Outline

- Program generation
  - Programs that generate programs
- The Scheme programming language
  - Dynamically typed functional language
  - Concrete syntax = abstract syntax
- Closures in C# and Java
- Program generation in Scheme
  - Two-level languages
  - Distinguishing binding-times
- Runtime bytecode generation in C#/.NET
The power(n, x) function

- Computing $x^n$ efficiently in Java/C#
- Using that $x^{2m} = (x^2)^m$ and $x^{m+1} = x^m x$

```java
public static int Power(int n, int x) {
    int p;
    p = 1;
    while (n > 0) {
        if (n % 2 == 0)
            { x = x * x; n = n / 2; }
        else
            { p = p * x; n = n - 1; }
    }
    return p;
}
```

Specialized power(n,x) for n=5

- What if we must compute $x^5$ for many $x$
- Then this function would be faster:
  - (Why?)

```java
public int Power_5(int x) {
    int p;
    p = 1;
    p = p * x;
    x = x * x;
    x = x * x;
    x = x * x;
    p = p * x;
    return p;
}
```
Generator of specialized power\((n,x)\) functions

```java
public static void PowerTextGen(int n) {
    System.out.println("public int Power\(_\) \+ n \+ \"(int x) \{");
    System.out.println(" int p;\");
    System.out.println(" p = 1;\");
    while (n > 0) {
        if (n % 2 == 0) {
            System.out.println(" x = x \* x;\"");
            n = n / 2;
        } else {
            System.out.println(" p = p \* x;\"");
            n = n - 1;
        }
    }
    System.out.println(" return p;\");
    System.out.println("\}");
}
```

Binding times in Power\((n,x)\)

- **green**=static=early, **red**=dynamic=late

```java
public static int Power(int n, int x) {
    int p;
    p = 1;
    while (n > 0) {
        if (n % 2 == 0)
            { x = x \* x; n = n / 2; }
        else
            { p = p \* x; n = n - 1; }
        }
    return p;
}
```

- The generator performs the green code and prints (emits) the red code
The Scheme language

- Design by Guy L Steele 1978
  - Plus a revolutionary compilation technique
  - In his Master’s thesis from MIT
  - Co-author of *Java Language Specification*
  - Now designing a new language “Fortress”
- Descends from Lisp (McCarthy 1960)
- A higher-order functional language
- Like Standard ML but no static types
- Simple syntax inherited from Lisp

Scheme expressions

- Compute 7+9:
  $(+ \text{ 7 9})$
- Compute 7*9+13:
  $(+ (* \text{ 7 9}) \text{ 13})$
- Define variable x to be 42:
  $(\text{define x 42})$
- Define variable x to be value of 7+9:
  $(\text{define x (+ 7 9)})$
- If x<15 then $x^2$ else x-15:
  $(\text{if } (< \text{ x 15}) (* \text{ x x}) (\text{- x 15}))$
Scheme function definitions

• Defining the function \( f(x) = x \times 3 + 7 \):

\[
\text{(define (f x) (+ (* x 3) 7))}
\]

• Same in C/C++/Java/C#:

```c
int f(int x) { return x*3+7; }
```

• Calling the function on argument 10:

```scheme
(f 10)
```

• Defining a recursive function:

\[
\text{(define (fac n)}
\text{  (if (= n 0)}
\text{    1}
\text{  (* n (fac (- n 1)))}}
\text{)}
\]

The power function in Scheme

\[
\text{(define (sqr x) (* x x))}
\text{(define (power n x)}
\text{  (if (> n 0)}
\text{    (if (eq? (remainder n 2) 0)}
\text{      (sqr (power (/ n 2) x))}
\text{      (* x (power (- n 1) x))}}
\text{    1)}
\text{)}
\]

> (power 10 2)
1024
> (power 97 2)
158456325028528675187087900672
Scheme anonymous functions

- The anonymous function \( x \rightarrow x^3 + 7 \)
  \[(\text{lambda } (x) (+ (* x 3) 7))\]
- Applying the function to argument 10:
  \[(\text{lambda } (x) (+ (* x 3) 7))\ 10\]
- Anonymous functions in other languages
  
