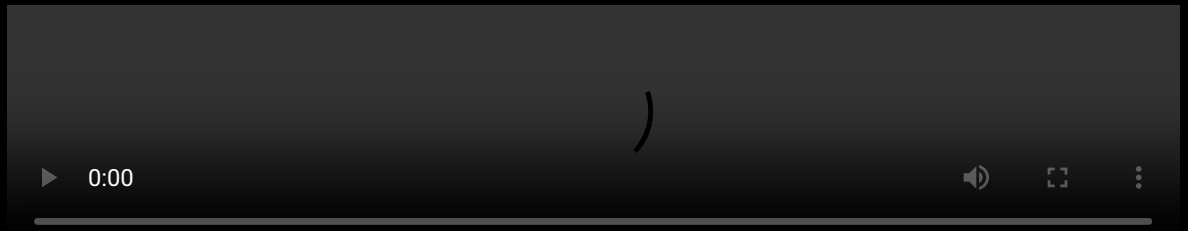
CS450

Structure of Higher Level Languages

Lecture 13: Specifying λ_S

Tiago Cogumbreiro

Why should we care about mathematical formalisms?



Source: www.youtube.com/watch?v=pgWTmOyUjtM

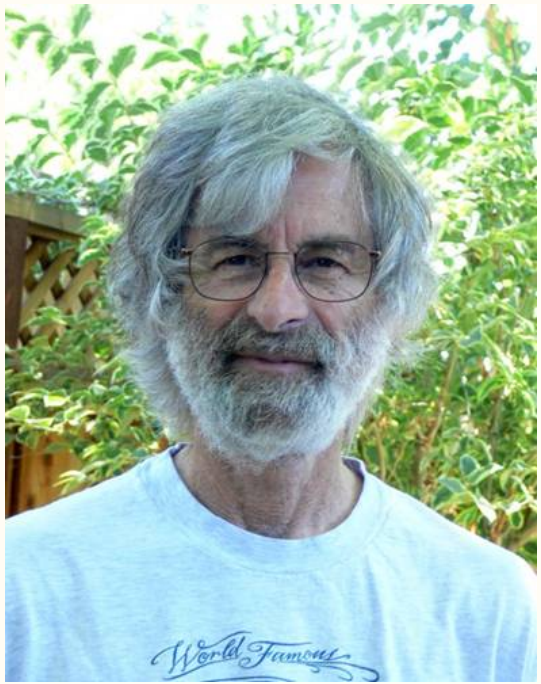
Leslie Lamport

2013 Turing Award:

> For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.

Did you know?

- Lamport is the creator of TLA+ "a high-level language for modeling programs and systems" and verifying the correctness of these models
- Lamport also created LaTeX!



Source: `lamport.org`

UMass Boston

The importance of formalisms

- A mathematically precise specification (no ambiguity)
- *Notation:* condense information, an abstraction tool, a visualization aid

Limitations

- Maintaining a specification up-to-date with the implementation takes time and effort
- Yet another thing to learn: Can everyone in the project understand the specification?
- What is the risk of a failure versus the time spent in specifying the software?

UMass Boston

Language λ_S

Evaluating function declarations

A great way to learn is to **implement**

How to implement a programming language?

> In this class, we will learn how to implement an **interpreter**. In CS451/651 you can learn how to implement a compiler.

1. *The interpreter:*
    1. **a parser** converts the source code into an abstract-syntax-tree (a logical representation of the program)
    2. **an interpreter** executes the abstract-syntax-tree
2. *The compiler:*
    1. **a parser** converts the source code into an abstract-syntax-tree
    2. **a compiler** converts the abstract-syntax-tree into assembly (through a series of steps)

UMass Boston

What is an interpreter

An interpreter executes programs of a given language

- Usually, an interpreter executes a logical representation of the source code (known as the **abstract syntax**).
- An interpreter repeatedly executes (evaluates) one instruction (expression) at a time, until the program terminates.

