Review for the Final
COMS W4115
Prof. Stephen A. Edwards
Fall 2007
Columbia University
Department of Computer Science

The Final
70 minutes
4–5 problems
Closed book
One single-sided 8.5 × 11 sheet of notes of your own devising
Comprehensive: Anything discussed in class is fair game
Little, if any, programming.
Ability to write 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
Name, Scope, and Bindings
Control-flow constructs

Topics 2
Types
Static Semantic Analysis
Code Generation
Functional Programming (ML, Lambda Calculus)
Logic Programming (Prolog)

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.

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

Text file is a sequence of characters

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
L4: sub b, b, a
L5: jmp L0
L1: ret a

Idealized assembly language w/ infinite registers

Generation of 80386 Assembly

gcd: pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
movl 12(%ebp), %edx
.L8: cmpl %edx, %eax
je .L3 % while (a != b)
jle .L5 % if (a < b)
subl %edx, %eax % a -= b
jmp .L8
.L5: subl %eax, %edx % b -= a
jmp .L8
.L3: leave % Restore SP, BP
ret

Scanning and Automata

Describing Tokens

Alphabet: A finite set of symbols
Examples: \{ 0, 1 \}, \{ A, B, C, ..., Z \}, ASCII, Unicode

String: A finite sequence of symbols from an alphabet
Examples: \( \epsilon \) (the empty string), Stephen, \( \alpha \beta \gamma \)

Language: A set of strings over an alphabet
Examples: \( \emptyset \) (the empty language), \{ 1, 11, 111, 1111 \}, all English words, strings that start with a letter followed by any sequence of letters and digits

Operations on Languages

Let \( L = \{ \epsilon, wo \} \), \( M = \{ \text{man, men} \} \)

Concatenation: Strings from one followed by the other
\( LM = \{ \text{man, men, woman, women} \} \)

Union: All strings from each language
\( L \cup M = \{ \epsilon, wo, \text{man, men} \} \)

Kleene Closure: Zero or more concatenations
\( M^* = \{ \epsilon, M, MM, MMM, ... \} = \{ \epsilon, \text{man, men, manman, manmen, mmenmen, ...} \} \)

Regular Expressions over an Alphabet \( \Sigma \)

A standard way to express languages for tokens.
1. \( \epsilon \) is a regular expression that denotes \( \{ \epsilon \} \)
2. If \( a \in \Sigma \), \( a \) is an RE that denotes \( \{ a \} \)
3. If \( r \) and \( s \) denote languages \( L(r) \) and \( L(s) \),
   \( r | s \) denotes \( L(r) \cup L(s) \)
   \( rs \) denotes \( \{ tu : t \in L(r), u \in L(s) \} \)
   \( r^* \) denotes \( \bigcup_{i=0}^{\infty} L^i \) (\( L^0 = \emptyset \) and \( L^i = LL^{i-1} \))

Nondeterministic Finite Automata

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

1. Set of states \( S = \{ A, B, C, D \} \)
2. Set of input symbols \( \Sigma = \{ 0, 1 \} \)
3. Transition function \( \sigma : S \times \Sigma \rightarrow 2^S \)

<table>
<thead>
<tr>
<th>state</th>
<th>( \epsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{ B }</td>
</tr>
<tr>
<td>B</td>
<td>{ A }</td>
</tr>
<tr>
<td>C</td>
<td>{ D }</td>
</tr>
<tr>
<td>D</td>
<td>{ C }</td>
</tr>
</tbody>
</table>

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

The Language induced by an NFA

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

Show that the string "010010" is accepted.

Translating REs into NFAs

Consider the RE \( r \)

\( r^* \)

\( \epsilon \)
Translating REs into NFAs

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

Show that the string "aabb" is accepted.

Simulating NFAs

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

Solution: follow them all and sort it out later.

"Two-stack" NFA simulation algorithm:

1. Initial states: the \(\epsilon\)-closure of the start state
2. For each character \(c\),
   - New states: follow all transitions labeled \(c\)
   - Form the \(\epsilon\)-closure of the current states
3. Accept if any final state is accepting

Simulating an NFA:

- \(\epsilon\)-closure
- \(a\)-path
- \(a\)-path, \(\epsilon\)-closure
- \(aa\)-path
- \(aa\)-path, \(\epsilon\)-closure
- \(aab\)-path
Simulating an NFA: $aab \cdot b$, $\epsilon$-closure

\[ \begin{array}{c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\epsilon & a & b & \epsilon & a & b & \epsilon & a & b & b & \epsilon \\
\end{array} \]

Simulating an NFA: $aabb$.

\[ \begin{array}{c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\epsilon & a & b & \epsilon & a & b & \epsilon & a & b & b & \epsilon \\
\end{array} \]

Simulating an NFA: $aabb$, Done

\[ \begin{array}{c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\epsilon & a & b & \epsilon & a & b & \epsilon & a & b & b & \epsilon \\
\end{array} \]

Deterministic Finite Automata

Restricted form of NFAs:

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

Diffs subtly from the definition used in COMS W3261 (Sipser, Introduction to the Theory of Computation)

Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.

