Normal-Order Evaluation

```
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 ) );
```

Applicative: arguments evaluated before function is called.

```
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 ) );
```

Normal: arguments evaluated when used.

```
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 ) );
```

Result: 1 2 3

Applicative- and Normal-Order Evaluation

```
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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
Applicative: arguments evaluated before function is called.
```

Normal: arguments evaluated when used.

```
Applicative: arguments evaluated before function is called.
```

Result: 1 2 3

Applicative- vs. and Normal-Order Evaluation

```
Applicative- vs. 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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
Applicative: arguments evaluated before function is called.
```

Normal: arguments evaluated when used.

```
Applicative: arguments evaluated before function is called.
```

Result: 1 2 3

Applicative- and Normal-Order Evaluation

```
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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
Applicative: arguments evaluated before function is called.
```

Normal: arguments evaluated when used.

```
Applicative: arguments evaluated before function is called.
```

Result: 1 2 3

Applicative- vs. and Normal-Order Evaluation

```
Applicative- vs. 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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
Applicative: arguments evaluated before function is called.
```

Normal: arguments evaluated when used.

```
Applicative: arguments evaluated before function is called.
```

Result: 1 2 3

Applicative- and Normal-Order Evaluation

```
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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
Applicative: arguments evaluated before function is called.
```

Normal: arguments evaluated when used.

```
Applicative: arguments evaluated before function is called.
```

Result: 1 2 3

Applicative- vs. and Normal-Order Evaluation

```
Applicative- vs. 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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
Applicative: arguments evaluated before function is called.
```

Normal: arguments evaluated when used.

```
Applicative: arguments evaluated before function is called.
```

Result: 1 2 3

Applicative- and Normal-Order Evaluation

```
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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
Applicative: arguments evaluated before function is called.
```

Normal: arguments evaluated when used.

```
Applicative: arguments evaluated before function is called.
```

Result: 1 2 3

Applicative- vs. and Normal-Order Evaluation

```
Applicative- vs. 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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
Applicative: arguments evaluated before function is called.
```

Normal: arguments evaluated when used.

```
Applicative: arguments evaluated before function is called.
```

Result: 1 2 3

Applicative- and Normal-Order Evaluation

```
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;
  printf("%d ", c);
  total *= c;
}
```

Applicative: arguments evaluated before function is called.

```
Applicative: arguments evaluated before function is called.
```

Normal: arguments evaluated when used.

```
Applicative: arguments evaluated before function is called.
```

Result: 1 2 3
Prolog

All Caltech graduates are nerds. \texttt{nerd}(X) :- \texttt{techer}(X).

Stephen is a Caltech graduate. \texttt{techer}(stephen).

Is Stephen a nerd? \texttt{?- nerd}(stephen).
\texttt{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:

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

What's a structure? Whatever you like.

A predicate \texttt{nerd}(stephen)

A relationship \texttt{teaches}(edwards, cs4115)

A data structure \texttt{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:

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

The Searching Algorithm

```
search(goal \texttt{g}, variables \texttt{e})
```

for each clause \texttt{h} :- \texttt{t}_1, \ldots, \texttt{t}_n in the database

\texttt{e} = \texttt{unify}(\texttt{g}, \texttt{h}, \texttt{e})

if successful,
  for each term \texttt{t}_1, \ldots, \texttt{t}_n,
  \texttt{e} = \texttt{search} (\texttt{t}_k, \texttt{e})
  if all successful, return \texttt{e}
return \texttt{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

\texttt{?- path(a, a)}.

\texttt{path(a,a)=path(X,Y)}

\texttt{Subgoal}

\texttt{path(a,Z)=path(X,Y)}

\texttt{X=a}
\texttt{Y=a}

\texttt{Unify}

\texttt{X=a}
\texttt{Y=a}

Like LL(k) grammars.

Concurrent

char c;
void scan() {
  char buf[10];
  c = ‘s’;
  transfer parse;
  buf[0] = c;
  c = ‘a’;
  transfer parse;
  buf[1] = c;
  c = ‘e’;
  transfer parse;
  buf[2] = c;
}
transfer parse;
}

Coroutines
Cooperative Multitasking

Typical MacOS < 10 or Windows < 95 program:

```java
void main() {
    Event e;
    while ( (e = get_next_event()) != QUIT ) {
        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

App 1
Disk read
App 2
App 1 read complete
App 3
Disk write
App 2
App 1 read complete
App 1
App 3 write complete

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 sees old values

Thread 1 runs before Thread 2

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

Thread 2
1. a.field1 = 1
2. a.field2 = 2

Thread 1 sees new values

Thread 1 runs after Thread 2

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

Thread 2
1. a.field1 = 1
2. a.field2 = 2

Thread 1 sees inconsistent values

Execution of Thread 1 interrupts execution of Thread 2

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

Thread 2
1. a.field1 = 1
4. a.field2 = 2

Synchronized Methods

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

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

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

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

```java
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

Building a Blocking Buffer

class OnePlace {
    El value;

    public synchronized void write(El e) { .. }  // Acquire lock on obj
    public synchronized El read() { .. }  // Relinquish lock on obj
}

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

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
- val it = 25 : int
- 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()

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.

Functional Programming

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

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

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:  \( \text{fn x => 2 * x} \) \hfill \text{The Lambda Calculus:} \( \lambda x. 2 \times x \)

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.
Reduction Order

The order in which you reduce things can matter.

\[(\lambda x.\lambda y.y) ( (\lambda z.z) (\lambda z.z) )\]

We could choose to reduce one of two things, either

\[(\lambda z.z) (\lambda z.z)\]
or the whole thing

\[(\lambda x.\lambda y.y) ( (\lambda z.z) (\lambda z.z) )\]

Reduction Order

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

\[(\lambda z.z) (\lambda z.z)\]

So always reducing it does not terminate.

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

\[(\lambda x.\lambda y.y) ( (\lambda z.z) (\lambda z.z) )\]

\[\lambda y.y\]

Applicative vs. Normal Order

Applicative order reduction: Always reduce the leftmost innermost redex.

Normative order reduction: Always reduce the leftmost outermost redex.

For \((\lambda x.\lambda y.y) ( (\lambda z.z) (\lambda z.z) )\), applicative order reduction never terminated but normative order did.

Normal Form

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

Thus,

\[\lambda y.y\]

is the normal form of

\[(\lambda x.\lambda y.y) ( (\lambda z.z) (\lambda z.z) )\]

The Church-Rosser Theorems

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

“Reduction in any way can eventually produce the same result.”

If \(E_1 \to E_2\), and \(E_2\) is in 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.