CSCI 334:
Principles of Programming Languages

Lecture 2: Lisp Wrapup & Fundamentals

Instructor: Dan Barowy

Williams

Garbage Collection

```
~] java -verbose:gc Garbage

[GC 17024K->3633K(83008K), 0.0067267 secs]
[GC 20657K->6988K(83008K), 0.0073014 secs]
[GC 24012K->10505K(83008K), 0.0059666 secs]
...
[GC 121496K->108035K(126912K), 0.0077921 secs]
[Full GC 125059K->110934K(126912K), 0.1330559 secs]
[Full GC 126911K->114224K(126912K), 0.1077395 secs]
[GC 114543K(126912K), 0.0021219 secs]
...
```


Higher-Order Functions

i.e., "functions that take functions"

```
(mapcar #'function list)
```

Activity

Write a function (using mapcar) that replaces the number 3 in a list with the number 6

```
(mapcar #'function list)
```

Higher-Order Functions

Write a function (using `mapcar`) that replaces the number 3 in a list with the number 6

```
(defun my-replace (x)
  (cond ((equal x 3) 6)
        (T x))
)

(mapcar #'my-replace '(1 2 3 4 5 6))

'(1 2 6 4 5 6)
```

Programs As Data

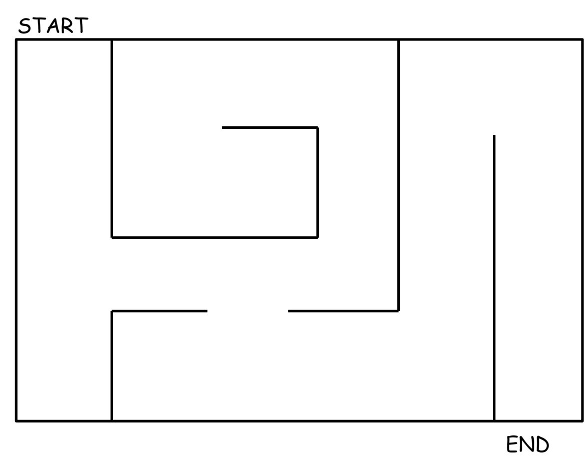
```
; substitute "to" for "from" in "term"
(defun substitute (to from term)
  (cond ((atom term)
              (cond ((eq term from) to)
                    (t term)))
        (t  (cons (substitute to from (car term))
                  (substitute to from (cdr term))))))


(substitute 3 'w '(+ (- 5 w) (* w w)))

     is (+ (- 5 3) (* 3 3))
```


Programs As Data

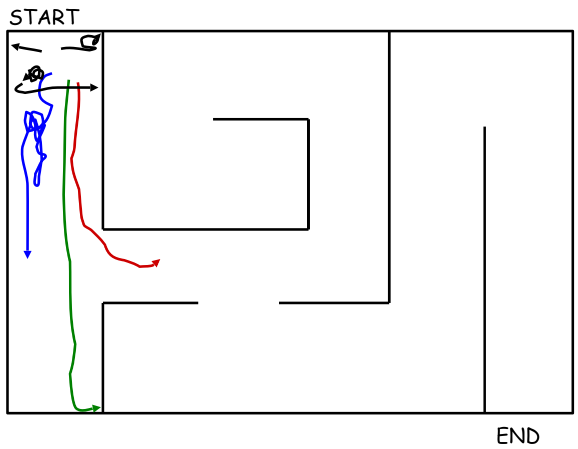
```
(defun substitute-and-eval (to from term)
  (eval (substitute to from term)))

(substitute-and-eval '* '+ '(+ 10 2 3))
  evaluates to 60


(derivative '(* 3 x x))  ->  '(* 6 x)

(substitute-and-eval 6
                    'x
                    (derivative '(* 3 x x))) -> 36
```


