Today

- Program generation
  - Programs that generate programs
- The Scheme programming language
  - Dynamically typed functional language
  - Concrete syntax = abstract syntax
- Program generation in Scheme
  - Two-level languages
  - Distinguishing binding-times
- Partial evaluation: automatic program specialization
The power($n$, $x$) function

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

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

Example:

\[
3^5 = 3 \cdot 3^4 = 3 \cdot (3^2)^2 = 3 \cdot 9^2 = 3 \cdot 81 = 243
\]

Specialized power($n$, $x$) for $n=5$

- What if we must compute $x^5$ for many $x$
- So we know, statically, that $n=5$, but don’t know $x$
- Then a specialized function Power_5($x$)
  - would compute exactly the same result
  - but would be faster (why?)

```java
static double Power_5(double x) {
    double p = 1;
    p = p * 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("static double Power_" + n + "(double x) {\n    System.out.println("  double p;\n    System.out.println("  p = 1;\n    while (n > 0) {\n        if (n % 2 == 0) {\n            System.out.println("  x = x * x;\n            n = n / 2;\n        } else {\n            System.out.println("  p = p * x;\n            n = n - 1;\n        }\n    System.out.println("  return p;\n    System.out.println("});\n}
```
The Scheme language

• Design by Guy L Steele 1978
  – Plus a revolutionary compilation technique
  – Master’s thesis from MIT
  – Co-author of Java Language Specification
  – Now designing a new language “Fortress”

• Scheme descends from Lisp (McCarthy 1960)
• A higher-order functional language
• Like Scala, ML, F# but no static types
• Very simple syntax, lots of parentheses

Scheme expressions

• Compute 7+9:
  (+ 7 9)

• Compute 7*9+13:
  (+ (* 7 9) 13)

• Define variable x to be 42:
  (define x 42)

• Define variable x to be value of 7+9:
  (define x (+ 7 9))

• If x<15 then x^2 else x-15:
  (if (< x 15) (* x x) (- 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:

  ```scheme
  (define (fac n)
    (if (= n 0)
      1
      (* n (fac (- n 1)))
    ))
  ```

The power function in Scheme

```scheme
(define (sqr x) (* x x))

(define (power n x)
  (if (> n 0)
    (if (eq? (remainder n 2) 0)
      (sqr (power (/ n 2) x))
      (* x (power (- n 1) x))
    )
  )
)
```

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

- The anonymous function \( x \rightarrow x \cdot 3 + 7 \)
  
  \[(\text{lambda} \ (x) \ (+ \ (* \ x \ 3) \ 7))\]

- Applying the function to argument 10:
  
  \[(((\text{lambda} \ (x) \ (+ \ (* \ x \ 3) \ 7)) \ 10)\]

- Anonymous functions in other languages

  - fun x -> x*3+7  
    F#, Ocaml
  
  - fn x => x*3+7  
    Standard ML, 1978
  
  - delegate(int x) { return x*3+7; }  
    C# 2.0
  
  - x => x*3+7  
    C# 3.0, Scala
  
  - \( \lambda \ x . \ x \cdot 3 + 7 \)  
    Lambda calculus, 1936

Closures in Scheme

- An anonymous function may use a variable from an enclosing scope

  \[(\text{define} \ (\text{makeadd} \ y) \n  \quad (\text{lambda} \ (x) \ (+ \ x \ y))\)\]

- A closure must be built for the function:

  \[(\text{define} \ g \ (\text{makeadd} \ 7)) \n  (g \ 42)\]

  g’s value is a closure

<table>
<thead>
<tr>
<th>g</th>
<th>Closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = 7</td>
<td></td>
</tr>
<tr>
<td>(\text{lambda} \ (x) \ (+ \ x \ y))</td>
<td></td>
</tr>
</tbody>
</table>
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:
  \[(\text{define } xs \ (11 \ 22 \ 33))\]
- The first element of xs:
  \[(\text{car } xs)\]
- The rest of xs:
  \[(\text{cdr } xs)\]
- The second element of xs:
  \[(\text{car } (\text{cdr } xs))\]

Pairs and lists: s-expressions

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

![Diagram of a cons cell with car and cdr](Diagram)

- Creating a new cons cell:
  \[(\text{cons } 44 \ (+ \ 44 \ 11))\]
- A constant cons cell:
  \[(44 . 55)\]
A list is a special s-expression

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

```
   11
     ↓
    22
      ↓
     33 nil
```

- Four ways to build that list:

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

Ten-minute exercise

- Assume `xs` is the list (11 22 33)
- Write Scheme expressions
  - for extracting the third element from `xs`
  - for extracting the list containing only the third element
  - for computing the sum of the first and second element
  - for testing whether first element is positive
- Write a Scheme expression corresponding to 11+x*(22+x*(33+x*0))
Some list-processing functions

