Course Overview

Programming language concepts

- Variables and expressions
- Assignments and commands
- Procedures / methods
- Objects and classes
- Types

We have used lambda calculus as a uniform notation to study all of these concepts.

But it is also important to relate these ideas to Java-like languages.
Make sure you understand:

- Syntax and parsing
- Terms as data
  (defining functions / relations on terms)
- Operational semantics
  - reduction
  - abstract machine
- Type checking
- Type inference
Variables

(see Handout 8)

Two distinct concepts

1. names used to denote quantities (parameters, procedure names, type names, ...)

2. cells that have memory/state changeable by assignments.

Example:

\[ a[i] = \text{pop} (s); \]

\[ a: \begin{array}{c}
\vdots \\
\vdots
\end{array} \quad \text{name of a cell whose value is an array of cells.} \]

\[ i: \begin{array}{c}
\vdots \\
\vdots
\end{array} \quad \text{name of a cell} \]

\[ a[i] \quad \text{expression denoting a cell.} \]

\[ s: \begin{array}{c}
\vdots \\
\vdots
\end{array} \quad \text{name of a cell whose value is a stack} \]

pop: \text{name of a procedure}
Diverse terminology (confusing!)

<table>
<thead>
<tr>
<th></th>
<th>names</th>
<th>cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$ calculus</td>
<td>variable</td>
<td>—</td>
</tr>
<tr>
<td>Algol 60</td>
<td>identifier</td>
<td>variable</td>
</tr>
<tr>
<td>ML</td>
<td>variable</td>
<td>reference</td>
</tr>
<tr>
<td>Scheme</td>
<td>variable</td>
<td>variable?</td>
</tr>
<tr>
<td>Java</td>
<td>?</td>
<td>variable</td>
</tr>
</tbody>
</table>

In Java, there is no uniform terminology for variables in the sense of lambda calculus:
- parameters
- method names
- class names
- type names, type variables
**Constants**

Symbols whose meaning is fixed.

42, 3.6, true - value constants

+, -, = - function constants

int, bool - type constants.

**Expressions**

Typically obtained by applying functions to arguments

- $3 \times x$ - apply $\times$ to 3, x
- pop(s) - apply pop to s
- a[i] - apply subscripting operation to a and i

```
  app
 /   \
*    app
   /   app
 3    pop
     /  app
    \    pop
     i   subs
      /  a
     \  i
      ```
Defining functions / procedures

```java
public static int f (int x) {
    return 3 * x;
}
```

---

Our notation (untyped)

```latex
\text{let } f = \lambda(x). \ast (3, x) \\
\text{in} \quad \ldots
```

Keyword for defining functions

Naming the function is a separate step from building the function.

Our typed notation

```latex
\text{let } f : \text{int} \rightarrow \text{int} \\
\quad = \lambda(x:\text{int}). \ast (3, x) \\
\text{in} \quad \ldots
```
Applied lambda calculus

- All functions are unary (single arguments)
- Functions are "first class" values.
- can use "curried" functions to simulate multiple arguments.

\[ \lambda x. (\times 3) x \]

\[ \times : \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \]

\[ \times 3 : \text{int} \rightarrow \text{int} \]

\[ (\times 3) x : \text{int} \]

Abstract syntax tree

