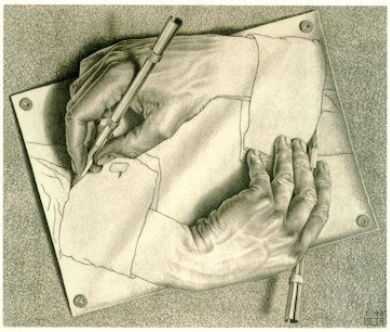
Lecture 3: Rules of Evaluation



CS150: Computer Science
University of Virginia
Computer Science

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

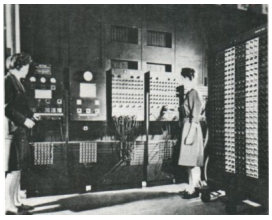
Menu

- Describing Languages
- Learning New Languages
- Evaluation Rules

My office hours are now scheduled:
Wednesdays, 1-2pm (right after class)
Thursdays, 3:30-4:30pm
If you can't make them, send email to arrange meetings at other times.

ENIAC: Electronic Numerical Integrator and Computer

- Early WWII computer
  - But **not** the world's first (PS4)
- Built to calculate bombing tables



Memory size:
twenty 10 decimal digit accumulators = 664 bits

ENIAC (1946): ½ mm
Apollo Guidance Computer (1969): 1 inch
You: 4.4 miles

Directions for Getting **6**

1. Choose any regular accumulator (ie. Accumulator #9).
2. Direct the Initiating Pulse to terminal *5i*.
3. The initiating pulse is produced by the initiating unit's *Io* terminal each time the Eniac is started. This terminal is usually, by default, plugged into Program Line 1-1 (described later). Simply connect a program cable from Program Line 1-1 to terminal *5i* on this Accumulator.
4. Set the Repeat Switch for Program Control 5 to 6.
5. Set the Operation Switch for Program Control 5 to .
6. Set the Clear-Correct switch to C.
7. Turn on and clear the Eniac.
8. Normally, when the Eniac is first started, a clearing process is begun. If the Eniac had been previously started, or if there are random neons illuminated in the accumulators, the ``Initial Clear" button of the Initiating device can be pressed.
9. Press the ``Initiating Pulse Switch" that is located on the Initiating device.
10. **Stand back.**

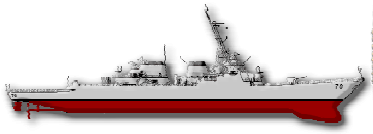
Admiral Grace Hopper

(1906-1992)

"Nobody believed that I had a running compiler and nobody would touch it. They told me computers could only do arithmetic."

- Mathematics PhD Yale, 1934
- Entered Navy, 1943
- First to program Mark I (first "large" computer, 51 feet long)
- Wrote first compiler (1952) – program for programming computers
- Co-designer of COBOL (most widely used programming language until a few years ago)

USS Hopper



"Dare and Do"

Guest on David Letterman

Nanostick

- How far does light travel in 1 nanosecond?

```
> (define nanosecond (/ 1 (* 1000 1000 1000))) ;; 1 billionth of a s
> (define lightspeed 299792458) ; m / s
> (* lightspeed nanosecond)
149896229/500000000
> (exact->inexact (* lightspeed nanosecond))
0.299792458
```

= just under 1 foot

Dell machines in Small Hall have "1.8-GHz Pentium 4 CPU"

GHz = GigaHertz = 1 Billion times per second
They must finish a step before light travels 6.6 inches!

Code written by humans

⇩

Compiler

⇩

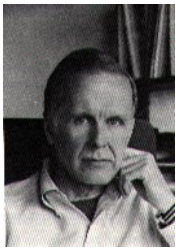
Code machine can run

Compiler translates from code in a high-level language to machine code

DrScheme uses an *interpreter*. An interpreter is like a compiler, except it runs quickly and quietly on small bits of code at a time.