Genetic Programming

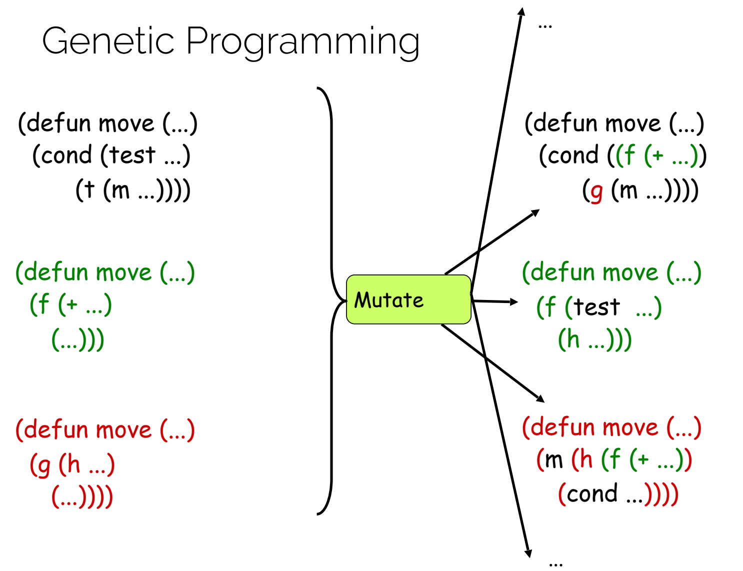
GP Example



Genetic Programming



Genetic Programming



Rule Based Systems

```
(rule symptom-predicate diagnosis treatment confidence)

(rule (and (> temp 99) (headache) (cough))
      (flu)
      (take tylenol)
      0.75))

(rule (and (williams-student) (sleeping-in-class))
      (African Trypanosomiasis)
      (prescribe pentamidine)
      1.0))

for rule X:
(if (and (symptoms X) (> (confidence X) 0.5))
    (print (diagnosis X) "-->" (treatment X))
```


Summary

- Successful language
  - symbolic computation, experimental programming
- Specific language ideas
  - expression-oriented: functions and recursion
  - lists as basic data structures
  - programs as data, with universal function eval
  - idea of garbage collection

https://exploringdata.github.io/vis/programming-languages-influence-network/

Parts of a language

1. Syntax
   - the text of a program
   - samples: http://www.rosettacode.org/wiki/Reverse_a_string
2. Semantics
   - the "meaning" or effect of a program
   - book discusses denotational system for describing semantics
   - here's another (operational semantics)

$$\frac{\langle C_1, s\rangle \longrightarrow s'}{\langle C_1; C_2, s\rangle \longrightarrow \langle C_2, s'\rangle} \qquad \frac{\langle C_1, s\rangle \longrightarrow \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \longrightarrow \langle C_1'; C_2, s'\rangle} \qquad \frac{}{\langle \mathbf{skip}, s\rangle \longrightarrow s}$$

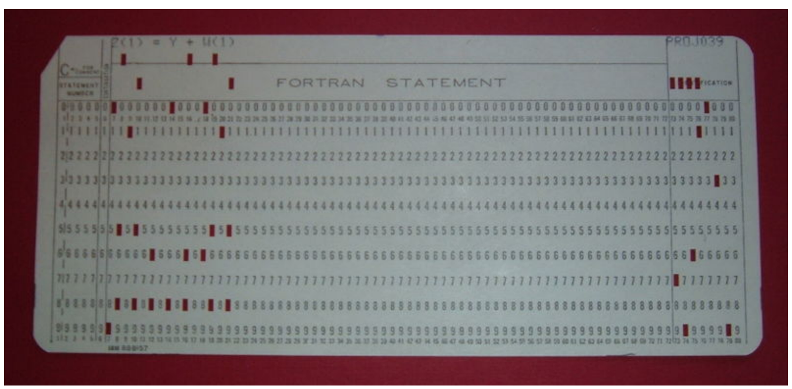
Level of Abstraction

- Concrete (assembly,C,C++,...)

```
movf 0x1233, fp2
mulf #60.0, fp2
movf $8(sp), fp1
addf fp2, fp1
movf fp1, $12(sp)
```

- More abstract (Python,Java,Lisp,...)
  - Lisp: lists, mapcar, higher-order functions
- Very Abstract (LIM,Dylan,FP,Agda)
  - Agda:

