Announcements

- Test on Wednesday starting at 8PM. Room assignments on Piazza.
- No lecture on Wednesday.
- No discussion section this week.
- There are labs this week (and office hours).
Lecture 28: Interpretating Scheme

A Scheme is essentially an extension of the calculator:

- A component known as the reader reads Scheme values (atoms and pairs).
- Since Scheme expressions and programs are a subset of Scheme values, no further parsing is necessary.
- A function eval evaluates Scheme expressions.
  - Atoms are its base cases.
  - For function calls, it uses a function apply, as for the calculator.
Apply

- The interpreter function \texttt{apply(func, args)} has the effect of allowing one to construct and evaluate function calls.

- **Aside:** In Python, we've been writing \texttt{func(*args)} to get the effect of \texttt{apply(func,args)} in ordinary programs.

- **Aside:** it is made available to Scheme programmers as the built-in function \texttt{apply}:

  (define L '(1 2 3))
  (apply + L) ===> (+ 1 2 3) ===> 6

- The \texttt{apply} function itself has two cases:
  - Either \texttt{func} is a primitive, built-in function, in which case, its code is part of the interpreter, or
  - \texttt{func} is a user-defined function, in which case its code is stored in it as a Scheme expression, and is evaluated by \texttt{eval}.

- So there is a “recursive dance” back and forth between \texttt{eval}, and \texttt{apply}.
Evaluation for Scheme

• Simple expressions are evaluated as for the calculator.
• A Scheme expression consisting of a number simply evaluates to that number.
• A function call \((E_0\ E_1\ \cdots\ E_n)\) is evaluated by recursively evaluating the \(E_i\) and then using \texttt{apply}.
• But Scheme has a number of other cases to handle.
• \textbf{Aside:} As for \texttt{apply}, the evaluation function for Scheme is also available to Scheme programmers, in the form of a function \texttt{eval}.
• E.g., \((\texttt{eval (list + 1 2)})\) and \((\texttt{eval '(+ 1 2)})\) produce 3.
Evaluation of Symbols

- In Scheme expressions, most symbols represent identifiers, which we did not encounter in the calculator.

- Obviously, we need more information to evaluate a symbol than just the symbol itself.

- Fortunately, we’ve already seen exactly what is needed: an environment.

- Thus, to evaluate a Scheme expression, we will need both the expression itself and the environment in which to evaluate it.

- As it happens, exactly the same kind of structure as in Python—environment frames linked by parent pointers—is what we need to interpret Scheme.

- This is because Scheme uses nearly the same scope rules as Python does.

- Earlier dialects of Lisp, however, used a different kind of scope rule.
Static and Dynamic Scoping

- The **scope rules** of a language are the rules governing what names (identifiers) mean at each point in a program.

- We call the scope rules of Scheme (and Python)—those that are described by environment diagrams as we’ve been using them—**static** or **lexical** scoping.

- But in original Lisp, scoping was **dynamic**.

- Example (using classic Lisp notation):

  ```lisp
  (defun f (x) ;; Like (define (f x) ...) in Scheme
      (g))
  (defun g ()
      (* x 2))
  (setq x 3) ;; Like set! and also defines x at outer level.
  (g) ;; ==> 6
  (f 2) ;; ==> 4
  (g) ;; ==> 6
  ```

- That is, the meaning of \(x\) depends on the most recent and still active definition of \(x\), even where the reference to \(x\) is not nested inside the defining function.
Eval and Scoping

- Dynamic scoping made `eval` easy to define: interpret any variables according to their “current binding.”

- But `eval` in pure Scheme behaves like normal functions; it would not have access to the current binding at the place it is called.

- To make it definable (without tricks) in Scheme, one must add a parameter to `eval` to convey the desired environment.

- In the fifth revision of Scheme, one had the choice of indicating an empty environment and the standard, builtin environment.

- Our STk interpreter goes its own way:
  - `(eval E)` evaluates in the global environment.
  - `(eval E (the-environment))` evaluates in the current environment.
  - `(eval E (procedure-environment f))` evaluates in the environment pointed to by function `f`: what we’ve been calling the `parent pointer` of a function value.
Remaining Cases

- We've dealt with function calls, numbers, and symbols.
- This leaves only the *special forms*.
- All special forms lists indicated by their first symbols:

\[
\text{(quote } \text{EXPR}) \quad ; \text{ Easy: return } \text{EXPR} \text{ unchanged}
\]

\[
\text{(lambda } (\text{ARGS} \text{) EXPR)}
\]

\[
\text{(define } \text{ID EXPR)}
\]

\[
\text{(define } (\text{ID ARGS}) \text{ EXPR)}
\quad ; \text{ Same as (define } \text{ID (lambda } (\text{ARGS} \text{) EXPR)})
\]

\[
\text{(if } \text{EXPR EXPR-IF-TRUE EXPR-IF-FALSE)}
\]

\[
\text{(begin } \text{EXPR}_1 \ldots \text{EXPR}_n) \quad ; \text{ Evaluate EXPR}_i, \text{ return last}
\]

\[
\text{(cond } (\text{COND-EXPR}_1 \text{ VAL-EXPR}_1)
\quad (\text{COND-EXPR}_2 \text{ VAL-EXPR}_2) \ldots)
\]

\[
\text{(and } \text{EXPR}_1 \text{EXPR}_2 \ldots)
\]

\[
\text{(or } \text{EXPR}_1 \text{EXPR}_2 \ldots)
\]
Lambda and Functions

- In the interpreter, evaluating the lambda special form returns a value of some type for representing functions.

- Its content is dictated by what `apply` will need:

  (lambda (ARGS) EXPR)

  - The list ARG.
  - The body EXPR.
  - The parent environment: The environment in which it is evaluated.
Other Special Forms

• Handling the other special forms is pretty straightforward:

• The \texttt{if} form is typical: to evaluate
  
  \texttt{(if EXPR EXPR-IF-TRUE EXPR-IF-FALSE)}

  - Evaluate \texttt{EXPR}.
  - If returned value is false (\#f), evaluate \texttt{EXPR-IF-FALSE} and return its value.
  - Otherwise, evaluate \texttt{EXPR-IF-TRUE} and return its value.