EECS 390 – Lecture 3

Scheme

1/22/24

Expressions

- An ***expression*** is a syntactic construct that is *evaluated* to produce a value
  - Examples: `3 + 4`, `foo()`
- Literals are one of the simplest kinds of expressions
  - Evaluate to the value they represent
- An identifier can syntactically be an expression
  - But only semantically valid if it names a first-class entity
  - Evaluates to the entity it names

Compound Expressions

- ***Precedence*** and ***associativity*** rules determine how subexpressions are grouped when multiple operators are involved

- Precedence: divides operators into priority groups
  - Example: `{*,/,%} > {+,-}`

- Associativity: how operators in the _same_ precedence group apply
  - Example: `x = y = 3 + 4 - - - 5`

- Order of evaluation is distinct from precedence and associativity
  - Can be specified (mostly left to right in Python, Java), unspecified (function arguments in Scheme), or partially specified (C++)
  - Example: `cout << ++x << x;`

Statements and Side Effects

- Imperative languages have ***statements***, which are ***executed*** to carry out some action
- Generally have ***side effects***, which change the state of the machine
- Language syntax determines what constitutes a statement and how it is terminated
  - C family: simple statements terminated by semicolon
  - Python: newline (usually) or semicolon (rare)
  - Scheme?

Declarations and Definitions

- A ***declaration*** introduces a name into a program, along with properties about what it names
  - Examples
    ```
    extern int x;
    void foo(int, int);
    class SomeClass;
    ```

- A ***definition*** additionally specifies the actual data or code that the name refers to
  - C, C++: definitions are declarations, but a declaration need not be a definition
  - Java: no distinction between definitions and declarations
  - Python: no declarations[1], definitions are statements that are executed

[1] Type annotations are not considered declarations. Quoting from PEP 526: "Type annotations should not be confused with variable declarations in statically typed languages."

Agenda

- Scheme

Running Scheme

- We recommend Racket
  - https://download.racket-lang.org/
  - Includes DrRacket IDE and command-line `plt-r5rs` interpreter
- Online interpreter for simple examples
  - https://repl.it/languages/scheme
- Be aware that most interpreters are not fully R5RS compliant, so we recommend sticking to Racket for homework/project development

On MacOS, Racket can be installed with Homebrew:

```
$ brew install --cask racket
```

Call Expressions

- Everything is an expression in Scheme
- Simple expressions: literals, names
- Compound expressions consist of a parenthesized list
- Call expressions:

```
(function arg1 arg2 ... argN)
```

- Examples:

```
(+ 3 4)
(+ (* 3 5) (- 10 6))
(quotient 10 3)
```

Integer division

Conditionals

- Special forms have their own evaluation rules
- Conditional evaluates test, then evaluates *then* expression if true, otherwise the *else* expression if provided

```
(if <test> <then_expr> <else_expr>)
```

- Value of whole expression is value of then or else expression
  - If test is false and no else expression, then value is unspecified
- Only `#f` is a false value, all other values are true

Definitions and Blocks

- Variables can be defined in the current frame using `define`

```
(define <name> <expr>)
```

- In standard Scheme, this can only be at the top level or at the beginning of a block
  - We will only use it at the top level in code we write

- Blocks can be introduced with `let`

```
(let ((<name1> <expr1>) ... (<nameN> <exprN>))
   <body_expr1> <body_expr2> ... <body_exprN>
)
```

`let` can be considered syntactic sugar for `lambda` definition and application

Functions

- Functions can also be defined using `define`

```
(define (<name> <param1> ... <paramN>)
   <body_expr1> ... <body_exprN>
)
```

- Anonymous functions can be defined using `lambda`

```
(lambda (<param1> ... <paramN>)
   <body_expr1> ... <body_exprN>
)
```

- Then the `define` form is equivalent to

```
(define <name>
   (lambda (<param1> ... <paramN>)
      <body_expr1> ... <body_exprN>
   )
)
```

Exercise: Functions

- Consider the following code. What is the result of the call `(fibonacci 5)`?

```
(define (fibonacci x)
  (if (= x 0)
      0
  )
  (if (= x 1)
      1
  )
  (+ (fibonacci (- x 1))
     (fibonacci (- x 2))
  )
)
```

Poll: What is the result of `(fibonacci 5)`?

A) 0

B) 5

C) Some other value

D) An error

Pairs

- Pairs are a fundamental mechanism for combining data

- Construct pair using `cons`

```
> (define x (cons 1 2))
> x
(1 . 2)
```

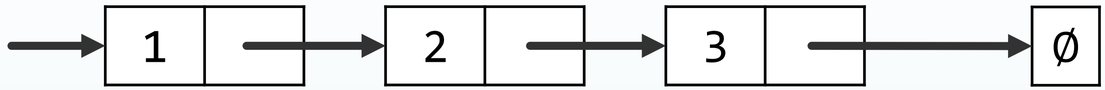
Dot denotes pair where the second is not a list

- Access the first and second with `car` and `cdr`

```
> (car x)
1
> (cdr x)
2
```

Lists

- A list is a sequence of pairs terminated by an empty list



- An empty list is denoted by `'()`

```
> (define y (cons 1 (cons 2 (cons 3 '()))))
> y
(1 2 3)
> (define y (list 1 2 3))
> y
(1 2 3)
> (car y)
1
> (cdr y)
(2 3)
> (cdr (cdr (cdr y)))
()
```

Also (cdddr y) in standard Scheme

Symbolic Data

- In Scheme, both code and data share the same representation

- Quotation specifies that what follows should be treated as data and not evaluated

```
> (define x 3)
> x
3
> 'x
x
> '(hello world)
(hello world)
```

Equivalent to `(quote x)`

The `eval` procedure can be used to evaluate symbolic data