Review for the Final

COMS W4115
Prof. Stephen A. Edwards
Fall 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
Scripting Languages
Scanning and Parsing
Regular Expressions
Context-Free Grammars
Top-down Parsing
Bottom-up Parsing
ASTs

Topics (2)

Name, Scope, and Bindings
Types
Control-flow constructs
Code Generation
Logic Programming: Prolog
Functional Programming: ML and the Lambda Calculus

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;
}
```

Lexical Analysis Gives Tokens

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

Parsing Gives an AST

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

Semantic Analysis Resolves Symbols

Text file is a sequence of characters

A stream of tokens. Whitespace, comments removed.

Abstract syntax tree built from parsing rules.

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  
    % a -= b  
L4: sub b, b, a  
    jmp L8  
    % b -= a  
L5: jmp L0  
    % 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 %edi, %eax  
    % Load a from stack  
    movl 12(%ebp), %edx  
    % Load b from stack  
    .L8:  
    cmp %edx, %eax  
    je .L3  
    % while (a != b)  
    jle .L5  
    % if (a < b)  
    subl %edx, %eax  
    jmp .L8  
    % a -= b  
    .L5:  
    subl %eax, %edx  
    jmp .L8  
    % b -= a  
    .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  
(uvε)m(aε)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.

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 → 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 '+' 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

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();
        case 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

\[
expr : \text{ID} \left( \text{expr} \right)
| \text{ID} \left( \text{expr} \right)
| \text{ID} \left( \text{expr} \right)
| \text{ID} \left( \text{expr} \right)
| \text{ID} \left( \text{expr} \right)
| \text{ID} \left( \text{expr} \right)
\]

becomes

AST expr() –

switch (next-token) –

```java
case ID : match(ID); match('('); expr(); match('));
```  

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

\[
expr : expr '+' term
| expr '-' term
| term
\]

becomes

\[
expr : term
| exprt
| term
\]

| termt : '+' term
| '-' term
| /* nothing */

Bottom-up Parsing

Rightmost Derivation

1: \(e \rightarrow t + e\)
2: \(e \rightarrow t\)
3: \(t \rightarrow \text{Id} \ast t\)
4: \(t \rightarrow \text{Id}\)

A rightmost derivation for \(\text{Id} \ast \text{Id} + \text{Id}\):

\[
\begin{align*}
\text{e} & \rightarrow t + e \\
& \rightarrow t \\
& \rightarrow \text{Id} \ast t \\
& \rightarrow \text{Id} \\
\end{align*}
\]

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

Shift-reduce Parsing

1: \(e \rightarrow t + e\)  
2: \(e \rightarrow t\)  
3: \(t \rightarrow \text{Id} \ast t\)  
4: \(t \rightarrow \text{Id}\)  

Scan input left-to-right, looking for handles.
An oracle tells what to do

LR Parsing

1: \(e \rightarrow t + e\)  
2: \(e \rightarrow t\)  
3: \(t \rightarrow \text{Id} \ast t\)  
4: \(t \rightarrow \text{Id}\)  

Action is reduce with rule \(4 (t \rightarrow \text{Id})\). 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 s1  |   7  | 2    
1 r4 r4 s3 r4 | 1    | 2    
2 r2 s4 r2 r2 | 1    | 2    
3 s1  | 5    | 2    
4 s1  | 6    | 2    
5 r3 r3 r3 r3 | 6    | 2    
6 r1 r1 r1 r1 | 7    | acc  
7 acc |      |      
```

1: \(e \rightarrow t + e\)  
2: \(e \rightarrow t\)  
3: \(t \rightarrow \text{Id} \ast t\)  
4: \(t \rightarrow \text{Id}\)  

Action is reduce with rule \(4 (t \rightarrow \text{Id})\). 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 s1  |   7  | 2    
1 r4 r4 s3 r4 | 1    | 2    
2 r2 s4 r2 r2 | 1    | 2    
3 s1  | 5    | 2    
4 s1  | 6    | 2    
5 r3 r3 r3 r3 | 6    | 2    
6 r1 r1 r1 r1 | 7    | acc  
7 acc |      |      
```
### LR Parsing

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>r4 r4 s3 r4</td>
<td>reduce w/ 4</td>
</tr>
<tr>
<td>2</td>
<td>t r2 s4 r2</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>4</td>
<td>s1</td>
<td>reduce w/ 1</td>
</tr>
<tr>
<td>6</td>
<td>t r3 r3 r3</td>
<td>shift, goto 4</td>
</tr>
<tr>
<td>7</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

### 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. $e \rightarrow t + e$
2. $e \rightarrow t$
3. $t \rightarrow \text{id} \ast t$
4. $t \rightarrow \text{id}$

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

- $e'$ is a placeholder.
- 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 $t$.

### Names, Objects, and Bindings

**Object1**

**Object2**

**Object3**

**Object4**

**Names, Objects, and Bindings**

- **Binding**
  - **Name1**
  - **Name2**
  - **Name3**
  - **Name4**

### Activation Records

<table>
<thead>
<tr>
<th>Argument 2</th>
<th>Argument 1</th>
<th>Frame Pointer</th>
<th>Return Address</th>
<th>Frame Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>B's variables</td>
<td>A's variables</td>
<td>frame pointer</td>
<td>return address</td>
<td>old frame pointer</td>
</tr>
<tr>
<td>Local variables</td>
<td>temporaries/arguments</td>
<td>stack pointer</td>
<td>stack pointer</td>
<td>growth of stack</td>
</tr>
</tbody>
</table>

### Nested Subroutines in Pascal

```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)();
}

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() {
    static
    a(1,q);
    i = 1, p = q
    b reference
    a(2,b)
    i = 2, p = b
    b

Layout of Records and Unions

Modern processors have byte-addressable memory.