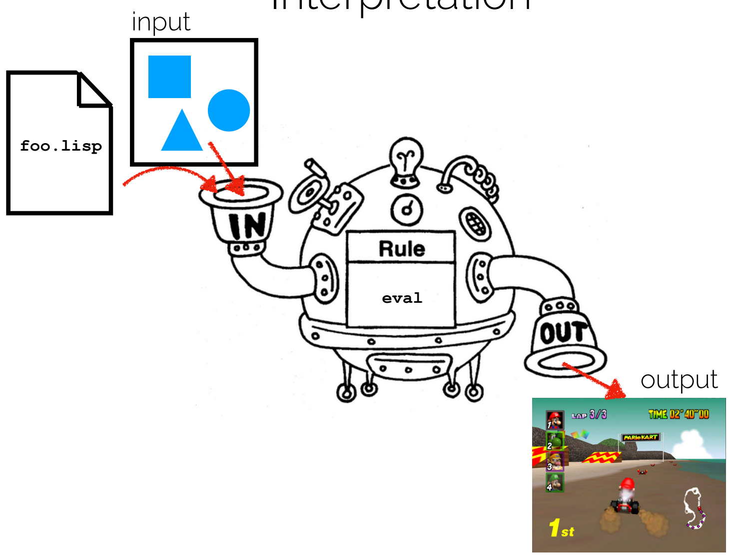
```
_∘_ : {A B C : Set} → (B → C) → (A → B) → A → C
(f ∘ g) x = f (g x)
```

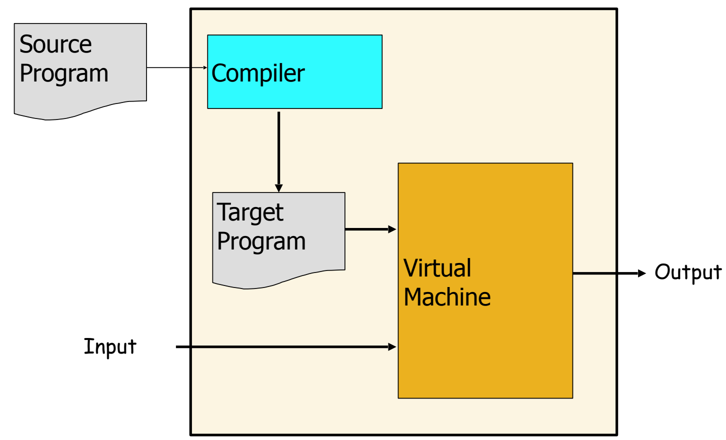
How do programs run?

1. Interpretation
2. Compilation

Interpretation



Inside an interpreter



Virtual machine

- VM simplifies (abstracts) hardware:
  - define types of data
  - define operations on data
- Why?
  - language implementation is easier
  - "porting language" easier
    - only need to redefine data / ops for each new hardware platform
  - language can be evolved rapidly

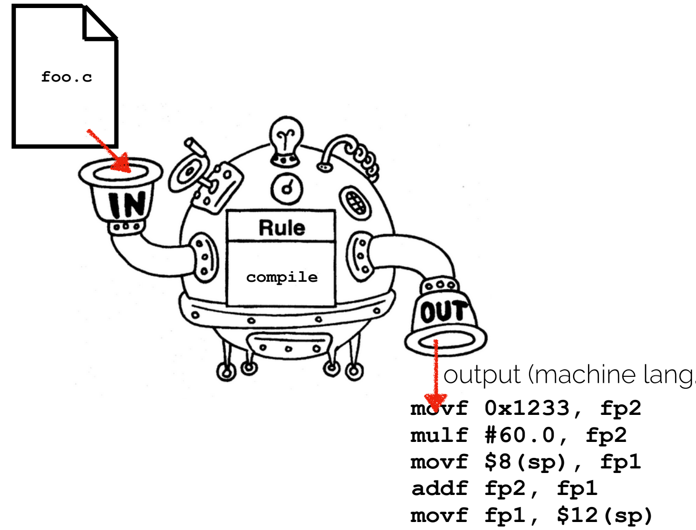
Downsides

- Usually (very) slow
  (often 100-200x slower than compilation)
- Program is just source code so "reverse engineering" is trivial

