Review for the Final
COMS W4115
Prof. Stephen A. Edwards
Fall 2006
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,
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
Name, Scope, and Bindings
Control-flow constructs

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

Compiling a Simple Program
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;
}
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
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
A stream of tokens. Whitespace, comments removed.

Abstract syntax tree built from parsing rules.
Types checked; references to symbols resolved.
Transformation into 3-Address Code

L0: sne $1, a, b
seq $0, $1, 0
btrue $0, L1 % if (a != b)
sl $3, b, a
seq $2, $3, 0
btrue $2, L4 % if (a < b)
sub a, a, b
jmp L5
sub b, b, a
jmp L4
L4: sub b, b, a
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
jle .L5
jle .L3
subl %edx, %eax
jmp .L8
.L5: subl %eax, %edx
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: ε (the empty string), Stephen, αβγ

**Language:** A set of strings over an alphabet
- Examples: ⟨ε⟩ (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
- \( M^* = \{\epsilon, M, MM, MMM, \ldots\} = \\
  \{\epsilon, \text{man, men, manman, manmen, menman, menmen, menmenmen, manmanman, ...}\} \)

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 \( L(s) \),
   - \( (r) \mid (s) \) denotes \( L(r) \cup L(s) \)
   - \( (r)s \) denotes \( \{tu : t \in L(r), u \in L(s)\} \)
   - \( (r)^* \) denotes \( \bigcup_{i=0}^{\infty} L^i \) (\( L^0 = \emptyset \) and \( L^1 = LL^0 \))

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 \( \sigma : S \times \Sigma \to 2^S \)
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 \( x \) if there is a path from the start state to an accepting state that "spells out" \( x \).

Translating REs into NFAs
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:
- \(\cdot aabb, \text{Start}\)
- \(\cdot aabb, \epsilon\)-closure
- \(\cdot aabb, a\cdot bb\)
- \(\cdot aabb, a\cdot bb, \epsilon\)-closure
- \(\cdot aa\cdot bb\)
- \(\cdot aa\cdot bb, \epsilon\)-closure
- \(\cdot aab\cdot b\)
Simulating an NFA: $aab$, ε-closure

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 $s$.

Differs 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.
Grammars and Parsing

Ambiguous Grammars

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

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

$$\begin{array}{cccccccc}
3 & 4 & 2 & + & 5 \\
\hline
3 & 2 & 4 & 5 \\
4 & 3 & 5 & 2 \\
5 & 4 & 3 & 2 \\
\end{array}$$

Ambiguous: no precedence or associativity.

Assigning Precedence Levels

Split into multiple rules, one per level

Original ANTLR grammar specification

```antlr
expr : expr '+' expr | expr '-' expr | expr '*' expr | expr '/' expr | NUMBER ;
```

Still ambiguous: associativity not defined

Assigning Associativity

Make one side or the other the next level of precedence

```antlr
expr : expr '+' term | term ;
term : term '*' atom | term '/' atom | atom ;
atom : NUMBER ;
```

A Top-Down Parser

```c
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

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

becomes

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

AST expr() {
    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();
    */ Infinite Recursion */
}
Writing LL(1) Grammars

Cannot have common prefixes

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

becomes

AST: \text{expr}() { switch (next-token) { case \text{ID} : match(ID); match(')'); expr(); match(')'); case \text{ID} : match(ID); match('='); expr();

Eliminating Common Prefixes

Consolidate common prefixes:

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

becomes

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

Eliminating Left Recursion

Understand the recursion and add tail rules

\[ \text{expr} : \text{term} \ \text{exprt} \ |
\text{exprt} : \text{expr} \ |
\text{expr} \ (-\text{term}) \ |
\text{term} \ ; \]

Rightmost Derivation

A rightmost derivation for \text{Id} + \text{Id} + \text{Id}:

```
1. e \rightarrow t + e
2. e \rightarrow t
3. t \rightarrow \text{Id} + t
4. t \rightarrow \text{Id}
```

A rightmost derivation for \text{Id} + \text{Id} + \text{Id}:

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

Handles

This is a reverse rightmost derivation for \text{Id} + \text{Id} + \text{Id}.

Each highlighted section is a handle.

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

Shift-reduce Parsing

