Announcements

- Second homework problem sets has been posted. Solutions will be posted on sakai.
- Please check whether you have access to piazza on sakai. If not, let me know.
Review: Our TINY language

\[ x \in \text{Variables} \]
\[ n \in \text{Integers} \]
\[ c ::= n \mid \#t \mid \#f \mid + \mid - \mid * \mid / \quad \text{constants} \]
\[ v ::= c \mid (\text{lambda} (x \ldots) e) \quad \text{values} \]
\[ e ::= v \mid x \mid (e_1 \ldots e_k) \mid (\text{if} \ e_1 \ e_2 \ e_3) \quad \text{expressions} \]
\[ p ::= e \quad \text{program} \]

This simple functional language does not have constructs to define recursive functions.

Note: (lambda (x \ldots) e) is a function with one or more arguments (that’s what the “…” mean).

We want TINY to be a lexically scoped language.
Observations

Is there any difference between call-by-value and call-by-name in terms of

- efficiency – How many computation (reduction) steps?
- answers computed – Different answers?

Examples:

$$((\lambda (x) (+ x x)) ((\lambda(x) x) 4))$$

$$((\lambda (x) (+ 1 1)) ((\lambda(x) x) 4))$$

$$((\lambda (x) 1) ((\lambda(x) (x x)) (\lambda(x) (x x))))$$

Summary:

1. We expect call-by-name to have more computation steps than call-by-value (exception: see above).
2. Upon termination, both produce the same answer.
3. There are cases where call-by-name terminates, but call-by-value does not.
More on Scheme

- list operations:
  
  + list building: ‘( . . . , m . . . )’ — inserts value of m into the list, not the symbol m.

- variable binding operations: define, let, let*

  (let ( (a 2) 
            ((lambda (a b) 
                (b 3)) 
                  (+ a b)) 
            (+ a b))

- (map f ( . . . )): applies function f separately to each element in the list and returns the list of results. Example: (map (lambda (x) (+ x 1)) ‘(0 1 2 3)) evaluates to ‘(1 2 3 4)

- (apply f ( . . . )): applies function f to an argument list and returns the resulting value

- (error . . . ): predefined error routine. Terminates program execution and returns error message with optional values

- When using DrRacket (racket), choose language FrTime in language menu
Closure interpreter \textit{ev} – basic structure

\textit{ev} takes as input an AST of an expression \textit{e} and an environment \textit{env}, and returns the AST of a value.

\begin{verbatim}
(define ev
  (lambda (e env)
    (cond
      ((eq? (car e) '&const) ;; e=(&const c)  ; e=(&const c)
        e)
      ((eq? (car e) '&var) ;; e=(&var v)  ; e=(&var v)
        (lookup env (cadr e)))
      ((eq? (car e) '&lambda) ;; e=(&lambda parms body)  ; e=(&lambda parms body)
        (mk-closure env e))
      ((eq? (car e) '&if) ;; e=(&if a b c)  ; e=(&if a b c)
        (let ((a (cadr e)) ;; e=(&if a b c)
            (b (caddr e))
            (c (cadddr e)))
          (ev (if (equal? (ev a env) '(&const #f)) c b) env)))
      ((eq? (car e) '&apply) ;; e=(&apply f args)  ; e=(&apply f args)
        (let*(((f (cadr e))
            (args (caddr e))
            (fv (ev f env))
            (av (map (lambda (a) (ev a env)) args)))
          (if (and (pair? fv) (eq? (car fv) '&const))
            (delta fv av)
            (apply-cl fv av)))))))
\end{verbatim}
Note:

Just by looking at the function $ev$ we do not know how environments or closures are implemented. $ev$ is said to be *representation independent*. 
TINY interpreter \textit{ev} — closures

\textbf{list (data) representation}

\begin{verbatim}
(define mk-closure ;; returns (&closure env parm-list body)
  (lambda (env v)
    (cond
      ((eq? (car v) '&lambda)
       ('(&closure ,env ,(cadr v) ,(caddr v)))))))

(define apply-cl
  (lambda (vf va)
    (cond
      ((eq? (car vf) '&closure)
       (let ((env (cadr vf)) ;; environment
             (p (caddr vf)) ;; parameter list
             (b (cadddr vf))) ;; body
          (if (= (length p) (length va))
            (ev b (extend* env p va))
            (error 'apply-cl "wrong number of arguments"))))))
\end{verbatim}
procedural (functional) representation

(define extend
  (lambda (env x v)
    (lambda (y)
      (if (eq? x y)
          v
          (lookup env y)))))

(define lookup
  (lambda (env y) (env y)))

(define extend*
  (lambda (env xs vs)
    (if (null? xs)
        env
        (extend* (extend env (car xs) (car vs))
                  (cdr xs)
                  (cdr vs)))))

TINY interpreter $ev$ — environments
delta is needed to apply a functional constant (our +, −, *, / operations) to a value. ev uses the corresponding Scheme operations. If a function symbol such as “+” is encountered, it is looked up in the environment and the list ‘(&const ⟨function+⟩) is returned. Note that an application of such a function requires all arguments to be constants, i.e., of the form ‘(&const ...).