```
\lambda x.
  \_\_\_
  /\  /
app  \
\  /
app  \\
*  3
  \_
  \_
  x
```
Bracketing conventions

* Memorise them!!

1. Application associates to the left.
2. The scope of \( \lambda \) extends as far to the right as possible.

\[
\begin{align*}
* \ 3 \ x \ & \text{ means } \ (\,* \ 3) \ x \\
\not= \ & \ (\,* \ (3 \ x)) \\
\lambda x. \ * \ 3 \ x \ & \text{ means } \ (\lambda x. \ (\,* \ 3) \ x) \\
\not= \ & \ (\lambda x. \ (\,* \ 3) \ x) \\
\not= \ & \ (\lambda x. \ (\:* \ 3) \ x)
\end{align*}
\]

Don't get confused!

If you need to understand clearly, put more brackets for yourself or draw the syntax tree.
Example:

\[ \lambda x. (\lambda y. y) \ x \) \ z \]
Syntax of applied lambda calculus

\[ M \rightarrow \text{to stand for terms} \]
\[ \alpha \rightarrow \text{to stand for variables} \]
\[ c \rightarrow \text{to stand for constants} \]

\[ M ::= c \mid \alpha \mid \lambda \alpha. M \mid M_1, M_2 \]

More elaborate notation:

\[ \langle \text{term} \rangle ::= \langle \text{constant} \rangle \]
\[ \mid \langle \text{variable} \rangle \]
\[ \mid \lambda \langle \text{variable} \rangle. \langle \text{term} \rangle \]
\[ \mid \langle \text{term} \rangle \langle \text{term} \rangle \]

Use brackets as necessary to group subterms.
Free and bound variables

\[ \lambda \alpha \cdot \ast 3 \uparrow \alpha \]

binding for \( \alpha \)

bound occurrence of \( \alpha \).

\[ (\lambda \alpha \cdot \ast 3 \alpha) \uparrow \]

free occurrence of \( \alpha \)

An occurrence of \( \alpha \) is bound if it is in the scope of a \( '\lambda \alpha ' \).

The occurrence is free if it is not in the scope of a \( '\lambda \alpha ' \).

Example:

\[ (\lambda \alpha \cdot \alpha) (\lambda y \cdot z) 2) \]
Substitution

$M \left[ N / \alpha \right] - \text{The result of substituting } N \text{ for } \alpha \text{ in the term } M$

$$(\lambda \alpha. \alpha) \ ((\lambda y. \alpha) \ 2) \ \left[ k / \alpha \right]$$

$$= (\lambda \alpha. \alpha) \ ((\lambda y. k) \ 2)$$

What about:

$$(\lambda \alpha. \alpha) \ ((\lambda y. \alpha) \ 2) \ \left[ * k y / \alpha \right] ?$$

$$(\lambda \alpha. \alpha) \ ((\lambda y. * k y) \ 2) \ \times$$

$$(\lambda \alpha. \alpha) \ ((\lambda y'. * k y) \ 2) \ \uparrow$$

rename the bound variable to avoid "variable capture".

Variable capture means importing a term with free variable into the scope of a binding for that variable.
Mathematical functions on terms

Terms are data.

So, we can define functions on terms.

("Mathematical" functions or "Meta-level" functions).

Function to count the number of applications.

\[ \# : \text{term} \to \text{integer} \]

\[ \# \ c = 0 \]
\[ \# \ x = 0 \]
\[ \# \ (\lambda x. M) = \# M \]
\[ \# \ (M \ N) = \# M + \# N + 1 \]

\[ \uparrow \]

Defined by "induction" on the structure of terms

\[ \uparrow \uparrow \]

Can use recursive calls for subterms
Definition of substitution

\( [N/x] : \text{term} \rightarrow \text{term} \) (partially defined)

\[
y[N/x] = \begin{cases} 
N, & \text{if } y \text{ is the same as } x \\
y, & \text{otherwise}
\end{cases}
\]

\[
c[N/x] = c
\]

\[
(\lambda y. M)[N/x] = \begin{cases} 
\lambda y. M, & \text{if } y \text{ is the same as } x \\
\lambda y^*. M[N/x], & \text{if } y \text{ is different from } x \\
\end{cases}
\]

\[
(M, M_2)[N/x] = M_1[N/x] M_2[N/x]
\]

- If the bound variables of \( M \) are free in \( N \), then rename them first.
Definition of free variables

\( FV : \text{term} \rightarrow \text{set of variables} \)

\[
FV(c) = \{\}\n\]

\[
FV(x) = \{x\}\n\]

\[
FV(\lambda x. M) = FV(M) \setminus \{x\}\n\]

\[
FV(M, M_2) = FV(M_1) \cup FV(M_2)\n\]

Example:

\[
FV((\lambda x. x)((\lambda y. z)2))\n\]

\[
= FV(\lambda x. x) \cup FV((\lambda y. z)2)\n\]

\[
= FV(x) \setminus \{x\} \cup FV(z) \setminus \{y\} \cup FV(2)\n\]

\[
= \{x\} \setminus \{x\} \cup \{z\} \setminus \{y\} \cup \{z\}\n\]

\[
= \{\}\cup \{z\} \cup \{z\}\n\]

\[
= \{z\}\n\]
Meaning of $\lambda$ calculus

$\kappa$-conversion law (variable renaming)

