From Programs to Executions:
An Odyssey in Language Translation

(with examples in Scheme)

Keith D. Cooper

Rice University
Houston, Texas

December 2000

An Example

Sum the series

n + n-1 + n-2 + … + 1

In Scheme, we might write

```
(define (summation n)
   (cond  [(= n 0) 0]
             [else  (+ n  (summation (sub1 n)))]))
(summation 3)
```

How do we really go from (summation 3) to an answer?

The Standard Answer

We explain DrScheme’s behavior by saying that it performs a series of rewriting steps

```
 (summation 3)
⇒  (cond  [(= 3 0) 0]
             [else  (+ 3  (summation (sub1 3)))])
⇒ (+ 3  (summation 2))
⇒ (+ 3 (cond  [(= 2 0) 0]
                 [else  (+ 2  (summation (sub1 2)))]))
⇒ (+ 3 (+ 2  (summation 1)))
⇒ (+ 3 (+ 2  (cond  [(= 1 0) 0]
                 [else  (+ 1  (summation (sub1 1)))])))
```

The Standard Answer *(continued)*

… a *long* series of rewriting steps …

```
⇒ (+ 3 (+ 2 (+ 1  (summation 0)))))
⇒ (+ 3 (+ 2 (+ 1   (cond  [(= 0 0) 0]
                         [else  (+ 0  (summation (sub1 0)))]))))
⇒ (+ 3 (+ 2 (+ 1   0)))
⇒ (+ 3 (+ 2 1))
⇒ (+ 3  3 )
⇒ 6
```

It eventually produces the answer: **6**

Is that how it really works? Probably not

Does it matter? Not unless we can tell the difference

The Big Lie(s)

Programming languages deal with abstractions

- Infinite precision numbers
- Symbols
- Lists, structs, vectors, trees
- Functions, programs, name spaces (*local*)

Computers deal with a limited repertoire of simpler ideas

- Finite integers, floating-point numbers (*approximate* $\boldsymbol{R}^n$)
- Memory locations
- Small set of fundamental operations (*add, sub, mult, div* …)

Language implementation must make good on the lies!

What is DrScheme?

Imagine a contract for DrScheme:

DrScheme: program x inputs → results

DrScheme is a *program* that manipulates *programs*

In particular, it

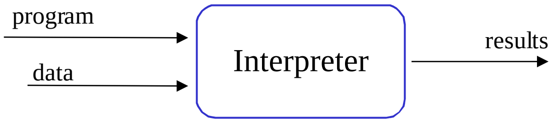
- Creates and maintains the Scheme Environment
  - Functions, objects, definitions,
  - Abstractions like “local” and “define-struct”
- Checks to see that programs are well formed
- Executes programs

DrScheme *implements* the programming language Scheme

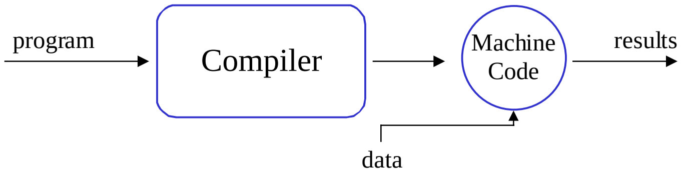
Implementing Programming Languages

Two principal ways to “implement” a language

- Interpreter: program x inputs ⟶ results



- Compiler: program ⟶ program



Inside an Interpreter

- Represent the program in some internal form

  (+ 3 4 5) ⇒ (list + 3 4 5)

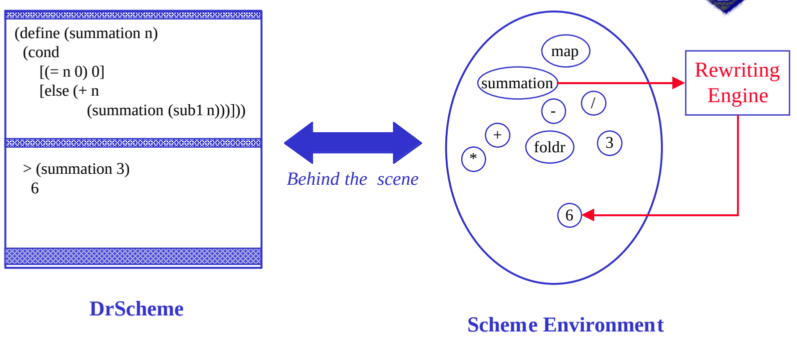
- Traverse that data structure and produce answers