Did you know?

> - Python, for instance, is known as an interpreter, but it actually compiles Python into an assembly-like language that is then interpreted (executed). Racket works the same way as Python. Java is also executed in the same way, but compilation is performed by the user.
> - Additionally, for performance reasons, some interpreters perform *just-in-time compilation*, which dynamically translates (compiles) small parts of the language as machine code that is executed directly by the CPU.

UMass Boston

Text file:

```
((lambda (x) x) 3)
```

⇓

Concrete syntax:

```
'((lambda (x) x) 3)
```

⇓

Abstract syntax:

```
(r:apply (r:lambda (list (r:variable 'x))
(list (r:variable 'x))) (list (r:number 3)))
```

⇓

Evaluation: `(r:number 3)`

Syntax of λ_S

Concrete Syntax of λ_S

- The concrete syntax dictates the syntactic structure of a program (how do we represent a number, a variable, etc). This is *not* the focus of our course (refer to CS451/651).
- We can easily sidestep the parsing issue by reminding ourselves that `quote` can serialize Racket expressions into a data structure.

UMass Boston

Abstract Syntax

In TypedRacket

```
(define-type s:value (U s:number s:lambda))
(define-type s:expression (U s:value s:variable s:apply))

(struct s:number ([value : Number]) #:transparent)
(struct s:variable ([name : Symbol]) #:transparent)
(struct s:lambda ([param : s:variable] [body : s:expression]) #:transparent)
(struct s:apply ([func : s:expression] [arg : s:expression]) #:transparent)
```

Mathematically

$$e ::= v \mid x \mid (e_1\ e_2) \qquad v ::= n \mid \lambda x.e$$

The abstract syntax of λ_S

A formal notation

$$e ::= v \mid x \mid (e_1\ e_2) \qquad v ::= n \mid \lambda x.e$$

- An expression is represented by letter e
- A value is represented by letter v
- A variable is represented by letter x
- A function application is represented by notation $(e_1\ e_2)$, where e_1 is an expression, and e_2 is another expression. The subscript numbers are just a way to distinguish each expression.
- A number is represented by letter n
- A function declaration is represented by $\lambda x.e$

UMass Boston

Semantics of λ_S

λ_S

Syntax

$$e ::= v \mid x \mid (e_1\ e_2) \qquad v ::= n \mid \lambda x.e$$

Semantics

$$v \Downarrow v \ \texttt{(E-val)} \qquad \frac{e_f \Downarrow \lambda x.e_b \qquad e_a \Downarrow v_a \qquad e_b[x \mapsto v_a] \Downarrow v_b}{(e_f\ e_a) \Downarrow v_b}\ \texttt{(E-app)}$$

Did you know?

- The cornerstone of functional programming, and a foundation of logic (and mathematics)
- The λ-calculus can be used to simulate any Turing machine
- Invented in 1930, by Alonzo Church

> Since λ_S is Turing complete, can you write a non-terminating program in λ_S?

UMass Boston

Example

$$\dfrac{\dfrac{}{\lambda x.x \Downarrow \lambda x.x} \qquad \dfrac{}{10 \Downarrow 10} \qquad \dfrac{x[x \mapsto 10] = 10 \qquad 10 \Downarrow 10}{x[x \mapsto 10] \Downarrow v_b}}{((\lambda x.x)\ 10) \Downarrow 10}$$

We show that evaluating $((\lambda x.x)\ 10)$ returns 10:

1. The expression is a function application, so we must apply rule **`E-app`**
2. We evaluate function $\lambda x.x$ with rule **`E-val`**, which is a value and therefore get the same value back
3. We evaluate argument 10 with rule **`E-val`**, which is also a value and therefore we get the same value back
4. Finally, we take the body of the function x and substitute variable x by the number 10 and evaluate 10, which because it is a value we get 10 back.

