# Lecture 3: LISP & Scheme, Compilers & Interpreters

CSC 131
Fall, 2008

Kim Bruce

# TA Hours

- Pomona CS lab: Sam, Th 8 to 10 p.m.
- HMC Beckman B 105 (Hot Air balloon lab):
  - Alejandro, W 8 to 10 p.m.
  - Marquis, Th 9 to 11 p.m

# Defining functions

- (lambda (x) (* x x)) *anonymous function*
- (define z 22) *naming exp*
- (define square (lambda (x) (* x x))) *or*
- (define (square x) (* x x))

# Recursive Functions

- ```
  (define (append l1 l2)
    (if (null? l1)
       l2
       (cons (car l1) (append (cdr l1) l2)))
    )
  ```
- (append '(1 2 3) '(4 5 6))

# More functions

- Predefined list functions:
  - (map f '(a b c d)) ⇒ ((f a) (f b) (f c) (f d))
  - (member 1 '(3 2 1 0)) ⇒ (1 0)
- Local variables:

```
- (define (roots a b c)
     (let ((disc (- (* b b) (* 4 a c))))
        (if (>= disc 0) (list (/ (+ (- 0 b) (sqrt disc)) (* 2 a))
                              (/ (- (- 0 b) (sqrt disc)) (* 2 a)))
                        '(0 0)
        )
     )
  )
```

# Dynamically Typed

- Types associated w/ values instead of variables.
- Values have tag w/type
- (* a b) -- actual operation depends on whether both ints, both doubles, or one something else
- Requires run-time check for type safety
```

# Evaluation

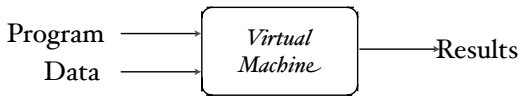
- Very successful in AI & elsewhere
- Good for experimental programming
- Blur boundaries between data and program
- Simple abstract machine:
  - Atoms and cons cells -- simple representation
  - expression, continuation, association list (environment), and heap.

# *Virtual Machines*

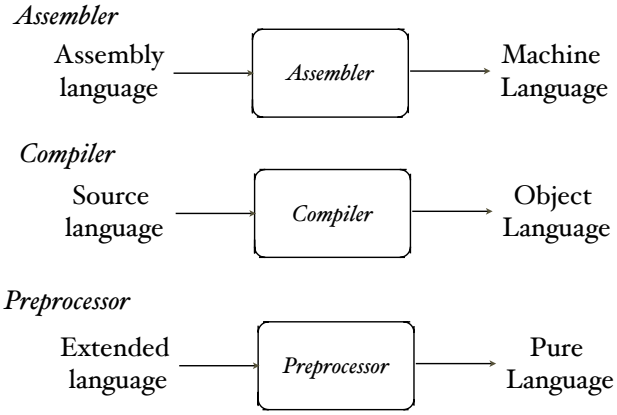
# Abstraction

- *Dijkstra*: Originally we were obligated to write programs so that a computer could execute them. Now we write the programs and the computer has the obligation to understand and execute them.
- Progress in PL design marked by increasing support for abstraction.
- What are data types and how to construct new?
- What are ops and how do we construct new?

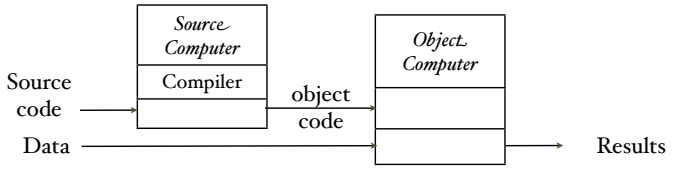
# Creating an Illusion




# Pure Translators

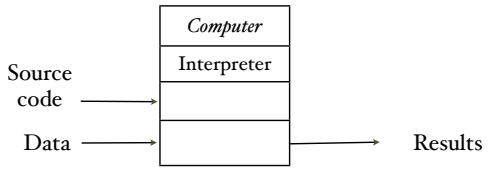



# Execution w/Compiler




*Virtual Machine*

# Interpreter




*Virtual Machine*

# Virtual Machine of Language

- Virtual machine defined by language
- Machine language is set of instructions supported by language
- Layers of VM's
  - bare Intel Chip ⇒ Mac OS ⇒ Java ⇒ Application
- Describe language by VM it defines

# VM of Language

- Problems:
  - Different implementors may have different conceptions of virtual machine
  - Different computers may provide different facilities and operations
  - Implementors may make different choices as to how to simulate elements of virtual computer
- May lead to different semantics, even on same computer.

# VM Problems

- How ensure different implementations give same semantics?
- Sometimes VM's are explicit
  - Pascal P-code & P-machine
  - Modula-2 M-code
  - Java VM & JVML

# More Detail: Interpreters

- Simulate virtual machine:

```
REPEAT
   Get next statement
   Determine action(s) to be executed
   Call routine to perform action
UNTIL done
```

# More Detail: Compiler

- Translate from one VM to another
  - Translate all units of program into object code
  - Link into single relocatable machine code
  - Load into memory
  - Begin execution

# Compiler vs Interpreter

| *Compiler* | *Interpreter* |
|---|---|
| Only translate each statement once | Translate only if executed. |
| Speed of execution | Error messages tied to source. More supportive environment. *No longer as true* |
| Only object code in memory when executing. May take more space because of expansion | Must have interpreter in memory while executing (though source may be more compact) |

# Lack of Purity

- Rarely pure compiler or interpreter
  - Typically compile source into form easier to interpret.
  - Ex. Remove white space & comments, build symbol table, or parse each line and store in more compact form (e.g. tree)
- Java originally hybrid
  - Compile into JVML and then interpreted
  - Now use just-in-time compiler

# Compiler Structure

- Analysis:
  - Break into lexical items, build parse tree, annotate parse tree e.g. via type checking)
- Synthesis:
  - generate simple intermediate code, optimization (look at instructions in context), code generation, linking and loading.