  (list + 3 4 5) ⇒ 12

Along the way

- Manages the name space
  - Variables, arguments/parameters, symbols, free variables
- Manages storage (the computer’s memory)
- Manages communication with outside world
  - Programmer or user, external files, other programs …

The Conceptual View

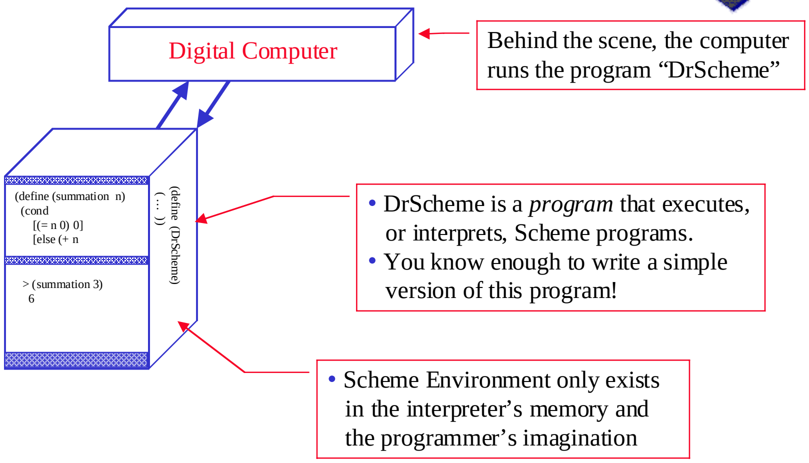


```
(define (summation n)
  (cond
    [(= n 0) 0]
    [else (+ n
             (summation (sub1 n)))]))
```

```
> (summation 3)
  6
```

1. You enter your code in the definitions window
2. You enter an expression in the interactions window
3. DrScheme rewrites until it has a solution

What Really Happens?



```
(define (summation n)
  (cond
    [(= n 0) 0]
    [else (+ n
```

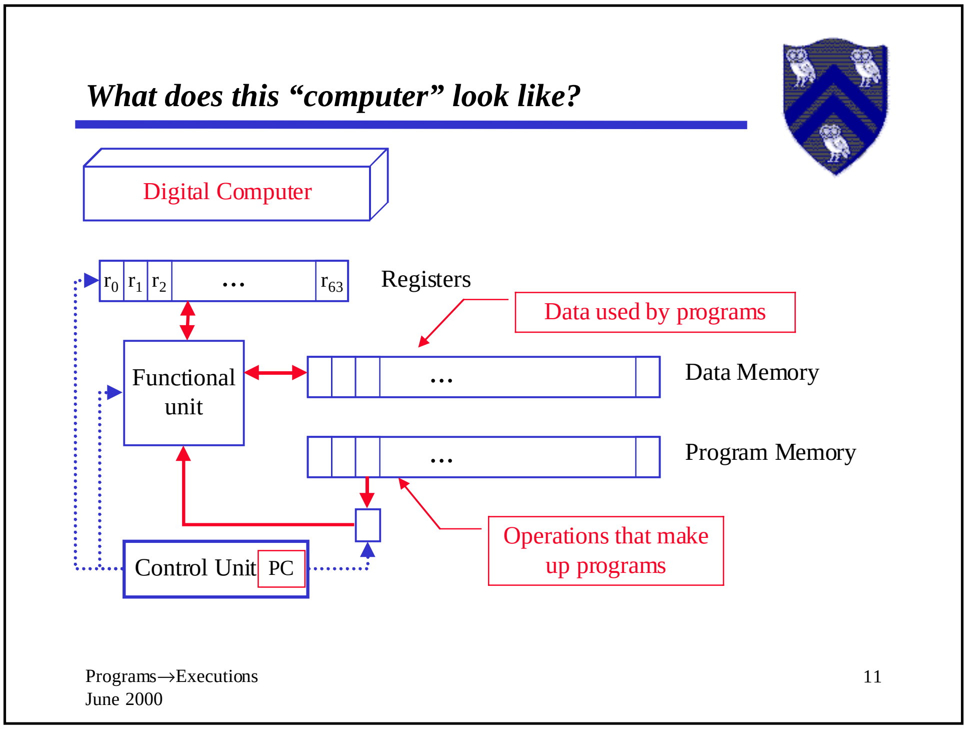
```
> (summation 3)
  6
```

```
(define (DrScheme)
  (… ))
```

Behind the scene, the computer runs the program “DrScheme”

- DrScheme is a *program* that executes, or interprets, Scheme programs.
- You know enough to write a simple version of this program!

- Scheme Environment only exists in the interpreter’s memory and the programmer’s imagination



What does this "computer" look like?