<table>
<thead>
<tr>
<th>16-bit integer:</th>
<th>32-bit integer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>3 2 1 0</td>
</tr>
</tbody>
</table>

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

Layout of Records and Unions

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

<table>
<thead>
<tr>
<th>3 2 1 0</th>
<th>7 6 5 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 10 9 8</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>3 2 1 0</th>
<th>7 6 5 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 10 9 8</td>
<td></td>
</tr>
</tbody>
</table>

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.
Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

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

Syntactic analysis: Makes sure tokens appear in correct order

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

Semantic analysis: Makes sure program is consistent

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

Implementing multi-way branches

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

Obvious way:

```plaintext
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

```plaintext
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.

```plaintext
#define p(x) (printf("%d ", x))
#define q(a, b, c) total = (a), \n  printf("%d ", (b)), \n  total += (c)
```

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

Prints 1 2 3.

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

Mid-test Loops

```plaintext
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;
```

Applicative- and Normal-Order Evaluation

```plaintext
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) );
```

What is printed by

```
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.

```plaintext
int p(int i) { printf("%d ", i); return i; }
int q(int a, int b, int c) {
  q( p(1), p(2), p(3) );
  printf(5 6 7. It will print one of
```
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

Order can cause Infinite Recursion

Consider the query
?- path(a, a).

Like LL(k) grammars.

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

Unification

The `=` operator checks whether two structures unify:

```
?- a = a.
   yes
?- a = b.
   no
?- 5.3 = a.
   no
?- 5.3 = x.
   x = 5.3?
?- foo(a, X) = foo(X, b).
   X = a?
?- foo(a, X) = foo(X, a).
   X = a?
?- foo(X, b) = foo(a, Y).
   X = a, Y = b?
?- foo(a, X, a) = foo(b, a, c).
   X = a?
```

The Searching Algorithm

```
search(goal g, variables e) for each clause h :- t1, .. , tn in the database
e = unify(g, h, e)
   if successful,
      for each term t1, .. , tn,
      e = search(tk, e)
   if all successful, return e
return no
```

Functional Programming

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.

- \( \text{fun max } a \ b = \text{if } a > b \text{ then } a \text{ else } b; \)
- \( \text{val max} = \text{fn} : \text{int} -> \text{int} -> \text{int} \)
- \( \text{val max5} = \text{max } 5; \)
- \( \text{val it = 5 : int} \)
- \( \text{val max5} = \text{fn} : \text{int} -> \text{int} \)
- \( \text{val max5 } 4; \)
- \( \text{val it = 5 : int} \)
- \( \text{val max5 } 6; \)
- \( \text{val it = 6 : int} \)
- \( \text{val max5 } 6; \)
- \( \text{val it = 6 : int} \)
- \( \text{val max5 } 6; \)
- \( \text{val it = 6 : int} \)

Recursion

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

So use recursion:

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

Reduce

Another popular functional language construct:

\[
\text{fun reduce } (f, z, \text{nil}) = z \quad \mid \quad \text{reduce } (f, z, \text{h::t}) = f(\text{h, reduce } (f, z, \text{t})) ;
\]
- \( \text{if } f \text{ is } "\text{--", reduce } (f, z, \text{a::b::c}) \text{ is } a - (b - (c - z)) \)
- \( \text{reduce } (\text{fn } (x,y) => x - y, 0, [1,5]); \)
- \( \text{val it = } -4 : \text{int} \)
- \( \text{reduce } (\text{fn } (x,y) => x - y, 2, [10,2,1]); \)
- \( \text{val it = } 7 : \text{int} \)

But why always name functions?

- \( \text{map } (\text{fn } x => x + 5, [10,11,12]); \)
- \( \text{val it = } [15,16,17] : \text{int list} \)
- \( \text{val add5 } = \text{fn } x => x + 5; \)
- \( \text{val add5 } 10; \)
- \( \text{val it = } 15 : \text{int} \)

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:

- \( \text{fun map } (f, []) = [] \)
- \( \mid \text{map } (f, [\text{h}]) = f(\text{h}) : \text{map } (f, []) \)
- \( \text{Pattern matching is order-sensitive. This gives an error.} \)
- \( \text{fun map } (f, [\text{h}]) = f(\text{hd}) : \text{map } (f, []) \)
- \( \mid \text{map } (f, []) = []; \)

Pattern Matching

More fancy binding

- \( \text{fun map } (_, []) = [] \)
- \( \mid \text{map } (f, \text{h :: t}) = f(\text{h}) : \text{map } (f, \text{t}) \)
- \( "_" \text{ matches anything} \)
- \( \text{h :: t} \text{ matches a list, binding } \text{h} \text{ to the head and } \text{t} \text{ to the tail.} \)

Pattern Matching

More fancy binding

\[
\text{fun map } ((\_), []) = [] \quad \mid \text{map } (f, \text{h :: t}) = f(\text{h}) : \text{map } (f, \text{t}) ;
\]
- \( "_" \text{ matches anything} \)
- \( \text{h :: t} \text{ matches a list, binding } \text{h} \text{ to the head and } \text{t} \text{ to the tail.} \)

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:
- \( \text{fn } x => 2 * x; \)

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:

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

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:

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

$\lambda y. y$

Evaluating Lambda Expressions

We need a reduction rule to handle $\lambda$s:

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

This is called $\beta$-reduction.

The formal parameter may be used several times:

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

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

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.

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

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

Not everything has a normal form

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