```
1. e \rightarrow t + e
2. e \rightarrow t
3. t \rightarrow \text{Id} + 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} + t
4. t \rightarrow \text{Id}
```

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

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 | 2 | 2
1 | 4 | 3 | 3 | 4
2 | 2 | 4 | 2 | 2
3 | 1 | 5
4 | 3 | 6 | 2
5 | 3 | 3 | 2 | 3
6 | 1 | 1 | 1 | 1
7 | 1 | 1 | 1 | 1
```

LR Parsing

```
1. e \rightarrow t + e
2. e \rightarrow t
3. t \rightarrow \text{Id} + t
4. t \rightarrow \text{Id}
```

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

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 | 2 | 2
1 | 4 | 3 | 3 | 4
2 | 2 | 4 | 2 | 2
3 | 1 | 5
4 | 3 | 6 | 2
5 | 3 | 3 | 2 | 3
6 | 1 | 1 | 1 | 1
7 | 1 | 1 | 1 | 1
```
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.

Names, Objects, and Bindings

Activation Records

Nested Subroutines in Pascal

Symbol Tables in Tiger
Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```c
if i 3 "This" /* valid */
#define123 /* 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 */
```

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

```c
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;
}
```

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.

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

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

Prints 1 3 2.

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.

```c
int p(int i) { printf("%d", i); return i; }
int q(int a, int b, int c) ()
    q( p(1), 2, p(3) );
```

Will not print 5 6 7. It will print one of 1 2 3 1 3 2, 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: 01
32-bit integer: 3 2 1 0
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
Slower to read an unaligned value: two reads plus shift.
3 2 1 0
7 6 5 4
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.

```c
struct padded {
    int x; /* 4 bytes */
    char z; /* 1 byte */
    short y; /* 2 bytes */
    char w; /* 1 byte */
};
```

Local arrays with fixed size are easy to stack.

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

Variable-sized local arrays aren’t as easy.

```c
void foo(int n) {
    return address <- FP
    a
    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

```c
void foo(int n) {
    return address <- FP
    a
    int a;
    int b[n];
    int c;
    b[n-1];
    c <- FP + ?
}
```

Variables remain constant offset from frame pointer.
Basic Blocks

A function that squares numbers:

```sml
val square = fn : int -> int
square 5;
val it = 25 : int
```

A more complex function

```sml
val max = fn : int -> int -> int
val max5 = max 5
val max5 = fn : int -> int
val max5 = max 4
val max5 = 6
```

Currying

Functions are first-class objects that can be manipulated with abandon and treated just like numbers.

```sml
val max = fn : int -> int -> int
val max5 = max 5
val max5 = fn : int -> int
val max5 = 4
val max5 = 6
```

Simple functional programming in ML

A function that squares numbers:

```sml
% 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
```

A more complex function

```sml
- fun max a b =
  if a > b then a else b;
val max = fn : int -> int -> int
- max 10 5;
val it = 10 : int
- max 5 10;
val it = 10 : int
- Notice the odd type:
- int -> int -> int
This is a function that takes an integer and returns a function that takes a function and returns an integer.
```

Fun with recursion

```sml
- fun add5 x = x + 5;
val add5 = fn : int -> int
- map(add5, [10,11,12]);
val it = [15,16,17] : int list
```

More recursive fun

```sml
- fun reduce (f, z, a::b::c) =
  if a = "-" then reduce(f, z, b::c)
  else reduce(f, reduce(f, z, b), c);
val reduce = fn : (char * 'a) list * 'a list
- 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
```

Control-Flow Graphs

A CFG illustrates the flow of control among basic blocks.

Separate Compilation

foo — An Executable

Currying

Functions are first-class objects that can be manipulated with abandon and treated just like numbers.

```sml
val max = fn : int -> int -> int
val max5 = max 5
val max5 = fn : int -> int
val max5 = 4
val max5 = 6
```

Reduce

Another popular functional language construct:

```sml
val reduce (f, z, nil) = z
| reduce (f, z, h::t) = f(h, reduce(f, z, t));
```

Currying

Functions are first-class objects that can be manipulated with abandon and treated just like numbers.