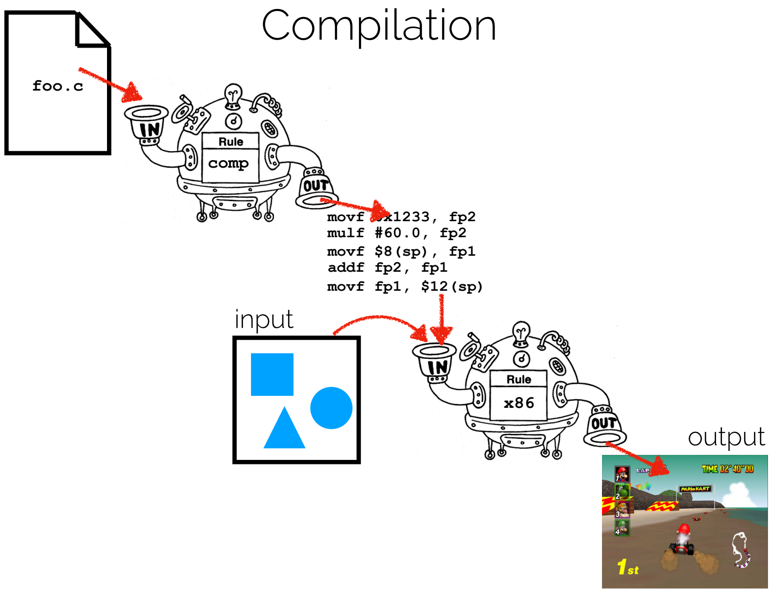
Some interpreted languages

- Most Lisps
- Python
- Ruby
- MATLAB
- R
- (sort of) Java and JavaScript

Compilation



Compilation



Some compiled languages

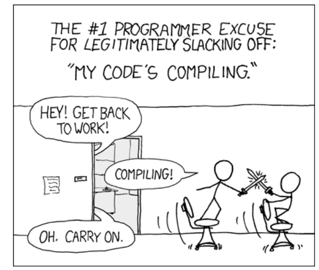
- assembly
- C
- C++
- Go
- FORTRAN
- Java (huh?)
- C# (ditto)

Advantages

- Usually (very) fast
  (often 1.5-2X slower than hand-optimized assembly code)
- Compiled program is in machine (binary) format; hard to reverse engineer (commercial software is often compiled)