$$\lambda \alpha. M \equiv \lambda y. M[y/\alpha]$$

$\beta$-conversion law (function expansion)

$$(\lambda \alpha. M) N \equiv M[N/\alpha]$$

$\eta$-conversion law (elimination of abstraction)

$$\lambda \alpha. M \alpha \equiv M$$

$\delta$-conversions (meaning of constants).

$$+ 2 3 \equiv 5$$

if true $M \, N \equiv M$

if false $M \, N \equiv N$

Reduction semantics

Use $\beta$-conversion and $\delta$-conversion

left to right.
Scheme-like Applied λ calculus

integers, booleans, lists

list constants

nil       - empty list
cons      - constructs a pair

\[
\begin{array}{c}
\text{\texttt{cons} (\texttt{x}, \texttt{y})} \\
\end{array}
\]

\( \rightarrow \)

\( \texttt{x} \)

(used for adding an element at the front of a list).

null      - check if a list is nil

\texttt{null} (\texttt{cons} \texttt{x} \texttt{y}) = \texttt{false}
\texttt{null} (\texttt{nil}) = \texttt{true}

car       - access the first component of a cons pair (head)

\texttt{car} (\texttt{cons} \texttt{x} \texttt{y}) = \texttt{x}

cdr       - access the second component of a cons pair (tail)

\texttt{cdr} (\texttt{cons} \texttt{x} \texttt{y}) = \texttt{y}
Recursion

\[ Y = \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x)) \]

The \( Y \) combinator has the property that

\[ (Y M) \equiv M (Y M) \]

Syntactic sugar

\[
\begin{align*}
\text{let } x &= M \\
\text{in } N &\quad \Rightarrow (\lambda x. N) M \\
\text{letrec } f &= M \\
\text{in } N &\quad \Rightarrow (\lambda f. N) (Y \lambda f. M) \\
Y \lambda f. M &\quad \Rightarrow (\lambda f. M) (Y \lambda f. M) \\
&\quad \Rightarrow M [ (Y \lambda f. M) / f ].
\end{align*}
\]

Any occurrence of \( f \) inside \( M \) can expand out into the whole definition again.
More syntactic sugar

\[ \text{let } f \ x_1 \ldots x_n = M \quad \text{in} \quad N \quad \mapsto \quad \text{let } f = \lambda x_1 \ldots \lambda x_n. M \quad \text{in} \quad N \]

\[ \text{let } f_1 = M_1 \quad \vdots \quad f_k = M_k \quad \text{in} \quad N \quad \mapsto \quad \text{let } f_1 = M_1 \quad \vdots \quad f_k = M_k \quad \text{in} \quad N \]
Examples

1. An infinite list of 0's.
   \[
   \text{letrec } \ l = \text{cons } 0 \ l
   \]
   \[
   \text{in } \ldots
   \]

   Note:
   \[
   \text{can } \ l \equiv \text{can} \ (\text{cons } 0 \ [\underline{l}])
   \equiv 0
   \]
   \[
   \text{can} \ (\text{cdr } \ l) \equiv \text{can} \ (\text{cdr} \ (\text{cons } 0 \ [\underline{l}]))
   \equiv \text{can} \ ([\underline{l}])
   \equiv \text{can} \ (\text{cons } 0 \ [\underline{l}])
   \equiv 0
   \]

   We are using the notation \([\underline{l}]\) for
   \[
   Y \ \lambda l. \ \text{cons } 0 \ l
   \]
   which reduces to
   \[
   \text{cons } 0 \ [\underline{l}]
   \]
2. An infinite list of integers

\[
\text{letrec } \text{intsfrom } n = \text{cons } n \ (\text{intsfrom } n+1) \\
\text{in } \ldots
\]

So, \( \text{intsfrom } 0 \)

\[
\equiv \text{cons } 0 \ (\text{intsfrom } 1) \\
\equiv \text{cons } 0 \ (\text{cons } 1 \ (\text{intsfrom } 2))
\]

3. The infinite list of Fibonacci numbers

\[
l = 0 \ 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ldots
\]

obtained by adding the previous two elements

\[
l = \ 0 \ 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ldots
\]

\[
\text{cdr } l = 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ldots
\]

\[
\underline{l} \ + \ \text{cdr } l = 1 \ 2 \ 3 \ 5 \ 8 \ldots
\]

\[
l = l + (\text{cdr } l)
\]
(4) Function to append lists

\[
\text{letrec \ append \ l \ m =}
\begin{align*}
\text{if (null } l) \\
\text{m} \\
\text{(cons (car } l) \\
\text{(append (cdr } l) \ m))}
\end{align*}
\]

(5) Function to apply a function \( f \) to all the elements of a list:

\[
\text{map : (int } \rightarrow \text{ int) } \rightarrow \text{ (list } \rightarrow \text{ list)}
\]

\[
\text{letrec \ map \ f \ l =}
\begin{align*}
\text{if (null } l) \\
\text{nil} \\
\text{(cons (f (car } l))} \\
\text{(map f (cdr } l))
\end{align*}
\]

Look through exercises for various example problems.