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.

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;
}
```

Symbols resolved; references to symbols checked.
Translation into 3-Address Code

<table>
<thead>
<tr>
<th>L0: sne $1, a, b</th>
<th>seq $0, $1, 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>btrue $0, L1</td>
<td>% while (a != b)</td>
</tr>
<tr>
<td>sl $3, b, a</td>
<td>seq $2, $3, 0</td>
</tr>
<tr>
<td>btrue $2, L4</td>
<td>% if (a &lt; b)</td>
</tr>
<tr>
<td>sub a, a, b</td>
<td>% a -= b</td>
</tr>
<tr>
<td>jmp L5</td>
<td></td>
</tr>
<tr>
<td>L4: sub b, b, a</td>
<td>% b -= a</td>
</tr>
<tr>
<td>L5: jmp L0</td>
<td></td>
</tr>
<tr>
<td>L1: ret a</td>
<td></td>
</tr>
</tbody>
</table>

Idealized assembly language w/ infinite registers

Generation of 80386 Assembly

```
gcd: pushl %ebp       % Save frame pointer
    movl %esp, %ebp
    movl b(%ebp), %eax % Load a from stack
    movl %ebp, %edx    % Load b from stack
.L8: cmpl %edx, %eax % 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: ε (the empty string), Stephen, αβγ

**Language:** A set of strings over an alphabet
Examples: ǫ (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
\( M^* = \{ \epsilon, M, MM, MMM, \ldots \} = \{ \epsilon, \text{man, men, manman, manmen, mennmen, messmen, manmanman, manmanmen, manmanmen, manmenmen, \ldots} \} \)

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 \( L(r) \cup L(s) \)
   - \( (r)(s) \) denotes \( \{ tu : t \in L(r), u \in L(s) \} \)
   - \( (r)^* \) denotes \( \bigcup_{i=0}^{\infty} L_i \) (where \( L_0 = \emptyset \) 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 \( \sigma : S \times \Sigma \rightarrow 2^S \)
4. Start state \( s_0 \): A
5. Set of accepting states \( F \): \{A\}

Translating REs into NFAs

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" \( x \).

Show that the string "010010" is accepted.
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: \(aabb\), Start

Simulating an NFA: \(aabb\), \(\epsilon\)-closure

Simulating an NFA: \(aa\cdot bb\)

Simulating an NFA: \(aa\cdot bb\), \(\epsilon\)-closure

Simulating an NFA: \(aab\cdot b\)
Simulating an NFA: $aab\cdot b$, $\epsilon$-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 $s$.

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

\[
3 - 4 * 2 + 5
\]

with the grammar

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

Ambiguous: no precedence or associativity.

Assigning Associativity

Make one side or the other the next level of precedence

\[
\begin{align*}
expr & : expr + term \\
& | expr - term \\
& | term ; \\
term & : term * atom \\
& | term / atom \\
& | atom ; \\
atom & : NUMBER ;
\end{align*}
\]

A Top-Down Parser

\[
\begin{align*}
stmt & : if expr then expr \\
& | while expr do expr \\
& | expr := expr ;
expr & : NUMBER | '(' expr ')' ;
AST stmt() {
switch (next-token) {
\text{case "if": match("if"); expr(); match("then"); expr();} 
\text{case "while": match("while"); expr(); match("do"); expr();} 
\text{case NUMBER or "": expr(); match("="); expr();}
}
\]

Writing LL(k) Grammars

Cannot have left-recursion

\[
\begin{align*}
expr & : expr + term | term ;
\end{align*}
\]

becomes

\[
\begin{align*}
AST expr() {
switch (next-token) {
\text{case NUMBER: expr(); /* Infinite Recursion */}
}
\]

Assigning Precedence Levels

Split into multiple rules, one per level

\[
\begin{align*}
expr & : expr + expr \\
& | expr - expr \\
& | expr * expr \\
& | expr / expr \\
& | NUMBER ;
\end{align*}
\]

Ambiguous: no precedence or associativity.