```sml
val max = fn : int -> int -> int
val max5 = max 5
val max5 = fn : int -> int
val max5 = 4
val max5 = 6
```

Reduce

Another popular functional language construct:

```sml
val reduce (f, z, nil) = z
| reduce (f, z, h::t) = f(h, reduce(f, z, t));
```

Currying

Functions are first-class objects that can be manipulated with abandon and treated just like numbers.

```sml
val max = fn : int -> int -> int
val max5 = max 5
val max5 = fn : int -> int
val max5 = 4
val max5 = 6
```

Reduce

Another popular functional language construct:

```sml
val reduce (f, z, nil) = z
| reduce (f, z, h::t) = f(h, reduce(f, z, t));
```

Currying

Functions are first-class objects that can be manipulated with abandon and treated just like numbers.

```sml
val max = fn : int -> int -> int
val max5 = max 5
val max5 = fn : int -> int
val max5 = 4
val max5 = 6
```
Another Example

Consider

```ml
- 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:

```
val x = 2 * x;
```

English:

“the function of \( x \) that returns the product of two and \( x \)”

Bound and Unbound Variables

In \( \lambda x. \ast \ x \), \( x \) is a bound variable. Think of it as a formal parameter to a function.

\( \ast \ x \) is the body.

The body can be any valid lambda expression, including another unnamed function.

```
\lambda \ x. \ 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, multiple-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 \( \lambda x. \ast \ x \ \ \ \ 2 \ x \) with 1 is written

```
(\lambda x. \ast \ 2) \ 4
```

This means 8.

Curried functions need more parentheses:

```
(\lambda x. (\lambda y. \ast \ (x + y) \ 2) \ 4) \ 5
```

This binds 4 to \( y \), 5 to \( x \), and means 18.

Grammar of Lambda Expressions

Utterly trivial:

```
expr → constant
| variable
| expr expr
| (expr)
| \lambda variable . expr
```

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:

```
(+ (* 5 6) (* 8 3)) 
```

```
(+ 30 (* 8 3)) 
```

```
(+ 30 24) 
```

Looks like deriving a sentence from a grammar.

Evaluating Lambda Expressions

We need a reduction rule to handle \( \lambda s \):

```
(\lambda x. \ast \ 2) \ 4 
```

```
(* 2 4) 
```

```
8
```

This is called \( \beta \)-reduction.

The formal parameter may be used several times:

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

```
(+ 4 4) 
```

```
8
```

Beta-reduction

May have to be repeated:

```
((\lambda x. (\lambda y. \ast \ x \ y) \ 5) \ 4) 
```

```
(\lambda y. \ast \ 5 \ y) \ 4 
```

```
(- 5 4) 
```

```
1
```

Functions may be arguments:

```
(\lambda f \ 3)(\lambda x. \ast \ 1) 
```

```
(\lambda x. \ast \ 1) \ 3 
```

```
(+ 3 1) 
```

```
4
```
More Beta-reduction

Repeated names can be tricky:

\[(\lambda x. (\lambda y. -y1) x) 3\]
\[(\lambda x. + (-x1) 93)\]
\[+(-91)3\]
\[+83\]
\[11\]

In the first line, the inner \(x\) belongs to the inner \(\lambda\), the outer \(x\) belongs to the outer one.

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.

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

Alpha Conversion

An easier way to attack the earlier example:

\[(\lambda x. (\lambda y. +(-y1)) x) 3\]
\[(\lambda x. (\lambda y. +(-y1)) x) 3\]
\[(\lambda y. +(-y1)) 93\]
\[+(-91)3\]
\[+83\]
\[11\]

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.

Reduction Order

The \(\text{redex}\) is a sub-expression that can be reduced.

The leftmost redex is the one whose \(\lambda\) 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 \((\lambda x. \lambda y. y) ( (\lambda z. z) (\lambda z. z) )\),

\((\lambda z. z) (\lambda z. z)\) is the leftmost innermost and

\((\lambda x. \lambda y. y) ( (\lambda z. z) (\lambda z. z) )\) is the leftmost outermost.

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."