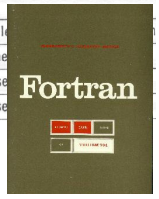
John Backus

- Chemistry major at UVA (entered 1943)
- Flunked out after second semester
- Joined IBM as programmer in 1950
- Developed Fortran, first commercially successful programming language and compiler



IBM 704 Fortran manual, 1956

STATEMENT	NORMAL SEQUENCING
a = b	Next executable statement
GO TO n	Statement n
GO TO n, ($n_1,n_2,\ldots,n_m$)	Statement last assigned
ASSIGN i TO n	Next executable statement
GO TO ($n_1,n_2,\ldots,n_m$), i	Statement n_i
IF (a) n_1,n_2,n_3	Statement n_1,n_2,n_3 as a l
SENSE LIGHT i	Next executable stateme
IF (SENSE LIGHT i) n_1,n_2	Statement n_1,n_2 as Sense
IF (SENSE SWITCH i) n_1,n_2	" " as Sense



Describing Languages

- Fortran language was described using English
  - Imprecise
  - Verbose, lots to read
  - Ad hoc

```
DO 10 I=1.10
```

Assigns `1.10` to the variable `DO10I`

```
DO 10 I=1,10
```

Loops for `I = 1` to `10`

(Often incorrectly blamed for loss of Mariner-I)

- Wanted a more precise way of describing a language

Backus Naur Form

symbol ::= *replacement*

We can replace *symbol* with *replacement*

A ::= *B* means anywhere you have an *A*, you can replace it with a *B*.

nonterminal – symbol that appears on left side of rule

terminals – symbol that **never** appears on the left side of a rule

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*
Primitives	
Means of Combination	
Means of Abstraction	
	48 pages total (includes formal specification and examples)

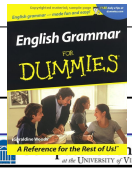
	Pages in *Revised[5] Report on the Algorithmic Language Scheme*	
Primitives	Standard Procedures Primitive expressions Identifiers, numerals	18 2 1
Means of Combination	Expressions Program structure	2 2
Means of Abstraction	Definitions	½
	48 pages total (includes formal specification and examples)	

	Pages in *Revised5 Report on the Algorithmic Language Scheme*		Pages in C++ Language Specification (1998)
Primitives	Standard Procedures Primitive expressions Identifiers, numerals	18 2 1	
Means of Combination	Expressions Program structure	2 2	
Means of Abstraction	Definitions	½	
	48 pages total (includes formal specification and examples)		

	Pages in *Revised[5] Report on the Algorithmic Language Scheme*		Pages in C++ Language Specification (1998)	
Primitives	Standard Procedures Primitive expressions Identifiers, numerals	18 2 1	Standard Procedures Primitive expressions Identifiers, numerals	356 30 10
Means of Combination	Expressions Program structure	2 2	Expressions, Statements Program Structure	197 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 469 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)			



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

Primitive Expressions

Expression ::= *PrimitiveExpression*

PrimitiveExpression ::= ***Number***
PrimitiveExpression ::= **#t** | **#f**
PrimitiveExpression ::= ***Primitive Procedure***

Evaluation Rule 1: Primitives

If the expression is a *primitive*, it evaluates to its pre-defined value.

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


Name Expressions

Expression ::= *NameExpression*

NameExpression ::= ***Name***

Evaluation Rule 2: Names

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

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

Application Expressions

Expression ::= *Application Expression*
ApplicationExpression
::= **(***Expression MoreExpressions***)**

MoreExpressions ::= ε
MoreExpressions
::= *Expression MoreExpressions*

Evaluation Rule 3: Application

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.

(Expression_0 Expression_1 Expression_2 … **)**

Rules for Application

1. **Primitives.** If the procedure to apply is a *primitive*, just do it.

2. **Constructed Procedures.** If the procedure is a *constructed procedure*, **evaluate** the body of the procedure with each formal parameter replaced by the corresponding actual argument expression value.

Eval and Apply are defined in terms of each other.

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

Making Procedures

lambda means "make a procedure"

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

Evaluation Rule 4: Lambda

4. Lambda expressions evaluate to a procedure that takes the given parameters and has the expression as its body.

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
```


Evaluation Rule 5: If

(if *Expression*$_{\text{Predicate}}$
Expression$_{\text{Consequent}}$
Expression$_{\text{Alternate}}$**)**

To evaluate an if expression:

(a) Evaluate *Expression*$_{\text{Predicate}}$.

(b) If it evaluates to #f, the value of the if expression is the value of *Expression*$_{\text{Alternate}}$. Otherwise, the value of the if expression is the value of *Expression*$_{\text{Consequent}}$.

Now You Know All of Scheme!

- Once you understand Eval and Apply, you can understand all Scheme programs!
- Except:
  - There are a few more special forms (like **if**)
  - We have not define the evaluation rules precisely enough to unambiguously understand all programs (e.g., what does "value associated with a name" mean?)

Charge

- Problem Set 2: out today (we'll talk about it next class)
- Reading: Chapters 4 and 5