Grammars and Parsing
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(\text{ID}); match(''); expr(); match('');
    case \text{ID} : match(\text{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

\[
\text{expr} : \text{term} \ \text{exprt} \\
\text{exprt} : '+' \text{term} \ \text{exprt} \\
| '-' \text{term} \ \text{exprt} \\
| /* nothing */
\]

Bottom-up Parsing

Rightmost Derivation

1: \text{e} \rightarrow \text{t} + \text{e}
2: \text{e} \rightarrow \text{t}
3: \text{t} \rightarrow \text{id} + \text{t}
4: \text{t} \rightarrow \text{id}

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

\[
\begin{array}{c|c|c}
\text{state} & \text{input} & \text{action} \\
0 & \text{id} & \text{shift}
1 & \text{id} & \text{shift}
2 & \text{id} & \text{shift}
3 & \text{id} & \text{reduce (4)}
4 & \text{id} & \text{reduce (3)}
5 & \text{id} & \text{shift}
6 & \text{id} & \text{reduce (4)}
7 & \text{id} & \text{reduce (1)}
8 & \text{id} & \text{accept}
\end{array}
\]

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

Shift-reduce Parsing

\[
\begin{array}{c|c|c|c}
\text{stack} & \text{input} & \text{action} \\
0 & \text{id} + \text{id} + \text{id} & \text{shift, goto 1}
1 & \text{id} + \text{id} + \text{id} & \text{shift, goto 3}
2 & \text{id} + \text{id} + \text{id} & \text{reduce w/ 4}
\end{array}
\]

Action is reduce with rule 4 (\text{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: 

LR Parsing

\[
\begin{array}{c|c|c|c}
\text{stack} & \text{input} & \text{action} \\
0 & \text{id} + \text{id} + \text{id} & \text{shift, goto 1}
1 & \text{id} + \text{id} + \text{id} & \text{shift, goto 3}
2 & \text{id} + \text{id} + \text{id} & \text{reduce w/ 4}
\end{array}
\]

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

An oracle tells what to do.
LR Parsing

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e</td>
<td>goto</td>
</tr>
<tr>
<td>1</td>
<td>t</td>
<td>push t</td>
</tr>
<tr>
<td>2</td>
<td>e</td>
<td>goto</td>
</tr>
<tr>
<td>3</td>
<td>t</td>
<td>push t</td>
</tr>
<tr>
<td>4</td>
<td>t</td>
<td>goto</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: \( t \rightarrow Id \ast t \)
4: \( t \rightarrow Id \)

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

- 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

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>e′</td>
<td>e</td>
</tr>
<tr>
<td>S1</td>
<td>t</td>
<td>Id</td>
</tr>
<tr>
<td>S2</td>
<td>t</td>
<td>Id + e</td>
</tr>
<tr>
<td>S3</td>
<td>t</td>
<td>Id + t</td>
</tr>
<tr>
<td>S4</td>
<td>t</td>
<td>Id + t</td>
</tr>
<tr>
<td>S5</td>
<td>t</td>
<td>Id + t</td>
</tr>
<tr>
<td>S6</td>
<td>t</td>
<td>Id + t</td>
</tr>
<tr>
<td>S7</td>
<td>t</td>
<td>Id + t</td>
</tr>
</tbody>
</table>

Symbols, Objects, and Bindings

- Object1
- Object2
- Object3
- Object4

- Binding

- Name1
- Name2
- Name3
- Name4

Activation Records

- Parent
- Argument
- Return Address
- Frame Pointer
- Old Frame Pointer
- Local Variables
- Temporaries/Arguments
- Stack Pointer
- Growth of Stack

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

```tiger
let
var n := 8
var x := 3
function sqr(a:int)
    = a * a
in
n := array of int
```

Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```cpp
if i 3 "This" /* valid */
#41123 /* invalid */
```

Syntactic analysis: Makes sure tokens appear in correct order

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

Semantic analysis: Makes sure program is consistent

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

Basic paradigm: recursively check AST nodes.

```cpp
1 + break 1 - 5
```

```
check(+)
check(-)
check(1) = int
cHECK(5) = int
```

FAIL: int ≠ void
Types match, return int

```
Check(+)
```

```
Check(1)
```

```
Check(5)
```

```
1 - 5
```

```
Check(1)
```

```
Check(5)
```

```
Check(1)
```

Ask yourself: at a particular node type, what must be true?

Implementing multi-way branches

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

Obvious way:

```cpp
if (s == 1) { one(); }
else if (s == 2) { two(); }
else if (s == 3) { three(); }
else if (s == 4) { four(); }
```

Reasonable, but we can sometimes do better.

Implementing multi-way branches

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

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

```
labels l[] = { L1, L2, L3, L4 }; /* Array of labels */
if (s >= 1 && s <= 4) goto l[s-1]; /* not legal C */
L1: one(); goto Break;
L2: two(); goto Break;
L3: three(); goto Break;
L4: four(); goto Break;
Break:
```

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;
    return;
}
```

```
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 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) );
```

 Prints 1 2 3.

Some functional languages also use normal order evaluation to avoid doing work. "Lazy 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;
    return;
}
```

```
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 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) );
```

Prints 1 2 3.

Some functional languages also use normal order evaluation to avoid doing work. "Lazy 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;
    return;
}
```

```
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 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) );
```

Prints 1 2 3.