(define delta
  (lambda (f a)
    (let ((R (lambda (s) ‘(&const ,s)))
          (R-1 (lambda (cl)
               (cond
                ((eq? (car cl) ‘&const)
                 (cadr cl))
                (else
                 (error ’delta "non-const args")))))
      (R (apply (R-1 f) (map R-1 a))))))

(define interpret-free-var
  (lambda (x)
    ‘(&const ,(eval x)))

(define empty-env interpret-free-var)
TINY interpreter $ev$ — parser

(define parse
  (lambda (m)
    (cond
      ((number? m) '(&const ,m))
      ((eq? #t m) '(&const #t))
      ((eq? #f m) '(&const #f))
      ((name? m) '(&var ,m))
      ((pair? m)
        (cond
          ((eq? 'if (car m))
            (if (= 4 (length m))
              '(&if ,(parse (cadr m))
                 ,(parse (caddr m)) ,(parse (cadddr m)))
              (error 'parse "Syntax error")))
          ((eq? 'lambda (car m))
            (if (and (= 3 (length m))
                  (list? (cadr m))
                  (andmap name? (cadr m)))
              '(&lambda ,(cadr m) ,(parse (caddr m)))
              (error 'parse "Syntax error")))
          (else
            '(&apply ,(parse (car m)) ,(parse* (cdr m))))
          (else (error 'parse "Syntax error")))
      )))
  (define parse* (lambda (m) (map parse m)))
(define name?
  (lambda (s)
    (and (symbol? s) (not (memq s '(if lambda))))))

(define andmap
  (lambda (f l)
    (if (null? l)
        #t
        (and (f (car l)) (andmap f (cdr l))))))

(define unparse
  (lambda (a)
    (cond
      ((eq? (car a) '&const) (cadr a))
      ((eq? (car a) '&var) (cadr a))
      ((eq? (car a) '&if)
        'if ,(unparse (cadr a)) ,(unparse (caddr a))
        ,(unparse (cadddr a))))
      ((eq? (car a) '&lambda)
        'lambda ,(cadr a) ,(unparse (caddr a)))
      ((eq? (car a) '&apply)
        (cons (unparse (cadr a)) (map unparse (caddr a))))
      ((eq? (car a) '&closure)
        'lambda ,(caddr a) ,(unparse (cadddr a)))
      (else (error 'unparse "unexpected syntax tree" a))))
Call-by-value closure interpreter \textit{evaluate}

That’s it! We are now ready to put everything together.

\begin{verbatim}
(define evaluate
  (lambda (m)
    (unparse (ev (parse m) empty-env))))
\end{verbatim}

Questions

- What parameter passing style does our interpreter use?
- What is the order of evaluation of the actual parameters for a function application?
Call-by-name closure interpreter

How do we have to modify our interpreter to implement call–by–name instead of call–by–value?

**Key idea:**
If the argument of an application is not a value, we can postpone its evaluation by wrapping it into a closure that treats the argument expression as a lambda abstraction without parameters. Such a “special” closure is called a **thunk**. If we need the “real” value of an argument expression, we just evaluate the thunk in its environment.

Note that now our environment can contain three types of values, namely **constants**, **closures**, and **thunks**.
Dynamically-scoped interpreter

How do we have to modify our interpreter to implement call-by-value with dynamic scoping?

**Key idea:**
The difference between static and dynamic scoping in TINY is with respect to the rules how closures are applied.

- **static** ⇒ use environment within the closure
- **dynamic** ⇒ ignore environment within the closure and use current environment at the application.

See example interpreters on **ilab**:

- **static**:  
  \(~uli/cs515/examples/scheme/DefiningInterpreters/ValueStatic.ss\)

- **dynamic**:  
  \(~uli/cs515/examples/scheme/DefiningInterpreters/ValueDynamic.ss\)

- **static-name**:  
  \(~uli/cs515/examples/scheme/DefiningInterpreters/NameStatic.ss\)
Closure interpreters — Remember the Y

(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))

This does not work for call-by-value interpreters!