Building a DFA from an NFA

Subset construction algorithm

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

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

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

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

A grammar can easily be ambiguous. Consider parsing

\[3 - 4 \times 2 + 5\]

with the grammar

\[e \rightarrow e + e | e - e | e \times e | e / e\]

Still ambiguous: associativity not defined

Assigning Precedence Levels

Split into multiple rules, one per level

\[\text{expr} : \text{expr} \ '+' \text{term} | \text{expr} \ '-' \text{term} | \text{term} ;\]

\[\text{term} : \text{term} \ '*' \text{atom} | \text{term} \ '/' \text{atom} | \text{atom} ;\]

\[\text{atom} : \text{NUMBER} ;\]

Writing LL(k) Grammars

Cannot have left-recursion

\[\text{expr} : \text{expr} \ '+' \text{term} | \text{term} ;\]

becomes

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

A Top-Down Parser

\[\text{stmt} : 'if' \text{expr} \ 'then' \text{expr} \]

\[| 'while' \text{expr} \ 'do' \text{expr} \]

\[| \text{expr} \ ':=' \text{expr} ;\]

\[\text{expr} : \text{NUMBER} | (\text{expr}) ;\]

\[\text{AST stmt()} {\}

\[\text{switch (next-token)} {\}

\[\text{case 'if' : match("if"); expr(); match("then"); expr();\}

\[\text{case "while" : match("while"); expr(); match("do"); expr();\}

\[\text{case NUMBER or "}" : expr(); match("="); expr();\}

\]

Assigning Associativity

Make one side or the other the next level of precedence

\[\text{expr} : \text{expr} \ '+' \text{term} | \text{term} ;\]

\[\text{term} : \text{term} \ '*' \text{atom} | \text{atom} ;\]

\[\text{atom} : \text{NUMBER} ;\]
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 :  e→t + e
2 :  e→t
3 :  t→Id * t
4 :  t→Id

A rightmost derivation for Id * Id + Id:

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

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

Shift-reduce Parsing

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

```
  stack           input          action
  7              2               goto
  5              3               reduce (4)
  3              5               reduce (3)
  1              6               reduce (2)
  0              7               reduce (1)
```

1. Look at state on top of stack and the next input token
2. To find the next action
3. In this case, shift the token onto the stack and go to state 1.

LR Parsing

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

```
  stack           input          action
  7              2               goto
  5              3               reduce (4)
  3              5               reduce (3)
  1              6               reduce (2)
  0              7               reduce (1)
```

Action is reduce with rule 4 (t→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
  7      2
```

Handles

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.
LR Parsing

1: \[ e \rightarrow t + e \]
2: \[ t \rightarrow e \]
3: \[ t \rightarrow \text{Id} \times t \]
4: \[ t \rightarrow \text{Id} \]

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{Id} \times \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>[ \text{Id} + \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>shift, goto 1</td>
</tr>
<tr>
<td>[ \text{Id} + \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>shift, goto 4</td>
</tr>
<tr>
<td>[ \text{Id} + \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>reduce w/ 4</td>
</tr>
<tr>
<td>[ \text{Id} + \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>reduce w/ 3</td>
</tr>
<tr>
<td>[ \text{Id} + \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>shift, goto 4</td>
</tr>
<tr>
<td>[ \text{Id} + \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>shift, goto 1</td>
</tr>
<tr>
<td>[ \text{Id} + \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>reduce w/ 2</td>
</tr>
<tr>
<td>[ \text{Id} + \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>reduce w/ 1</td>
</tr>
<tr>
<td>[ \text{Id} + \text{Id} \text{S} ]</td>
<td>[ t ]</td>
<td>accept</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: \[ t \rightarrow \text{Id} \times t \]
4: \[ t \rightarrow \text{Id} \]

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

\[ e' \rightarrow \cdot e \]
\[ e \rightarrow \cdot t \]
\[ t \rightarrow \cdot \text{Id} \times t \]
\[ t \rightarrow \cdot \text{Id} \]

The first is a placeholder. These 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

Activation Records

Nested Subroutines in Pascal

Symbol Tables in Tiger

<table>
<thead>
<tr>
<th>parent</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
</tbody>
</table>
**Static Semantic Analysis**

Lexical analysis: Make sure tokens are valid

```c
if i 3 "This" /* valid */
#endif /* invalid */
```

Syntactic analysis: Makes sure tokens appear in correct order

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

Semantic analysis: Makes sure program is consistent

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

---

**Basic paradigm:** recursively check AST nodes.