  - Standard ML, 1978
    \[\text{fn } x \Rightarrow x^3 + 7\]
  
  - Lambda calculus, 1936
    \[x \Rightarrow x^3 + 7\]
  
  - C# 3.0
    \[\lambda x. x^3 + 7\]
  
  - C# 2.0
    \[\text{delegate(int } x)\ \{\ \text{return } x^3 + 7; \ \}\]

Closures

- An anonymous function may use a variable from an enclosing scope
  \[(\text{define } (\text{makeadd } y)\)
  \[\quad (\text{lambda } (x) (+ x y))\]
  \[
  )\]
- A closure must be built for the function:
  \[(\text{define } g (\text{makeadd } 7))\]
  \[(g\ 42)\]
  \[
  g\]
  \[
  \begin{array}{|c|}
  \hline
  \text{Closure} \\
  \hline
  y = 7 \\
  \hline
  (\text{lambda } (x) (+ x y)) \\
  \hline
  \end{array}
  \]
Scheme data: lists and pairs

- Scheme data are either
  - atoms (numbers, Booleans, symbols ...) or
  - S-expressions: pairs, lists
- The list containing 11, 22 and 33:
  '(11 22 33)
- Defining xs to be that list:
  (define xs '(11 22 33))
- The first element of xs:
  (car xs)
- The rest of xs:
  (cdr xs)
- The second element of xs:
  (car (cdr xs))

Pairs and lists: s-expressions

- Structured data are built from cons cells
- A cons cell’s components are car and cdr:

![Diagram of cons cell with car and cdr]

- Creating a new cons cell:
  (cons 44 (+ 44 11))  44 55
- A constant cons cell:
  '(44 . 55)  44 55
A list is a special s-expression

• A list (11 22 33) is a right-linear tree ending in nil, the empty list ()

```
  11
  |
  22
  |
  33
  |
  nil
```

• Four ways to build the list:

```
'(11 22 33)
'(11 . (22 . (33 . ()))))
(list 11 22 33)
(cons 11 (cons 22 (cons 33 ())))
```

Ten-minute exercise

• Assume \( \text{x} \text{s} \) is the list (11 22 33)

• Write Scheme expressions
  - for extracting the third element from \( \text{x} \text{s} \)
  - for extracting the list containing the third element from \( \text{x} \text{s} \)
  - for computing the sum of the first and second element of \( \text{x} \text{s} \)
  - for testing whether first element of \( \text{x} \text{s} \) is positive

• Write Scheme expression for

\[ 11 + x \times (22 + x \times (33 + x \times 0)) \]
Some list-processing functions

- Length of a list:

\[
\text{(define (len xs)}
  \text{(if (null? xs)}
    \text{0)
  \text{(+ 1 (len (cdr xs)))}})
\]

- Sum of a list’s elements:

\[
\text{(define (sum xs)}
  \text{(if (null? xs)}
    \text{0)
  \text{(+ (car xs) (sum (cdr xs)))}})
\]

Some tree-processing functions

- Representing a tree:

\[
\text{(define t)}
  \text{’(11 . (22 . 33))}
\]

- Depth of a tree:

\[
\text{(define (depth t)}
  \text{(if (pair? t)}
    \text{(+ 1 (max (depth (car t))}
      \text{(depth (cdr t)))})
  \text{0)}
\]

Higher-order functions

- Mapping a function over a list:

```scheme
(define (map f xs)
  (if (null? xs)
      ()
      (cons (f (car xs)) (map f (cdr xs))))
)
```

- Example use:

```scheme
(define xs '(11 22 33))
(map (lambda (x) (* 2 x)) xs)
```

Running Scheme programs

- Some Scheme implementations:
  - PLT Scheme
  - Jaffer’s SCM
  - MIT/GNU Scheme
  - Chez Scheme (commercial license)
  - Petite Chez Scheme

- Documentation, lots, among which:
Closures in C# (since 2.0)

• C# 2.0 anonymous function syntax

```csharp
delegate int I2I(int x);
I2I f = delegate(int x) { return x*3 + 7; };
int res = f(32);
```

• C# 3.0 lambda syntax

```csharp
Func<int,int> f = x => x*3 + 7;
int res = f(32);
```