**Trick:** – Enclose the self application (x x) within a function (lambda(y) ((x x) y))

(lambda (f)
  ((lambda (x) (lambda (y) ((f (x x)) y)))
   (lambda (x) (lambda (y) ((f (x x)) y)))))

Let’s try an example using our call-by-value interpreter:

> (define test-fac3 '(((lambda (f)
    ((lambda (x) (lambda (y) ((f (x x)) y)))
     (lambda (x) (lambda (y) ((f (x x)) y)))))
    (lambda (fac)
     (lambda (x)
      (if (equal? x 1) 1 (* x (fac (- x 1))))) 3))
> (evaluate test-fac3)
> 6

**NOTE:** Check out
~uli/cs515/examples/scheme/DefiningInterpreters/YY.ss on the ilab cluster.
Closure interpreters

- Interpreters are important tools to understand and specify the semantics of programming languages.

- Closures are an important concept for any language with functions as first order objects and static scoping. It gives us an idea what “price” we have to pay if we want first–order functions or a particular parameter passing style.

- So far, we only talked about “pure” functional languages without a store and assignment (no side effects). How to extend our interpreters to deal with a store?

  - Introduce a store as a function from addresses to values: \( \text{store} = \text{Memory Locations} \rightarrow \text{Values} \)
  
  - Modify environments to map names to memory locations: \( \rho \in \text{Env} = \text{Variables} \rightarrow \text{Memory Locations} \)
  
  - Modify \( \text{ev} \) to take one more argument:
    
    \[
    \text{(define ev (lambda (e env store) \ldots)).}
    \]
    
    In addition, \( \text{ev} \) will return a value and a store.

  - Of course, there is a lot more work to be done here, but I hope you get the idea.
Next topic: Typing and Type Systems

Read Scott: Chapter 7.1 - 7.2; ALSU Chapter 6.5

type:
A set of values and meaningful operations on them.

Types provide semantic “sanity checks” (consistency checks). Types help identify:

- errors, if an operator is applied to an incompatible operand.
  - dereferencing of a non-pointer
  - adding a function name to something
  - incorrect number of parameters to a procedure
  - ...

- which operation to use for overloaded names and operators, or what type coercion to use (e.g.: 3.0 + 1)

- identification of polymorphic functions. Such functions can be executed with arguments of several types (e.g.: list of objects of unknown type α)
Type systems

Each language construct (operator, expression, statement, ...) has a type

**basic types:** integer, real, character, ...

**constructed types:** arrays, records, sets, pointers, functions

A **type system** is a collection of *rules* for assigning *type expressions* to operators, expressions, ... in the program. Type systems are language dependent.

A **type checker** implements the type system, i.e., deduces type expressions for program constructs based on the type inference rules of the type system. The type checker “computes” or “reconstructs” type expressions.
Example type rules

- If both operands of the arithmetic operators of addition, subtraction, and multiplication are of type integer, then the result is of type integer (Pascal definition).

Rule for + (analogue rules for − and *):

\[
\frac{E \vdash e_1 : integer \quad E \vdash e_2 : integer}{E \vdash (e_1 + e_2) : integer}
\]

where \( E \) is a type environment that maps constants and variables to their types.

In combination with the following axiom template in the type system \( \{ c : \alpha \} \vdash c : \alpha \)
we can now infer that \((2 + 3)\) is of type integer:

\[
\frac{E \vdash 2 : integer \quad E \vdash 3 : integer}{E \vdash (2 + 3) : integer}
\]

where \( E = \{2 : integer, 3 : integer\}\).

In general, type deduction proofs work bottom up.
Example type rules

- The result of the unary & operator is a pointer to the object referred to by the operand. If the operand is of type "foo", then the type of the result is a "pointer to foo". (C and C++ definition)

\[
E \vdash e : \alpha \\
\hline
E \vdash \&e : \text{pointer}(\alpha)
\]

- Two expressions can only be compared if they have the same types. The result is of type boolean.

\[
E \vdash e_1 : \alpha \quad E \vdash e_2 : \alpha \\
\hline
E \vdash (e_1 = e_2) : \text{boolean}
\]

In the examples, integer and \(\alpha\) are type expressions.
Next class

• More on type checking
• Overview of compiler frontends
• LL(1) and LR(1) parsing strategies