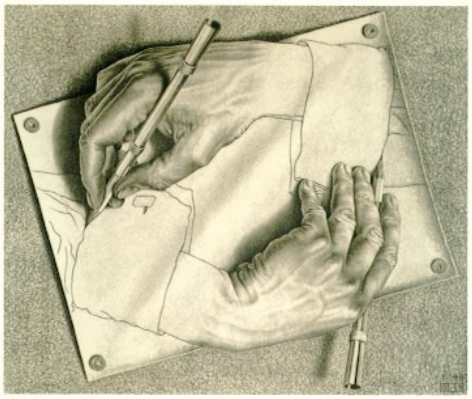


Class 3: Rules of Evaluation

David Evans
cs1120 Fall 2011

Menu

Describing Languages

Questions from Notes

- Computing photomosaics, non-recursive languages, hardest language elements to learn

Scheme: Grammar and Rules of Evaluation

Code written by humans

↓

Compiler/Interpreter

↓

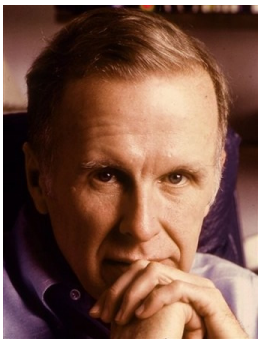
Code machine can run

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

Scheme 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

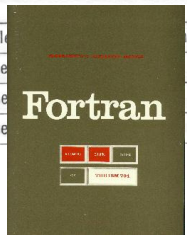
- Chemistry major at UVA (entered 1943)
- Flunked out after second semester
- Joined IBM as programmer in 1950
- Developed **Fortran**, first commercially successful programming language and compiler



John Backus, 1924 – 2007

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,...,n_m)	Statement last assigned
ASSIGN i TO n	Next executable statement
GO TO (n_1,n_2,...,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 statem
IF (SENSE LIGHT i) n_1,n_2	Statement n_1,n_2 as Sens
IF (SENSE SWITCH i) n_1,n_2	" " " as Sens



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*)

Backus wanted a **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

Note: this is named for John Backus for being the first person to advocate using it for describing programming languages, but linguists were using similar techniques much earlier.

Recap: Zero, One, Infinity

word ::= **anti-***word*

This rule can make **0** words.

word ::= **hippopotomonstrosesquipedaliophobia**

This rule can make **1** word.

word ::= **anti-***word*

word ::= **hippopotomonstrosesquipedaliophobia**

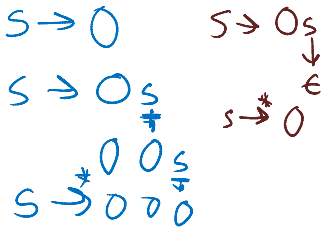
These two rules can make **infinitely** many words, enough to express all ideas in the universe!

Simple BNF Grammar

S ::= **0***S*

~~*S* ::= **0**~~

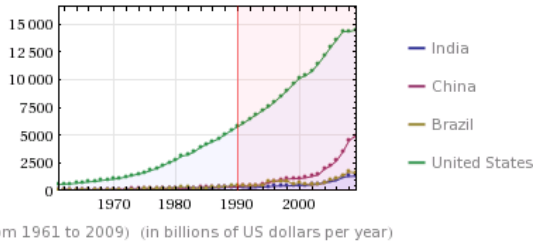
S ::= ϵ



Rapid Exponential Growth

Question from Class 1: *What other things have changed as much as (or more that!) computing power in your lifetime?*

Communication (global IP traffic, PB/month) (Alex)
Energy Consumption (world energy use, TW) (Filip)
Human (number of cells) (Ouamdwipwaw)
Knowledge (Tyson's measure) (Deirdre Regan)
National Debt (Michael)
TV (number of pixels) (gtc5sn)
Wealth (world GDP) (Chris Smith)



(from 1961 to 2009) (in billions of US dollars per year)

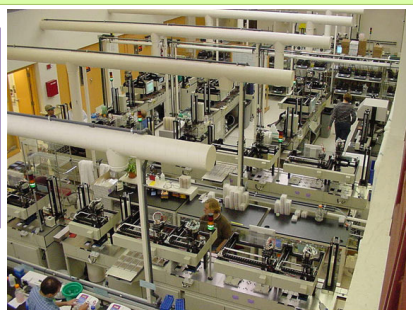
http://www.wolframalpha.com/input/?i=india+china+brazil+usa+gdp+1990-2011