Some functional languages also use normal order evaluation to avoid doing work. "Lazy 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;
    return;
}
```

```
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 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) );
```

Prints 1 2 3.

Some functional languages also use normal order evaluation to avoid doing work. "Lazy Evaluation"
Modern processors have byte-addressable memory.

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

16-bit integer: 10
32-bit integer: 3210

Layout of Records and Unions

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.

Layout of Records and Unions

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 */
};
```

Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

```c
void foo() {
    int a;
    int b[10];
    int c;
}
```

Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

```c
void foo(int n) {
    int a;
    int b[n];
    int c;
}
```

Stack-Based IR: Java Bytecode

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

Register-Based IR: Mach SUIF

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

The statements in a basic block all run if the first one does. Starts with a statement following a conditional branch or is a branch target. Usually ends with a control-transfer statement.

Control-Flow Graphs

A CFG illustrates the flow of control among basic blocks.

Separate Compilation

Separate Compilation

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 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.

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 max5 = max5 4;
val it = 5 : int
val it = 6 : int

Fun with recursion

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

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));
val reduce = fn : ('a -> 'a) * 'a -> 'a list -> 'a
val reduce = reduce( fn (x, y) => x - y, 0, [1, 5])
val it = -4 : int
val 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:
The Lambda Calculus:
\[ \lambda x. \ast \ 2 \ x \]
English:
“the function of \( x \) that returns the product of two and \( x \)”

Bound and Unbound Variables

In \( \lambda x. \ast \ 2 \ x \), \( x \) is a **bound variable**. Think of it as a formal parameter to a function.
“\( \ast \ 2 \ x \)” 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
\[ fn \ x \Rightarrow \ fn \ y \Rightarrow \ (x + y) \ * \ 2; \]
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.

Evaluating Lambda Expressions

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

Calling Lambda Functions

To invoke a Lambda function, we place it in parentheses before its argument.
Thus, calling \( \lambda x. \ast \ 2 \ x \) with 1 is written
\[ (\lambda x. \ast \ 2) \ 4 \]
This means 8.
Curried functions need more parentheses:
\[ (\lambda x. (\lambda y. \ (+ \ x \ y) \ 2)) \ 4 \]
This binds 4 to \( y \), 5 to \( x \), and means 18.

Grammar of Lambda Expressions

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

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

Beta-reduction

May have to be repeated:
\[
\begin{align*}
((\lambda x. (\lambda y. \ (+ \ x \ y)) \ 5) \ 4 \\
(\ast \ 2) \ 4 \\
8 \\
\end{align*}
\]

Functions may be arguments:
\[
\begin{align*}
(\lambda f \ 3)(\lambda x. \ (+ \ x \ 1)) \\
(\ast \ 2) \ 4 \\
(+ \ 4 \ 4) \\
8 \\
\end{align*}
\]
More Beta-reduction

Repeated names can be tricky:

\[(\lambda x. (\lambda x. + ( - x 1 )) x) 3\]
\[(\lambda x. + ( - x 1 )) 9 3\]
\[+ ( - 9 1 ) 3\]
\[+ 8 3\]
\[11\]

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.

\(\lambda x. (\lambda x. x) (+ 1 x) \leftrightarrow \alpha \lambda x. (\lambda y. y) (+ 1 x)\)

Alpha Conversion

An easier way to attack the earlier example:

\[(\lambda x. (\lambda x. + ( - x 1 )) x) 3\]
\[(\lambda x. (\lambda y. + ( - y 1 )) x) 3\]
\[(\lambda y. + ( - y 1 )) 9 3\]
\[+ ( - 9 1 ) 3\]
\[+ 8 3\]
\[11\]

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

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

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:

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

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 

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

**Normal Form**

Not everything has a normal form 

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

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
- `?- a = b.` % Mismatched constants
- `?- b = a.` % Mismatched constants
- `?- 5.3 = 5.3.` % Variables unify
- `?- foo(a,X) = foo(X,b) .` % X=a required, but inconsistent
- `?- foo(a,X) = foo(X,a) ,` % X=a is consistent
- `?- X = a .` % X=a, then b=Y
- `?- foo(a,X) = foo(b,a,c) .` % X=b required, but inconsistent

**The Searching Algorithm**

search(goal g, variables e)

for each clause h := t1, . . . , tn in the database 

c = unify(g, h, e)

if successful,

for each term t1, . . . , tm,

c = search(tk, e)

if all successful, return e

return no

**Order Affects Efficiency**

Consider the query

`?- path(a, a).`

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

**Order Affect 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).`

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