Closures may capture variables

• Local variable y is captured in delegate:

```csharp
I2I MakeAdd(int y) {
    return delegate(int x) { return x+y; };
}
```

```csharp
I2I g = MakeAdd(7);
int res = g(42);
```
C#: Captured variables may be updated

- Apply \( f \) to every element of the array:

```csharp
void Apply(int[] arr, I2I f) {
    foreach (int x in arr)
        f(x);
}
```

- Compute the sum of array’s elements:

```csharp
void M() {
    int[] arr = ...
    int sum = 0;
    Apply(arr, delegate(int x) { sum += x; });
    Console.WriteLine(sum);
}
```

C#: Captures lvalue of variable \textit{sum}

Closures in Java (since 1.2)

- Use anonymous inner classes

```java
interface I2I {
    int invoke(int x);
}
```

```java
I2I f =
    new I2I() {
        public int invoke(int x) {
            return x*3+7;
        }
    };
int res = f.invoke(32);
```

type \textit{I2I} = int->int

Closure

More precisely: Instance of anonymous inner class that implements \textit{I2I}
A Java closure capturing variables

• Captured variables must be final:

```java
I2I MakeAdd(final int y) {
    return new I2I() {
        public int invoke(int x) { return x+y; }
    };
}
```

```java
I2I g = MakeAdd(7);
int res = g.invoke(42);
```

• So using an `Apply(int[] arr, I2I f)` method in Java is more cumbersome – exercise

Implementation of C# closures

• A C# closure captures the lvalue:
  - stores the variable in a context object on heap
  - both method and closure use the context object
Implementation of Java closures

- A Java closure captures the rvalue:
  - stores the variable’s value in the closure
  - method and closure each have a copy

```
stack
y = 7
invoke()

heap
: I2I
y = 7
invoke()
```

Now two copies of y; is that a problem?

Back to Scheme: Abstract syntax and eval

- Abstract syntax for (+ 2 3) is '(+ 2 3)
- Abstract syntax can be evaluated by eval

```
> (+ 2 3)
5
> '(+ 2 3)
(+ 2 3)
> (eval '(+ 2 3))
5
> (define myexpr '(+ (* x x) 7))
> myexpr
(+ (* x x) 7)
> (define x 10)
> (eval myexpr)
107
```
Constructing abstract syntax

• Abstract syntax can be built with list and quote:

> (define e '(* x 3))
> (list '+ e '7)
(+ (* x 3) 7)
> (define (addsqr y)(list '+ 'x (* y y)))
> (addqrgen 7)
(+ x 49)
> (define (addsqrdef y)
  (list 'define '(f x)
    (list '+ 'x (* y y)))))
> (addsqrdef 7)
(define (f x) (+ x 49))

Scheme quasiquotation: Comma and backquote

• Using list and quote can be confusing
• Backquote and comma make life easier
  – Backquote quotes everything except subexpressions preceded by comma
  – Comma “unquotes” a subexpression

> (define (addsqrdef y)
  `(define (f x) (+ x ,(* y y))))
> (addsqrdef 7)
(define (f x) (+ x 49))
Generator of specialized power functions

\[
(\text{define} \ (\text{powgen} \ n)
\begin{array}{l}
(\text{if} \ (> \ n \ 0) \\
\hspace*{1em} (\text{if} \ (\text{eq?} \ (\text{remainder} \ n \ 2) \ 0) \\
\hspace*{2em} (\text{sqr} \ ,(\text{powgen} \ (/ \ n \ 2))) \\
\hspace*{2em} (* \ x \ ,(\text{powgen} \ (- \ n \ 1)))) \\
\hspace*{1em} 1)
\end{array}
)\\n\]

\[
(\text{define} \ (\text{mkpower} \ n) \\
(\text{eval} \ '(\text{define} \ (\text{pow} \ x) \ ,(\text{powgen} \ n))))
\]