```
check(1) = int
check(5) = int
```

---

**Implementing multi-way branches**

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

Obvious way:

```c
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;
}
```

If the cases are dense, a branch table is more efficient:

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

```
lables 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- 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)
```

```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

---

**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
```
Modern processors have byte-addressable memory.

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

16-bit integer: 
\[
\begin{array}{c}
0x00 \\
0x01 \\
0x02 \\
0x03 \\
0x04 \\
\end{array}
\]

32-bit integer: 
\[
\begin{array}{c}
0x0000 \\
0x0001 \\
0x0002 \\
0x0003 \\
0x0004 \\
0x0005 \\
0x0006 \\
0x0007 \\
0x0008 \\
0x0009 \\
0x000A \\
0x000B \\
0x000C \\
0x000D \\
0x000E \\
0x000F \\
\end{array}
\]

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

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

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.

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

Local arrays with fixed size are easy to stack.

```
void foo() {
    return address ← FP
    a
    b[0]
    int a;
    int b[10];
    int c;
    b[9]
    c ← FP + 12
}
```

```
return address ← FP
a
b[0]
int a;
int b[n];
int c;

b[n-1]
c ← FP + ?
```

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

As always: add a level of indirection

```
void foo(int n) {
    return address ← FP
    a
    b.ptr
    c
    b[0]
    int a;
    int b[n];
    int c;
    b[n-1]
}
```

Variables remain constant offset from frame pointer.
Basic Blocks

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
```

A more complex function

```
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
    val it = 5 : int
```

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

```
val it = [3,4,5] : int list
```

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
```

```
val it = 5 : int
```

Fun with recursion

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

```
val it = 7 : 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
```

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

