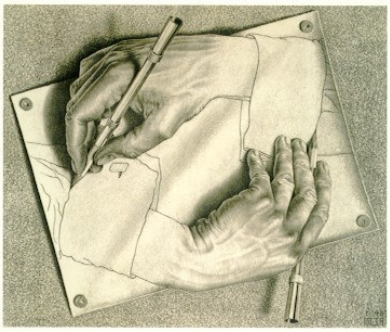
Lecture 3: Rules of Evaluation



CS150: Computer Science
University of Virginia
Computer Science

David Evans
http://www.cs.virginia.edu/evans

Menu

- Language Elements
- Why don't we just program computers using English?
- Evaluation
- Procedures

Are there any non-recursive natural languages? What would happen to a society that spoke one?

Not for humans at least.
They would run out of original things to say.

Chimps and Dolphins are able to learn non-recursive "languages" (some linguists argue they are not really "languages"), but **only humans can learn recursive languages**.

Running out of Ideas

"Its all been said before."

Eventually true for a non-recursive language.

Never true for a recursive language.
There is always something original left to say!

Language Elements

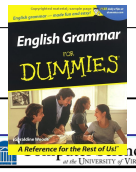
When learning a foreign language, which elements are hardest to learn?

- Primitives: lots of them, and hard to learn real *meaning*
- Means of Combination
  - Complex, but, all natural languages have similar ones [Chomsky]
    - SOV (45% of all languages) *Sentence* ::= *Subject Object Verb* (Korean)
    - SVO (42%) *Sentence* ::= *Subject Verb Object*
    - VSO (9%) *Sentence* ::= *Verb Subject Object* (Welsh)
      "Lladdodd y ddraig y dyn." (Killed the dragon the man.)
    - OSV (<1%): Tobati (New Guinea)
    - Schemish: *Expression* ::= (*Verb Object*)
- Means of Abstraction: few of these, but tricky to learn differences across languages
  - English: I, we
  - Tok Pisin (Papua New Guinea): mi (I), mitupela (he/she and I), mitripela (both of them and I), mipela (all of them and I), yumitupela (you and I), yumitripela (both of you and I), yumipela (all of you and I)

	Pages in *Revised*^{5} *Report on the Algorithmic Language Scheme*		Pages in C++ Language Specification (1998)	
Primitives	Standard Procedures	18	Standard Procedures	356
	Primitive expressions	2	Primitive expressions	30
	Identifiers, numerals	1	Identifiers, numerals	10
Means of Combination	Expressions	2	Expressions, Statements	197
	Program structure	2	Program Structure	35
Means of Abstraction	Definitions	½	Declarations, Classes	173
	48 pages total (includes formal specification and examples)		**776** pages total (includes no formal specification or examples)	

C++ Core language issues list has 529 items!

	Pages in *Revised*^{5} *Report on the Algorithmic Language Scheme*		English	
Primitives	Standard Procedures Primitive expressions Identifiers, numerals	18 2 1	Morphemes Words in Oxford English Dictionary	? 500,000
Means of Combination	Expressions Program structure	2 2	Grammar Rules *English Grammar for Dummies* Book	100s (?) 384 pages
Means of Abstraction	Definitions	½	Pronouns	~20
	48 pages total (includes formal specification and examples)			



Why don't we just program computers using English?

- Too hard and complex

Non-native English speakers don't need convincing. The rest of you have spent your whole life learning English (and first 5 years of your life doing little else) and still don't know useful words like floccipoccinihilipilification! There are thoughts that even native speakers find it hard to express.

By the end of today you will know enough Scheme (nearly the entire language) to express and understand ***every*** computation. By PS7, you will know enough to completely and precisely describe Scheme in terms of itself (try doing that in English!)

Why don't we just program computers using English?

- Not concise enough

English:
To find the maximum of two numbers, compare them. If the first number is greater than the second number, the maximum is the first number. Otherwise, the maximum is the second number.

Scheme:
(define (max a b) (if (> a b) a b))

Why don't we just program computers using English?

- Limited means of abstraction

There are only a few pronouns: he, she, it, they, these, … (English doesn't even have a gender-neutral pronoun for a person!)

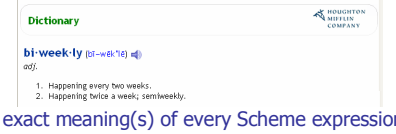
Only Webster and Oxford can make up new ones.

define allows any programmer to make up as many pronouns as she wants, and use them to represent anything.

Why don't we just program computers using English?

- Mapping between surface forms and meanings are ambiguous and imprecise

Would you rather be paid biweekly or every week?


Dictionary
HOUGHTON MIFFLIN COMPANY
bi·week·ly (bī-wēk'lē)
adj.
1. Happening every two weeks.
2. Happening twice a week; semiweekly.

The exact meaning(s) of every Scheme expression is determined by simple, unambiguous rules we will learn today (and refine later in the course).

