Week 1: Semantics and Pure Functional Programs

CSC324 Principles of Programming Languages

David Liu, Department of Computer Science

The Lambda Calculus, revisited
The Lambda Calculus (original)

```
expr = ID  (* Identifier *)
    | "(" "\" ID "." expr ")" (* Function value *)
    | "(" expr expr ")" (* Function call *)
```

Example: (λ x . x)

The Lambda Calculus (Racket)

```
expr = ID  (* Identifier *)
    | "(" "lambda" "(" ID ")" expr ")" (* Function *)
    | "(" expr expr ")" (* Call *)
```

;
The Lambda Calculus (Haskell)

```
expr = ID          (* Identifier *)
     | "\" ID "->" expr (* Function *)
     | expr expr   (* Call *)
```

The Lambda Calculus (Python)

```
expr = ID          (* Identifier *)
     | "lambda" ID ":" expr (* Function *)
     | expr "(" expr ")" (* Call *)
```
What is a “program” in the lambda calculus?
A single expression generated from its grammar (consisting of identifiers, function values, and function calls).

```
prog = expr ;
expr = (* omitted *) ;
```

What does it mean to “run” a program?
The semantics of the lambda calculus is based on evaluating function calls by substitution.

```
((\ x . x) y)  ==>  y
```
Extending the lambda calculus

In Racket/Haskell, we'll add another principle to program semantics: built-in data types and functions that operate on them.

```
((lambda (x) (+ x 10)) 20)
; ==> (+ 20 10)
; ==> 30
```
WTF

Wow That's Factorial

(name binding)

an association of an identifier to a value

(* Racket *)

\[
\text{binding} = \left( \text{"define" } \text{ID } \text{expr } \text{"\}} \right)
\]

(* Haskell *)

\[
\text{binding} = \text{ID "=} \text{expr}
\]
What is a “program” for us?

Zero or more name bindings followed by an expression.

```
prog = binding ... expr ;

binding = (* omitted *) ;
expr = (* omitted *) ;
```

Summary

In our pure functional programs, the semantics use just three principles:

1. Function call evaluation by substitution
2. Built-in data types and functions operating on them
3. Name bindings and name lookup
Why is functional programming called “pure”?

**Function call evaluation**

A function call is evaluated by substituting arguments into the function body, evaluating the resulting expression, and then returning the value.

That’s it—we have no way to express side effects:

1. Mutating an argument (or any other value)
2. Perform any external I/O (e.g., `print`, `open`).
Immutability

All values are immutable.

Names cannot be rebound.

Results in **referentially transparent** names: a name can be replaced by its value everywhere in the program without changing the program’s meaning.

<table>
<thead>
<tr>
<th>Racket</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(define nums (list 1 2 3))</code></td>
<td><code>nums = [1, 2, 3]</code></td>
</tr>
<tr>
<td>; ... code omitted ...</td>
<td># ... code omitted ...</td>
</tr>
<tr>
<td><code>(f nums)</code></td>
<td><code>f(nums)</code></td>
</tr>
<tr>
<td>; ... code omitted ...</td>
<td># ... code omitted ...</td>
</tr>
<tr>
<td><code>(g nums)</code></td>
<td><code>g(nums)</code></td>
</tr>
<tr>
<td><strong>Racket</strong></td>
<td><strong>Python</strong></td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td><code>(define nums (list 1 2 3))</code></td>
<td><code>nums = [1, 2, 3]</code></td>
</tr>
<tr>
<td>; ... code omitted ...</td>
<td># ... code omitted ...</td>
</tr>
<tr>
<td><code>(f (list 1 2 3))</code></td>
<td><code>f([1, 2, 3])</code></td>
</tr>
<tr>
<td>; ... code omitted ...</td>
<td># ... code omitted ...</td>
</tr>
<tr>
<td><code>(g (list 1 2 3))</code></td>
<td><code>g([1, 2, 3])</code></td>
</tr>
</tbody>
</table>

**Structural decomposition**
**structural recursion**

An algorithm design in which structured input data is decomposed into subcomponents with the same structure, which are then processed recursively.

---

**Recursive data definition**

A list is either:

- empty
- a value $x$ “in front of” another list $xs$

(we say “$x$ cons $xs$”)
Recursive function example

The sum of a list is either:

• _____, if the list is empty
• _____, if the list is “x cons xs”

Recursive function example

The sum of a list is either:

• 0, if the list is empty
• x plus the sum of xs, if the list is “x cons xs”
Recursive function template

**Racket**

```racket
(define (f lst)
  (if (empty? lst))
      ...
      ...
  (... (first lst) ... (f (rest lst)) ...)))
```

**Haskell**

```haskell
f lst = if null lst
      then
        ...
      else
        ... (head lst) ... (f (tail lst)) ...
```

Pattern-matching (value-based)

**Racket**

```racket
(define/match (f x)
  [(0) 10]
  [(1) 20]
  [(x) (+ x 30)])
```

**Haskell**

```haskell
f 0 = 10
f 1 = 20
f x = x + 30
```
Pattern-matching (structural)

\[
\text{sum} \ lst = \begin{cases} 
\ 0 & \text{if null} \ lst \\
\ x + \text{sum} \ xs & \text{else} 
\end{cases}
\]

\[
\text{sum} \ [] = 0
\]

\[
\text{sum} \ (x:xs) = x + \text{sum} \ xs
\]

\[
(\text{define/match} \ (\text{sum} \ lst) \\
\quad [((\text{list})) 0] \\
\quad [((\text{cons} \ x \ xs)) (+ x (\text{sum} \ xs))])
\]

---

Recursion: efficient implementations
Denotational semantics

The meaning of an expression as an abstract mathematical value.
Intuitively, “what this expression evaluates to.”

Operational semantics

The meaning of an expression as what computational steps it represents.
Intuitively, “how this expression should be evaluated.”
(define (count n)
  (if (zero? n)
      0
      (+ 1 (count (- n 1)))))

(count 0)
(count 1)
(count 99)
(count 100)
(define (count n)
  (if (zero? n)
      0
      (+ 1 (count (- n 1)))))

(define (countz n)
  (if (zero? n)
      0
      (countz (- n 1))))
Let $E$ be an expression, and $E'$ be a subexpression in $E$. $E'$ is in a **tail position** with respect to $E$ if evaluating $E'$ is the last step in evaluating $E$.

A function call in tail position is called a **tail call**.

A recursive function is **tail recursive** when all of its recursive calls are tail calls.
tail call elimination

An optimization that removes (i.e., deallocates) the current stack frame when a tail call is made.

Transforming recursion into tail recursion

Use accumulators to track “leftover” computations.
Transforming tail recursion into a loop

Parameters become loop-updated variables.