• Length of a list:

```lisp
(define (len xs)
  (if (null? xs)
      0
      (+ 1 (len (cdr xs))))
)
```

• Sum of a list’s elements:

```lisp
(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs))))
)
```

Some tree-processing functions

• Representing a tree:

```lisp
(define t
  '(11 . (22 . 33))
)
```

• Depth of a tree:

```lisp
(define (depth t)
  (if (pair? t)
      (+ 1 (max (depth (car t))
                 (depth (cdr t))))
      0)
)
```
Higher-order functions

• Mapping a function over a list:

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

• Example use:

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

Running Scheme programs

• Some Scheme implementations:
  – Racket, formerly PLT Scheme
  – Jaffer’s SCM, for Linux, MacOS (and Windows)
  – MIT/GNU Scheme
  – Chez Scheme (commercial license)
  – Petite Chez Scheme
  – More at http://schemers.org/ > implementation

• Documentation, lots, among which:
Data representing expressions: Abstract syntax and the `eval` function

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

```scheme
> (+ 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:

```scheme
> (define ee '(* x 3))
> (list '+ ee '7)
(+ (* x 3) 7)
> (define (addsqr y) (list '+ 'x (* y y)))
> (addsqr 7)
(+ x 49)
> (define (addsqrd ef y)
    (list 'define '(f x)
        (list '+ 'x (* y y)))
> (addsqrd ef 7)
(define (f x) (+ x 49))
```

Build AST for function that adds $y^2$ to $x$

AST only
From AST to defined function

- The abstract syntax tree (AST) is just data
- To define a function, we must eval the AST
  - Just like javac followed by java; here just one step

```
> (add sqrdef 7)
(define (f x) (+ x 49))

> (f 10)
# ... undefined identifier: f

> (eval (addsqrdef 7))
> (f 10)
59
```

Scheme quasiquotation: Comma and backquote

- Using list and quote can be confusing
- Backquote and comma make life easier
  - Backquote quotes everything so it get constructed, not evaluated
  - Comma “unquotes” a subexpression so it gets evaluated, not constructed

```
> (define (addsqrdef y)
  `(define (f x) (+ x ,(* y y))))
> (addsqrdef 7)
(define (f x) (+ x 49))
> (eval (addsqrdef 7))
```
Generator of specialized power \( \text{power}(n,x) \) functions

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

(\text{define} \ (\text{mkpow} \ n) \\
  (\text{eval} \ `(\text{define} \ (\text{pow} \ x), (\text{powergen} \ 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) \\
    (\text{if} \ (\text{eq?} \ (\text{remainder} \ n \ 2) \ 0) \\
      (\text{sqr} \ (\text{power} \ (/ \ n \ 2) \ x)) \\
      (* \ x \ (\text{power} \ (- \ n \ 1) \ x)) \\
    ) \\
    1)
  )
)
**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 (using `power`) and inserted as number

- Ex 2: Assume \(x\) is static and \(y\) dynamic in
  \[ (+ (\times \ 11 \ x) (\times \ y \ 22)) \]
  - 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
- Studied in USSR and Sweden in the 1970'es
- Idea:
  - Assume \(p\) is a two-input program \(p(\text{in}1,\text{in}2)\)
  - But we have only part of the input, \(\text{in}1\)
  - Then we cannot run (evaluate) program \(p\)
  - But can partially evaluate, or specialize, it
    \[ r = [\text{spec}](p,\text{in}1) \]
    - We don’t get a result, but a new program \(r\)
    - Running \(r\) on \(\text{in}2\) will then give the result
- Two-stage execution …
**Interpreter, compiler and partial evaluator (spec)**

- Let $s$ be a source program and $\text{inp}$ input data
- Running $s$ on input $\text{inp}$ gives output $\text{out}$
  - In symbols: $[s](\text{inp}) = \text{out}$
- An interpreter $\text{int}$ is a program such that
  - $[\text{int}](s, \text{inp}) = \text{out}$
- A compiler $\text{comp}$ is a program such that
  - If $\text{target} = [\text{comp}](s)$ then $[\text{target}](\text{inp}) = \text{out}$
- Now let $p$ be a two-input program,
  - $[p](\text{in}_1, \text{in}_2) = \text{out}$
- A partial evaluator is a program $\text{spec}$ such that
  - If $r = [\text{spec}](p, \text{in}_1)$ then $[r](\text{in}_2) = \text{out}$
- The partial evaluator runs $p$ on only part of its input, giving a residual program $r$

**The three Futamura projections**

- First: compilation
  - If $\text{target} = [\text{spec}](\text{int}, s)$ then $[\text{target}](\text{in}) = \text{out}$
- Second: compiler generation
  - If $\text{comp} = [\text{spec}](\text{spec}, \text{int})$ then $[\text{comp}](s) = \text{target}$
- Third: compiler generator generation
  - If $\text{cogen} = [\text{spec}](\text{spec}, \text{spec})$ then $[\text{cogen}](\text{int}) = \text{comp}$
- There’s no Fourth Futamura projection:
  - Because $[\text{cogen}](\text{spec}) = \text{cogen}$
This actually works!

- Self-application is non-trivial in practice
  - Avoid generating large trivial specializations
- Success 1985 at University of Copenhagen
- We invented *binding-time analysis*, a static analysis:
  - Which computations depend only on known data
  - Those may be performed at specialization time
- See *dataflow analysis* in Claus Brabrand's part of this course

http://www.itu.dk/people/sestoft/pebook/pebook.html

Example function: (power n x)