λ_S semantics

We will use a more familiar notation. We define the evaluation function $\texttt{eval}(e) = v$ that takes an expression e and returns a value v.

Rule **(E-val)**

$$\texttt{eval}(v) = v$$

Rule **(E-app)**

$$\frac{\texttt{eval}(e_f) = (\lambda(x)\ e_b) \qquad \texttt{eval}(e_a) = v_a \qquad \texttt{eval}(\texttt{subst}(e_b, x, v_a)) = v_b}{\texttt{eval}((e_f\ e_a)) = v_b}$$

Mathematical notation

$$v \Downarrow v \ \texttt{(E-val)} \qquad \frac{e_f \Downarrow \lambda x.e_b \qquad e_a \Downarrow v_a \qquad e_b[x \mapsto v_a] \Downarrow v_b}{(e_f\ e_a) \Downarrow v_b}\ \texttt{(E-app)}$$

Code notation

Rule **(E-val)**

$$\texttt{eval}(v) = v$$

Rule **(E-app)**

$$\frac{\texttt{eval}(e_f) = (\lambda(x)\ e_b) \qquad \texttt{eval}(e_a) = v_a \qquad \texttt{eval}(\texttt{subst}(e_b, x, v_a)) = v_b}{\texttt{eval}((e_f\ e_a)) = v_b}$$

λ_S semantics, informally

Our objective is to evaluate an expression.

1. If the expression e is a value, then we are in the base case, and we return value e.
2. If the expression is a variable x, this is an error, report it as such.
3. Otherwise, we have a function application $(e_f\ e_a)$. Recursively evaluate function e_f down to a value. Ensure that the result is a function declaration, say $(\texttt{lambda}(x)\ e_d)$.
4. Recursively evaluate the argument of the function e_a down to a value v_a.
5. Substitute variable x by value v_a in e_d, say $e_d[x \mapsto v_a]$. Recursively evaluate $e_d[x \mapsto v_a]$.

UMass Boston

Variable substitution

Variable substitution, formally

$$
\begin{aligned}
n[x \mapsto v] &= n \\
x[x \mapsto v] &= v \\
y[x \mapsto v] &= y \quad \text{if } x \neq y \\
(\lambda x.e)[x \mapsto v] &= \lambda x.e \\
(\lambda y.e)[x \mapsto v] &= \lambda y.(e[x \mapsto v]) \quad \text{if } x \neq y \\
(e_1\ e_2)[x \mapsto v] &= (e_1[x \mapsto v]\ e_2[x \mapsto v])
\end{aligned}
$$

Variable substitution, informally

Objective: substitute variable x by value v in expression e, notation $e[x \mapsto v]$.

1. If the expression is a number, say n, then return n.
2. If the expression is a variable y and $x = y$, then return v. Otherwise, return y.
3. If the expression is a function call $(e_1\ e_2)$, then return $(e_1[x \mapsto v]\ e_2[x \mapsto v])$. That is, recursively substitute each sub-expression.
4. If the expression is a function declaration $(\lambda(y)\ e)$, then . . .

What should we do?

Test-case utility function

Function `(check-r:eval? exp val)` is given in the template to help you test your code effectively.

> The use of `check-r:eval?` is **optional**. You are encouraged to play around with `r:eval` directly.

1. The first parameter is an S-expression that represents a valid λ **expression**
2. The second parameter is an S-expression that represents a valid λ **value**

UMass Boston

Test-cases

```
; a number is a value, so we just return it
(check-r:eval? 1 1)
; a lambda is a value, so we just return it
(check-r:eval? '(lambda (x) x) '(lambda (x) x))
; function application
(check-r:eval? '((lambda (x) x) 10) 10)
; function application that returns a lambda and replaces a variable
(check-r:eval? '((lambda (y) (lambda (x) y)) 1) '(lambda (x) 1))
```

More examples

- The University of Birmingham: Principles of Programming Languages 2009
- Church encoding, Wikipedia