```
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 \( \sim \), \( reduce(f,z,a::b::c) \) is \( a - (b - (c - z)) \)
```

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

Consider
- fun find1(a,b) =
  if b then true else (a = 1);
val find1 = fn : int * bool -> bool
- reduce(find1, false, [3,3,3]);
val it = false : bool
- reduce(find1, false, [5,1,2]);
val it = true : bool

The Lambda Calculus

Fancy name for rules about how to represent and evaluate expressions with unnamed functions.
Theoretical underpinning of functional languages.
Side-effect free.
Very different from the Turing model of a store with evolving state.
ML: reduce(find1, false, [3,3,3]);
The Lambda Calculus: fn x => 2 * x;
English: “the function of \( x \) that returns the product of two and \( x \)”

Bound and Unbound Variables

In \( \lambda x.2x \), \( x \) is a bound variable. Think of it as a formal parameter to a function.
“\( 2x \)” is the body.
The body can be any valid lambda expression, including another unnamed function.
\[ \lambda x.\lambda y. (+ x y) 2 \]
“The function of \( x \) that returns the function of \( y \) that returns the product of the sum of \( x \) and \( y \) and 2.”

Arguments

\( \lambda x.\lambda y. (+ x y) 2 \)
is equivalent to the ML
fn x => fn y => (x + y) * 2;
All lambda calculus functions have a single argument.
As in ML, multi-argument functions can be built through such “currying.”
In this context, currying has nothing to do with Indian cooking. It is due to Haskell Brooks Curry (1900–1982),
who contributed to the theory of functional programming.
The Haskell functional language is named after him.

Calling Lambda Functions

To invoke a Lambda function, we place it in parentheses before its argument.
Thus, calling \( \lambda x.2x \) with 4 is written
(\( \lambda x.2x \)) 4
This means 8.
Curried functions need more parentheses:
(\( \lambda x.\lambda y. (+ x y) 2 \)) 4 5
This binds 4 to \( y \), 5 to \( x \), and means 18.

Grammar of Lambda Expressions

Utterly trivial:
\[
\begin{align*}
expr & \rightarrow \text{constant} \\
 & \mid \text{variable} \\
 & \mid expr \ expr \\
 & \mid (expr) \\
 & \mid \lambda \text{variable} . expr
\end{align*}
\]

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

Evaluating Lambda Expressions

Pure lambda calculus has no built-in functions; we'll be impure.
To evaluate \( (+ (* 5 6) (* 8 3)) \), we can't start with \(+ \) because it only operates on numbers.
There are two reducible expressions: \( (* 5 6) \) and \( (* 8 3) \).
We can reduce either one first. For example:
\[
\begin{align*}
(+ (* 5 6) (* 8 3)) \\
(+ 30 (* 8 3)) \\
(+ 30 24)
\end{align*}
\]
Looks like deriving a sentence from a grammar.

Evaluating Lambda Expressions

We need a reduction rule to handle \( \lambda s \):
(\( \lambda x.2x \)) 4
* 2 4
8
This is called \( \beta \)-reduction.
The formal parameter may be used several times:
(\( \lambda x.+ x x \)) 4
(+ 4 4)
8

Beta-reduction

May have to be repeated:
((\( \lambda x.\lambda y. (+ x y) 2 \)) 5) 4
(\( \lambda y.5y \)) 4
(\( \lambda f.f 3 \))(\( \lambda x.+ x 1 \))
(\( \lambda x.+ x 1 \)) 3
(+ 3 1)
More Beta-reduction

Repeated names can be tricky:
\[
\begin{align*}
&\lambda x. (+ (- x 1)) x 3) 9 \\
&\lambda x. (+ (- x 1)) x 3) 9 3 \\
&\lambda y. (+ (- y 1)) x 3) 9 3 \\
&\lambda y. (+ (- y 1)) x 3) 9 1 \\
&\lambda y. (+ (- y 1)) x 3) 9 3 \\
&\text{In the first line, the inner } x \text{ belongs to the inner } \lambda, \text{ the outer } x \text{ belongs to the outer one.}
\end{align*}
\]

Free and Bound Variables

In an expression, each appearance of a variable is either “free” (unconnected to a \( \lambda \)) or bound (an argument of a \( \lambda \)).

\( \beta \)-reduction of \( \lambda x. E \) \( y \) replaces every \( x \) that occurs free in \( E \) with \( y \).

Free or bound is a function of the position of each variable and its context.

Free variables
\[
(\lambda x. y (\lambda y. + y)) x
\]
Bound variables

Alpha conversion

One way to confuse yourself less is to do \( \alpha \)-conversion.

This is renaming a \( \lambda \) argument and its bound variables.

Formal parameters are only names: they are correct if they are consistent.

\[
\lambda x. (\lambda x. x) (+ 1 x) \leftrightarrow_\alpha \lambda x. (\lambda y. y) (+ 1 x)
\]

Alpha Conversion

An easier way to attack the earlier example:

\[
\begin{align*}
&\lambda x. (+ (- x 1)) x 3) 9 \\
&\lambda x. (+ (- x 1)) x 3) 9 3 \\
&\lambda y. (+ (- y 1)) x 3) 9 3 \\
&\lambda y. (+ (- y 1)) x 3) 9 1 \\
&\lambda y. (+ (- y 1)) x 3) 9 3 \\
&\text{In the first line, the inner } x \text{ belongs to the inner } \lambda, \text{ the outer } x \text{ belongs to the outer one.}
\end{align*}
\]

Reduction Order

The order in which you reduce things can matter.

The \textit{leftmost} redex is the one whose \( \lambda \) is to the left of all other redexes. You can guess which is the \textit{rightmost}.

The \textit{outermost} redex is not contained in any other.

The \textit{innermost} redex does not contain any other.

For \( (\lambda x. y) (\lambda z. z) (\lambda z. z) \),
\( (\lambda z. z) (\lambda z. z) \) is the leftmost innermost and
\( (\lambda x. y) (\lambda z. z) (\lambda z. z) \) 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 \( (\lambda x. y) (\lambda z. z) (\lambda z. z) \), 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 
\( (\lambda x.\lambda y.y) (\lambda z.z) (\lambda z.z) \)

Normal Form

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.”

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. | % Constant unifies with itself |
| yes       | |
| ?- a = b. | % Mismatched constants         |
| no        | |
| ?- 5.3 = a. | % Mismatched constants         |
| no        | |

| ?- foo(a,X) = foo(X,b). | % X=a required, but inconsistent |
| no                     | |

| ?- foo(a,X) = foo(X,a). | % X=a is consistent |
| no                     | |

| ?- foo(X,b) = foo(a,Y). | % X=a, then b=Y |
| no                     | |

| ?- foo(X,a,X) = foo(b,a,c). | % X=b required, but inconsistent |
| no                     | |

The Searching Algorithm

search(goal g, variables e)

for each clause \( h = t_1, \ldots, t_n \) in the database

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

if successful,

for each term \( t_1, \ldots, t_n \),

\( e = \text{search}(t_k, e) \)

if all successful, return \( e \)

return no

Order Affects Efficiency

edge(a, b). edge(b, c). edge(c, d). edge(d, e). edge(b, e). edge(d, f).

path(X, Y) :-
  edge(X, Z), path(Z, Y).

Consider the query

?- path(a, a).

Good programming practice: Put the easily-satisfied clauses first.

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