```scheme
((define (pow n x)
  (if (op equal? n 0)
      1
      (if (op even? n)
          (call pow (op quotient n 2) (op * x x))
          (op * x (call pow (op - n 1) x))))))

((define (((pow 2) s d) (n) (x))
  (ifs (ops equal? n 0)
    (lift 1)
    (ifs (ops even? n)
      (calld ((pow 2) s d)
        ((ops quotient n 2))
        ((opd * x x)))
      (opd * x (calls ((pow 2) s d) ((ops - n 1) (x)))))))))
```

"Scheme0" version of (power n x)

After binding-time analysis and annotation
(s=static, d=dynamic)
Specializing for n=97

\[
\begin{align*}
&> (\text{pretty (scheme (spec pa2 '(97))})) \\
&\{(\text{define (pow*sd-1 } x \text{) (* } x \text{ (pow*sd-2 (* } x \text{x)))}) \\\n&\{(\text{define (pow*sd-2 } x \text{) (pow*sd-3 (* } x \text{x)))}) \\\n&\{(\text{define (pow*sd-3 } x \text{) (pow*sd-4 (* } x \text{x)))}) \\\n&\{(\text{define (pow*sd-4 } x \text{) (pow*sd-5 (* } x \text{x)))}) \\\n&\{(\text{define (pow*sd-5 } x \text{) (pow*sd-6 (* } x \text{x)))}) \\\n&\{(\text{define (pow*sd-6 } x \text{) (* } x \text{ (pow*sd-7 (* } x \text{x)))}) \\\n&\{(\text{define (pow*sd-7 } x \text{) (* } x \text{ '1)))}
\end{align*}
\]

\[
> (\text{make 'power97 (spec pa2 '(97))}) \\
> (\text{power97 3}) \\
19088056323407827075424486287615602692670648963
\]

Generating and using a specializer

\[
\begin{align*}
&> (\text{define sann (monotate specializer '(s d))}) \\
&> (\text{define sp2 (spec sann (list pa2))}) \\
&\text{Compile and use it} \\
&> (\text{make 'powergen sp2}) \\
&> (\text{powergen '(97))}
\end{align*}
\]

Result is as before

\[
\{(\text{define (pow*sd-1 } x \text{) (* } x \text{ (pow*sd-2 (* } x \text{x)))}) \\
\{(\text{define (pow*sd-2 } x \text{) (pow*sd-3 (* } x \text{x)))}) \\
\{(\text{define (pow*sd-3 } x \text{) (pow*sd-4 (* } x \text{x)))}) \\
\{(\text{define (pow*sd-4 } x \text{) (pow*sd-5 (* } x \text{x)))}) \\
\{(\text{define (pow*sd-5 } x \text{) (pow*sd-6 (* } x \text{x)))}) \\
\{(\text{define (pow*sd-6 } x \text{) (* } x \text{ (pow*sd-7 (* } x \text{x)))}) \\
\{(\text{define (pow*sd-7 } x \text{) (* } x \text{ '1)))}
\]
Generating a specializer generator

> (define cc (spec sann (list sann)))

Use resulting specializer generator to generate a power specializer

> (make 'cogen cc)
> (define cp2 (cogen (list pa2)))

Result cp2 is identical to sp2

> (define ccc (cogen (list sann)))
> (equal? cc ccc)

Result ccc is identical to cc

Pitfalls of partial evaluation

- Sometimes nothing can be specialized
  - Eg x static (known) but n dynamic in power(n,x)
  - Static computation under dynamic control, dangerous
- Infinitely large "specialized" programs
- Very large specialized programs, no speedup
  - Subsequent compilation/code generation slow
  - Many instruction cache misses
- Inadequate binding-time separation
  - Variables that "should" be static become dynamic
- Lots of research on these problems since 1985
  - For Scheme, Prolog, C, ML, Java, C#, ...
  - The toy Scheme0 specializer used here is very simple
- Specialization, spreadsheets, graphics processors
What's next

- Monday 7 March: Runtime code generation in (Java and) C#; and Bastian's presentation
- Wednesday 9 March onwards: Extended static checking, by Joe Kiniry
- Exercises week 5: Scheme, program generation in Scheme, and runtime code generation in C#/CLI