Essential Scheme

Expression ::= **(***Expression*$_1$ *Expression**)

Expression ::= **(if** *Expression*$_1$ *Expression*$_2$ *Expression*$_3$**)**

Expression ::= **(define** *name Expression***)**

Expression ::= *Primitive*

Primitive ::= *number*

Primitive ::= **+** | **-** | ***** | …

Primitive ::= …

Grammar is clear, just follow the replacement rules. But what does it all mean?

Evaluation

Expressions and Values

- (Almost) every *expression* has a *value*
  - Have you seen any expressions that don't have values?
- When an expression with a value is *evaluated*, its value is produced

Evaluation Rule 1: Primitives

If the expression is a *primitive*, it is self-evaluating.

```
> 2
2
> #t
#t
> +
#<primitive:+>
```


Evaluation Rule 2: Names

If the expression is a *name*, it evaluates to the value associated with that name.

```
> (define two 2)
> two
2
```


Evaluation Rule 3: Application

3. If the expression is an application:
   a) Evaluate all the subexpressions of the combination (in any order)
   b) Apply the value of the first subexpression to the values of all the other subexpressions.

(expression_0 expression_1 expression_2 … **)**

Rules for Application

1. If the procedure to apply is a *primitive*, just do it.
2. If the procedure is a *compound procedure*, **evaluate** the body of the procedure with each formal parameter replaced by the corresponding actual argument expression value.

Making Procedures

- **lambda** means "make a procedure"

Expression ::=
(lambda (*Parameters***)** *Expression***)**
Parameters ::=
Parameters ::= *Name Parameters*

Lambda Example: Tautology Function

```
(lambda      make a procedure
 ()          with no parameters
 #t)         with body #t

> ((lambda () #t) 150)
#<procedure>: expects no arguments, given 1: 150
> ((lambda () #t))
#t
> ((lambda (x) x) 150)
150
```


You've Already Used Lambda!

```
(define (closer-color?
          sample color1 color2)
  Expr)
```

is a shortcut for:

```
(define closer-color?
  (lambda (sample color1 color2)
    Expr))
```


Eval and Apply are defined in terms of each other.

Without Eval, there would be no Apply, Without Apply there would be no Eval!

All of Scheme

- Once you understand Eval and Apply, you can understand all Scheme programs!
- Except:
  - We have special Eval rules for special forms (like **if**)

  Evaluation Rule 4: If it is a *special form*, do something special.

Evaluating Special Forms

- **Eval 4-if.** If the expression is
  (if $Expression_0$ $Expression_1$ $Expression_2$**)**
  evaluate $Expression_0$. If it evaluates to #f, the value of the if expression is the value of $Expression_2$. Otherwise, the value of the if expression is the value of $Expression_1$.
- **Eval 4-lambda.** Lambda expressions self-evaluate. (Do not do anything until it is applied.)

More Special Forms

- **Eval 4-define.** If the expression is
  (define *Name Expression***)**
  associate the *Expression* with *Name*.
- **Eval 4-begin.** If the expression is
  (begin $Expression_0$ $Expression_1$ … $Expression_k$**)**
  evaluate all the sub-expressions. The value of the begin expression is the value of $Expression_k$.

Scheme

Expression ::= *Primitive*

Eval 1: If the expression is a *primitive*, it is self-evaluating.

Expression ::= *Name*

Eval 2: If the expression is a *name*, it evaluates to the value associated with that name.

Expression ::= **(***Expression ExpressionList***)**

Eval 3: If the expression is an application:

(a) Evaluate all the subexpressions (in any order)

(b) Apply the value of the first subexpression to the values of all the other subexpressions.

ExpressionList ::=

ExpressionList ::= *Expression ExpressionList*

Special Forms

Expression ::= **(lambda (***Parameters***)** *Expression***)**

Eval 4-lambda. Lambda expressions self-evaluate.

Parameters ::=

Parameters ::= *Name Parameters*

Expression ::= **(define** *Name Expression***)**

Eval 4-define. If the expression is **(define** *Name Expression***)** associate the *Expression* with *Name*.

Expression ::= **(if** $Expression_0$ $Expression_1$ $Expression_2$**)**

Eval 4-if. Evaluate $Expression_0$. If it evaluates to #f, the value of the if expression is the value of $Expression_2$. Otherwise, the value of the if expression is the value of $Expression_1$.

Expression ::= **(begin** *ExpressionList Expression***)**

Eval 4-begin. Evaluate all the sub-expressions. The value of the begin expression is the value of *Expression*.

Now, you know all of Scheme!

(Except for 3 more special forms you will learn for PS5.)

Charge

- PS1 Due Wednesday
  - Staffed Lab hours:
    today, 3:30-5pm
    Tuesday, 4-5:30pm
- Reading for Friday: SICP, 1.2
- Reading for Monday: GEB, Ch 5