	1991 value	2011 value	Growth	Doubling time
Communication (global IP traffic, PB/month) (Alex)				38 years
Energy Consumption (world energy use, TW) (Filip)				12 years
Human (number of cells) (Ouamdwipwaw)				15 years
Knowledge (Tyson's measure) (Deirdre Regan)				9 years
National Debt (Michael)				1.73 years
TV (number of pixels) (gtc5sn)				9.9 months
Wealth (world GDP) (Chris Smith)				5.3 months
Computing Power/Dollar (Moore's Law)				**18 months**

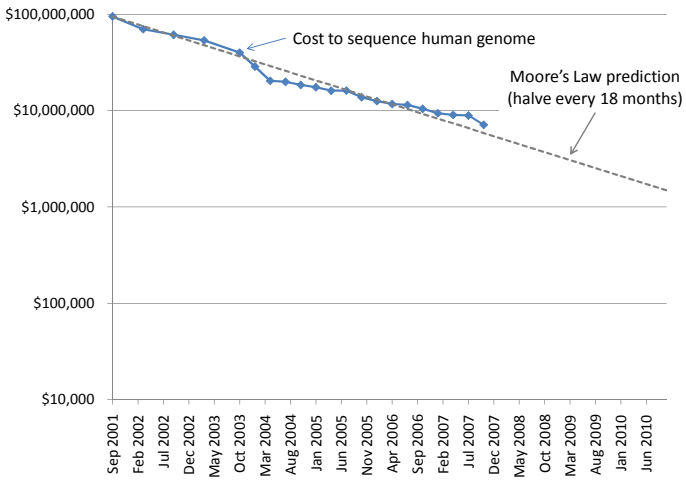
Genome Sequencing

1990: Human Genome Project starts, estimate $3B to sequence one genome ($0.50/base)

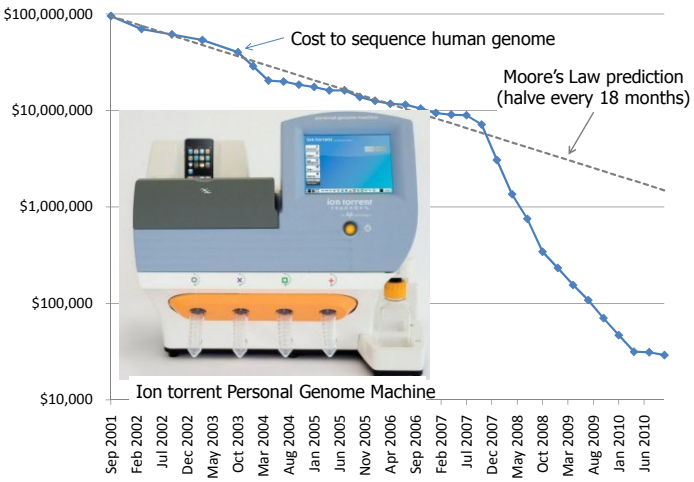
2000: Human Genome Project declared complete, cost ~$300M



Whitehead Institute, MIT



Data from National Human Genome Research Institute: http://www.genome.gov/sequencingcosts



Data from National Human Genome Research Institute: http://www.genome.gov/sequencingcosts

Doubling time: 38000 in 2 years = 48 days

George Church (Personal Genome Project)

Year		Reported sequencing consumables cost	Estimated cost per 40-fold coverage
[redacted]		$10,000,000	$57,000,000
		$1,000,000	$5,700,000
		$250,000	$330,000
		$48,000	$69,000
2009	t	$8,005	$3,700
2009	t	$3,451	$2,200
2009	t	$1,726	$1,500

Human Geno … *ling DNA Nanoarrays*. Radoje Drmanac, An… L. Burns, Bahram G. Kermani, Paolo Carnevali, Igo… Andres Fernandez, Bryan Staker, Krishna P. Par… Ryan Cedeno, Linsu Chen, Dan Chernikoff, Alex… Coleen R. Hacker, Robert Hartlage, Brian Hauser, S… Alvin Kong, Tom Landers, Catherine Le, Jia Liu, Celeste E. McBride, Matt Morenzoni, Robert E. Morey, Karl Mutch, Helena Perazich, Kimberly Perry, Brock A. Peters, Joe Peterson, Charit L. Pethiyagoda, Kaliprasad Pothuraju, Claudia Richter, Abraham M. Rosenbaum, Shaunak Roy, Jay Shafto, Uladzislau Sharanhovich, Karen W. Shannon, Conrad G. Sheppy, Michel Sun, Joseph V. Thakuria, Anne Tran, Dylan Vu, Alexander Wait Zaranek, Xiaodi Wu, Snezana Drmanac, Arnold R. Oliphant, William C. Banyai, Bruce Martin, Dennis G. Ballinger, George M. Church, Clifford A. Reid. ***Science*, January 2010.**