Two-level languages and binding-times

- Scheme with backquote and comma is a two-level language:
  - Backquote: dynamic (late) computation
  - Comma: static (early) computation in dynamic context

\[
(\text{define} \ (\text{power} \ n \ x) \\
(\text{if} \ (> \ n \ 0) \\
\hspace*{1em} (\text{if} \ (\text{eq?} \ (\text{remainder} \ n \ 2) \ 0) \\
\hspace*{2em} (\text{sqr} \ (\text{power} \ (/ \ n \ 2) \ x)) \\
\hspace*{2em} (* \ x \ (\text{power} \ (- \ n \ 1) \ x)) \\
\hspace*{1em} 1)
)\\n)
Ten-minute exercise

• Ex 1: Use backquote and comma to write an expression that builds
  
  \[(+ \ y \ 2^{97})\]
  where the value of \(2^{97}\) must be computed and inserted

• Ex 2: Assume \(x\) is static and \(y\) dynamic in
  
  \[(+ (*) 11 \ x) (*) 22)\]

  – Mark static and dynamic parts (green, red)
  – Write expression that builds the above for any given value of \(x\)

Partial evaluation

• Proposed by Yoshihiko Futamura 1970
• Initially studied in USSR and Sweden
• Idea:
  – Assume \(p\) is a two-input program \(p(\text{in1, in2})\)
  – But we have only part of the input, \(\text{in1}\)
  – Then we cannot run (evaluate) program \(p\)
  – But can partially evaluate it \([\text{mix}](p,\text{in1})\)
  – We don’t get a result, but a new program \(r\)
  – Running \(r\) on \(\text{in2}\) will then give the result
• Two-stage execution …
**Interpreter, compiler and partial evaluator (mix)**

- Let p be a program and inp input data
- Running p on input inp gives output out
  - In symbols: \([p](\text{inp}) = \text{out}\)
- An interpreter int is a program such that
  - \([\text{int}](p, \text{inp}) = \text{out}\)
- A compiler comp is a program such that
  - If target = \([\text{comp}](p)\) then \([\text{target}](\text{inp}) = \text{out}\)
- Now let p be a two-input program, \([p](\text{in1},\text{in2}) = \text{out}\)
- A partial evaluator is a program mix such that
  - If \(r = [\text{mix}](p,\text{in1})\) then \([r](\text{in2}) = \text{out}\)
- The partial evaluator runs p on only part of its input, giving a residual program r

---

**The three Futamura projections**

- First: compilation
  - If target = \([\text{mix}](\text{int},p)\) then \([\text{target}](d) = r\)
- Second: compiler generation
  - If comp = \([\text{mix}](\text{mix},\text{int})\) then \([\text{comp}](p) = t\)
- Third: compiler generator generation
  - If cogen = \([\text{mix}](\text{mix},\text{mix})\) then \([\text{cogen}](\text{int}) = \text{comp}\)
- There’s no Fourth Futamura projection:
  - Because \([\text{cogen}](\text{mix}) = \text{cogen}\)
This works in practice

- Self-application is non-trivial in practice
- First success 1985 by Jones, Sondergaard, Sestoft at DIKU
- Had to invent binding-time analysis
- Try the Scheme0 partial evaluator

Runtime code generation in C#

- Could generate C# code, then call compiler
- But compiler must be installed, big overhead
- Better generate .NET bytecode directly
- Example: To generate

```csharp
public static int MyMethod(int x) {
    Console.WriteLine("MyMethod() was called");
    return x + 42;
}
```

- Use an ILGenerator `ilg` to make the body

```csharp
ilg.EmitWriteLine("MyMethod() was called");
ilg.Emit(OpCodes.Ldarg_0);
ilg.Emit(OpCodes.Ldc_I4, 42);
ilg.Emit(OpCodes.Add);
ilg.Emit(OpCodes.Ret);
```
The full story

- The ILGenerator belongs to a DynamicMethod

```csharp
DynamicMethod methodBuilder =
    new DynamicMethod("MyMethod",
    typeof(int),
    new Type[] { typeof(int) },
    typeof(String).Module);
ILGenerator ilg = methodBuilder.GetILGenerator();
... use ilg as on previous slide ...
```

- Creating and calling the new method:

```csharp
int res;
I2I mm = (I2I)methodBuilder.CreateDelegate(typeof(I2I));
res = mm(17);
```

```csharp
public delegate int I2I(int x); as before
```