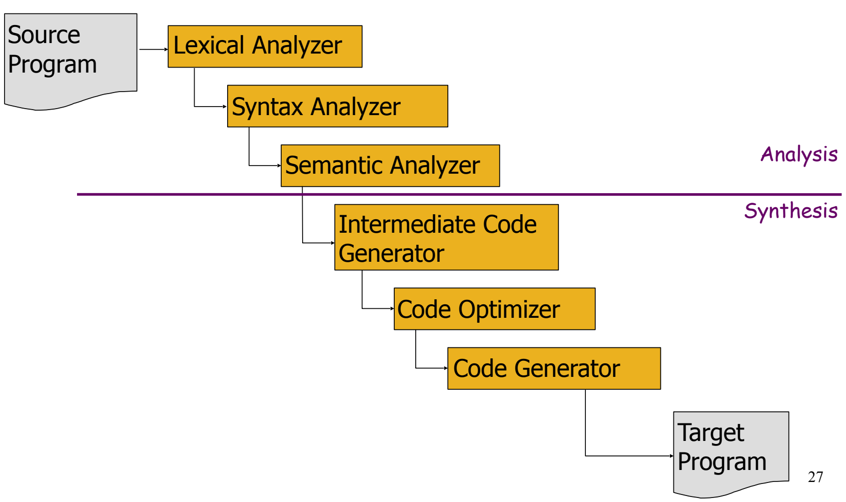
Downsides

- Compilation can take a long time

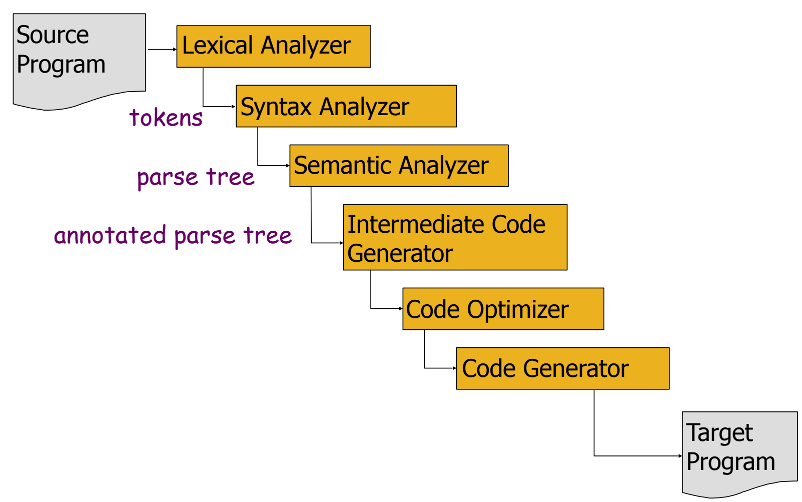


- Cannot modify program without code.
- Hard to evolve language; compilers are complex.

Typical Compiler



Typical Compiler



```
<expr> ::=  <num>
        |  <expr> + <expr>
        |  <expr> * <expr>
<num>  ::=  0 | 1 | 2 | ...
```

- Non-Terminals:
  `<expr>, <num>`
- Terminals:
  `0, 1, 2, 3, ...., + , *`
- Productions for each non-terminal

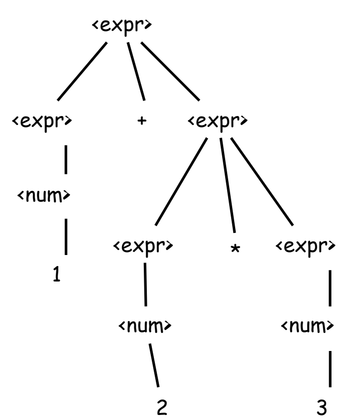
John Backus

- Turing Award, 1977
- Designed FORTRAN
- Invented BNF Notation
  - influenced by Chomsky's work on context-free grammars

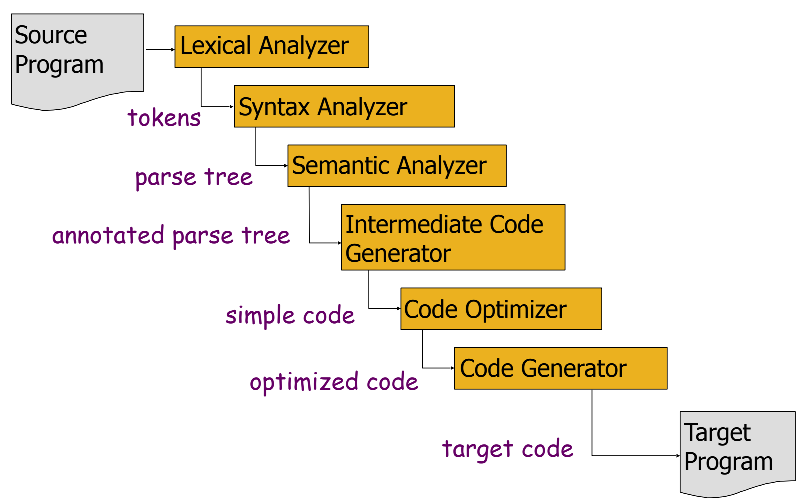

```
<expr> ::= <num>
         | <expr> + <expr>
         | <expr> * <expr>
<num>  ::= 0 | 1 | 2 | ...
```

```
<expr> → <num>
       → 5
```

```
<expr> → <expr> + <expr>
       → <num> + <expr>
       → 1 + <expr>
       → 1 + <expr> * <expr>
       → 1 + <num> * <expr>
       → 1 + 2 * <expr>
       → 1 + 2 * <num>
       → 1 + 2 * 3
```



Typical Compiler



Compiler "Back End" Stages

- Intermediate Code:

```
temp1 = convert_int_to_double(60)
temp2 = mult(rate, temp1)
temp3 = add(initial, temp2)
position = temp3
```

- Optimized Code:

```
temp1 = mult(rate, 60.0)
position = add(initial, temp1)
```

- Generated Machine Code:

```
movf rate, fp2
mulf #60.0, fp2
movf initial, fp1
addf fp2, fp1
movf fp1, position
```


Some hybrid (JIT) languages

- Java
- JavaScript

JIT-Compilation