Scheme Grammar

Program ::= ε | *ProgramElement Program*

ProgramElement ::= *Expression* | *Definition*

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

Expression ::= *PrimitiveExpression* | *NameExpression*
| *ApplicationExpression*
| *ProcedureExpression* | *IfExpression*

PrimitiveExpression ::= ***Number*** | **true** | **false**
| ***PrimitiveProcedure***

NameExpression ::= ***Name***

ApplicationExpression ::= (*Expression MoreExpressions*)

MoreExpressions ::= ε | *Expression MoreExpressions*

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

Parameters ::= ε | ***Name*** *Parameters*

IfExpression ::= (**if** $Expression_{Pred}$ $Expression_{Consequent}$ $Expression_{Alt}$)

Assigning Meanings

Program ::= ε | *ProgramElement Program*
ProgramElement ::= *Expression* | *Definition*
Definition ::= (define ***Name*** *Expression*)
Expression ::= *PrimitiveExpression* | *NameExpression*
| *ApplicationExpression* | *ProcedureExpression* | *IfExpression*
PrimitiveExpression ::= ***Number*** | true | false | ***PrimitiveProcedure***
NameExpression ::= ***Name***
ApplicationExpression ::= (*Expression MoreExpressions*)
MoreExpressions ::= ε | *Expression MoreExpressions*
ProcedureExpression ::= (lambda (*Parameters*) *Expression*)
Parameters ::= ε | ***Name*** *Parameters*
IfExpression ::= (if $Expression_{Pred}$ $Expression_{Consequent}$ $Expression_{Alt}$)

ME → E ME
ME → ε

This grammar generates (nearly) all surface forms in the Scheme language.
What do we need to do to **know the meaning of every Scheme program**?

Expressions and Values

Expression ::= *PrimitiveExpression* | *NameExpression*
| *ApplicationExpression*
| *ProcedureExpression* | *IfExpression*

When an expression with a value is *evaluated*, a value is produced

Our goal is to define a meaning function, **Eval**, that defines the value of every Scheme expression:
Eval(*Expression*) ⇒ **Value**
Today we do this informally with rules in English.
(In PS7 we will do it with a program.)

Primitive Expressions

PrimitiveExpression ::= ***Number*** | true | false | ***PrimitiveProcedure***

Evaluation Rule 1: Primitives

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

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

Primitives are the ***smallest units of meaning***: they can't be broken down further, you need to know what they mean.

Name Expressions

Expression ::= *NameExpression*
NameExpression ::= ***Name***

Evaluation Rule 2: Names

A *name* evaluates to the value associated with that name.

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

Caveat: this simple rule only works if the value associated with a name never changes (until PS5).

Application Expressions

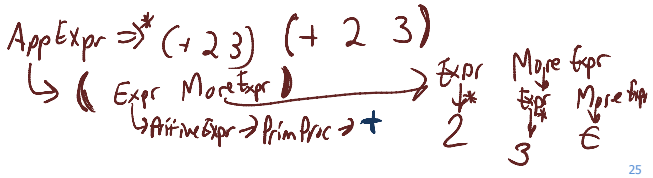
Expression ::= *Application Expression*

ApplicationExpression

::= (*Expression MoreExpressions*)

MoreExpressions ::= ε

MoreExpressions ::= *Expression MoreExpressions*



Evaluation Rule 3: Application

3. To evaluation an application expression:
   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*$_2$ …)

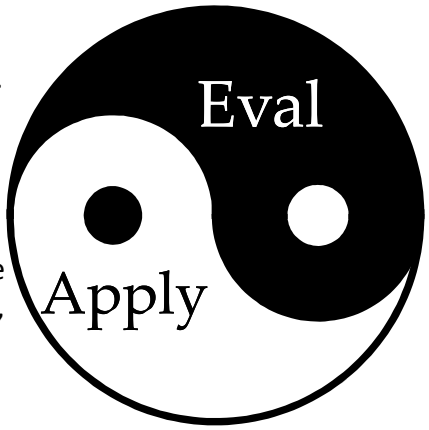
Rules for Application

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

2. **Constructed Procedures.** If the procedure is a *constructed procedure*, **evaluate** the body of the procedure with each parameter name bound to the corresponding input expression value.

Eval and **Apply** are defined recursively.

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



Language Elements

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

Charge

We will cover the rest of the rules Wednesday, then you will know enough to describe every possible computation!

Reading: should be finished with Chapter 3 now, Chapter 4, Gleick Ch 1-3 by Friday

PS1 is Due Monday: Find your assigned partner

Get started earlier and take advantage of scheduled help hours