

---

## Where are We

- We have covered enough basics to focus more on concepts now
- Before Scheme: Equivalence, parametric polymorphism, type inference, modules/abstract-types
- Homework 3: due Wednesday 4/26
- Midterm: Next Friday 4/28

## CSE 341: Programming Languages

Spring 2006

Lecture 11 — Equivalence and Syntactic Sugar

CSE 341 Spring 2006, Lecture 11

1

CSE 341 Spring 2006, Lecture 11

2

---

## Function-Call Efficiency

First: Function calls take constant ( $O(1)$ ) time, so until you're using the right algorithms and have a critical *bottleneck*, forget about it.

That said, ML's "all functions take one argument" can be inefficient in general:

- Create a new  $n$ -tuple
- Create a new function closure

In practice, implementations *optimize* common cases. In some implementations,  $n$ -tuples are faster (avoid building the tuple). In others, currying is faster (avoid building intermediate closures).

In the  $< 1$  percent of code where detailed efficiency matters, you program against an implementation. Worrying about this stuff at the wrong stage and for the wrong code is a bad idea.

CSE 341 Spring 2006, Lecture 11

3

---

## Equivalence

"Equivalence" is a fundamental programming concept

- Code maintenance / backward-compatibility
- Program verification
- Program optimization
- Abstraction and strong interfaces

But what does it mean for an expression (or program)  $e1$  to be "equivalent" to expression  $e2$ ?

CSE 341 Spring 2006, Lecture 11

4

## Equivalence I: where?

Context (i.e., “where equivalent”)

- Given where  $e1$  occurs in a program  $e$ , replacing  $e1$  with  $e2$  makes a program  $e'$  equivalent to  $e$
- At any point in any program, replacing  $e1$  with  $e2$  makes an equivalent program.

The latter (contextual equivalence) is much more interesting.

For the former, the body of an unused function body is equivalent to everything (that typechecks).

CSE 341 Spring 2006, Lecture 11

5

## Accounting for “Effects”

Consider whether  $\text{fn } x \Rightarrow e1$  and  $\text{fn } x \Rightarrow e2$  are totally contextually equivalent.

Is this enough? For all environments,  $e1$  terminates and evaluates to  $v$  under the environment if and only if  $e2$  terminates and evaluates to  $v$  under the environment.

Functions produce *values*; may also produce (*side-*) *effects*. Consider both!

Purely functional languages have fewer/none, but ML is not purely functional (mutation, exceptions, printing, files, ...)

In real languages, contextual equivalence usually requires many things. Nonetheless, “equivalence” usually means total contextual equivalence for practical purposes (optimization, reasoning about correctness, etc.).

CSE 341 Spring 2006, Lecture 11

7

## Equivalence II: how?

“how equivalent”

- “partial”:  $e$  and  $e'$  are partially equivalent if on any input where both give an output, they give the *same* output
- “total”: partial plus  $e$  and  $e'$  have the same termination behavior
- efficiency:  $e$  and  $e'$  are totally equivalent and one never takes more than (for example)  $c$  times longer than the other (or uses much more space or ...)
- syntactic notions:  $e$  and  $e'$  differ only in whitespace and comments (for example)

Key notion: what is observable? (memory, clock, REP-loop, file-system, ...)

CSE 341 Spring 2006, Lecture 11

6

## Syntactic Sugar

When all expressions using one construct are totally equivalent to another more primitive construct, we say the former is “syntactic sugar”.

- Makes language definition easier
- Makes language implementation easier
- (But may make error messages more cryptic...)

Examples:

```
e1 and also e2      ↦ if e1 then e2 else false
if e1 then e2 else e3 ↦ case e1 of true => e2 | false => e3
fun f x y = e       ↦ val f = fn x y => e
#1 e                ↦ let val (x,...) = e in x end
```

CSE 341 Spring 2006, Lecture 11

8

## More sugar

If we *ignore types*, then we have even more sugar:

```
let val p = e1 in e2 end  $\mapsto$  (fn p => e2) e1
```

In fact, if we let every program type-check (or just use one big datatype), then a language with just functions and function application is as powerful as ML or Java (in the Turing Tarpit sense).

This language is called “lambda calculus” – we’ll learn a bit more about it later.

CSE 341 Spring 2006, Lecture 11

9

## Systematic renaming requires care

Is  $\text{fn } x \Rightarrow e1$  equivalent to  $\text{fn } y \Rightarrow e2$  where  $e2$  is  $e1$  with every  $x$  replaced by  $y$ ?

What if  $e1$  is  $y$ ? (More generally, contains free  $y$ ?)

What if  $e1$  is  $\text{fn } x \Rightarrow x$ ? (More generally, contains bound  $x$ ?)

Need caveats:  $\text{fn } x \Rightarrow e1$  is equivalent to  $\text{fn } y \Rightarrow e2$  where  $e2$  is  $e1$  with every free  $x$  replaced by  $y$  and  $y$  is not free in  $e1$ , and no free  $x$  occurs within the scope of a binding for  $y$  (capture; e.g.:  $\text{fn } x \Rightarrow \text{let } y = 2 \text{ in } x + y \text{ end}$ ; see also next slide.)

Note: We can provide a very precise recursive (meta-)definition of *free variables* in an expression.

Next: Is  $(\text{fn } x \Rightarrow e1) e2$  equivalent to  $e3$  where  $e3$  is  $e1$  with every  $x$  replaced by  $e2$ ?

CSE 341 Spring 2006, Lecture 11

11

## Equivalences for Functions

While sugar defines one construct in terms of another, there are also important notions of *meaning-preserving* changes involving functions and bound variables.

They’re so important that a goal of language design is that a language supports them.

But the correct definitions are subtle.

First example: systematic renaming

Is  $\text{fn } x \Rightarrow e1$  equivalent to  $\text{fn } y \Rightarrow e2$  where  $e2$  is  $e1$  with every  $x$  replaced by  $y$ ?

CSE 341 Spring 2006, Lecture 11

10

## Argument Substitution

Is  $(\text{fn } x \Rightarrow e1) e2$  equivalent to  $e3$  where  $e3$  is  $e1$  with every  $x$  replaced by  $e2$ ?

- Every free  $x$  (of course).
- A free variable in  $e2$  must not be bound at an occurrence of  $x$ . (Called “capture”.)
  - Always satisfiable by renaming bound variables.
- Evaluating  $e2$  must have no effects (printing, exceptions, infinite-loop, etc.)
  - Closely tied to the rule that arguments are evaluated to values before function application. (Not true for all languages)
  - In ML, many expressions have no such effects ( $x, \#\text{foo } x, \dots$ ); much fewer in Java.
- Efficiency? Could be faster or slower. (Why?)

CSE 341 Spring 2006, Lecture 11

12

## Unnecessary Function Wrapping

---

A common source of bad style for beginners

Is `e1` equivalent to `fn x => e1 x`?

Sure, provided:

- `e1` is effect-free
- `x` does not occur free in `e1`

Example:

```
List.map (fn x => SOME x) lst  
List.map SOME lst
```

CSE 341 Spring 2006, Lecture 11

13

## Summary

---

We breezed through some core programming-language facts and design goals:

- Definition of equivalence depends on observable behavior
- Syntactic sugar “makes a big language smaller” by *defining* constructs in terms of equivalence
- Notion of free and bound variables crucial to understanding function equivalence.
- Three common forms of function equivalence:
  - Systematic Renaming
  - Argument Substitution
  - Unnecessary Function Wrapping

CSE 341 Spring 2006, Lecture 11

14