Digital Computer

r_0 r_1 r_2 … r_{63}

Registers

Data used by programs

Functional unit

…

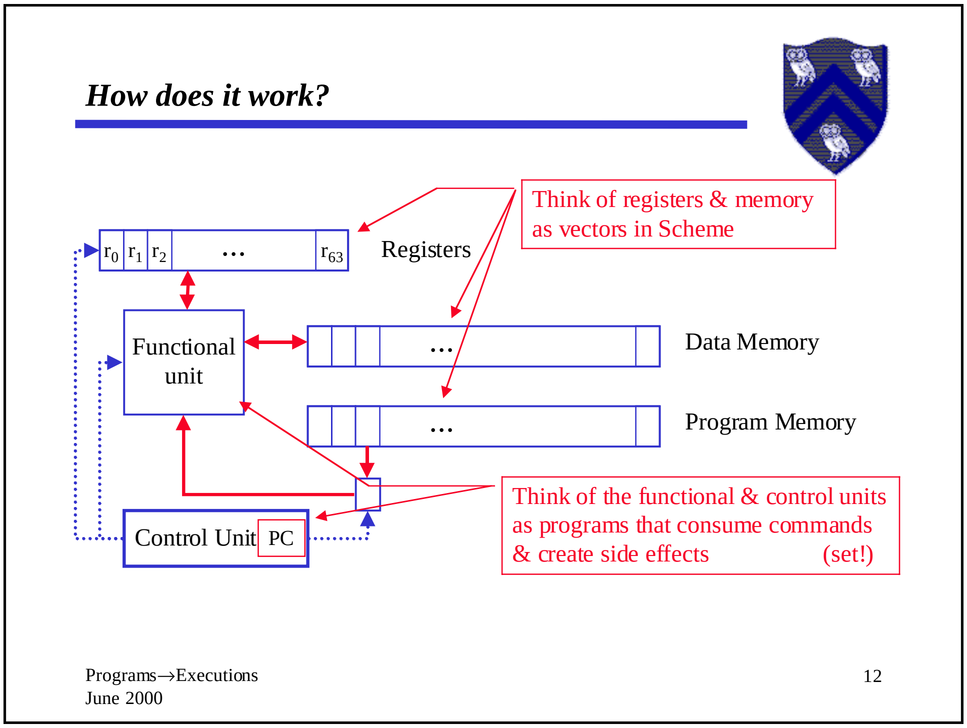
Data Memory

…

Program Memory

Operations that make up programs

Control Unit PC



How does it work?

Think of registers & memory as vectors in Scheme

r_0 r_1 r_2 … r_{63}

Registers

Functional unit

…

Data Memory

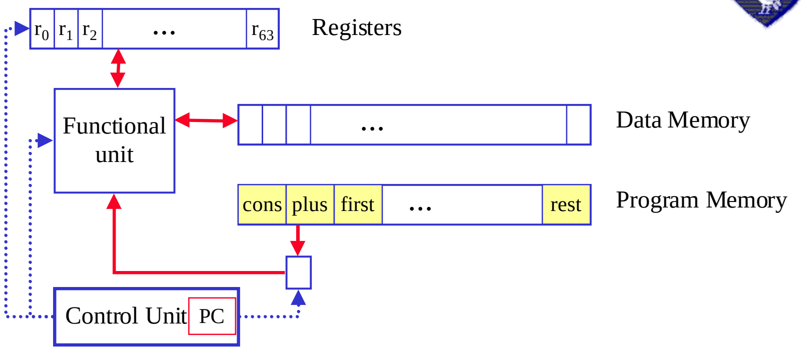
…

Program Memory

Think of the functional & control units as programs that consume commands & create side effects (set!)

Control Unit PC

Are the commands in Scheme?



r_0 r_1 r_2 … r_{63} Registers

Functional unit

… Data Memory

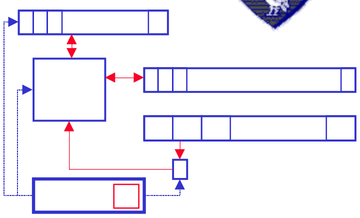
cons plus first … rest Program Memory

Control Unit PC

Such computers have been built

- They have not proven to be cost effective
- More general processors are the rule (today)

What commands does the "computer" run?



Computer's *instruction set*

- Low-level, imperative commands
  - Arithmetic operations
  - Memory operations
  - Control operations
  - Location-oriented programming
- We call these operations "assembly-language"

Arithmetic Operations	*Memory Operations*	*Control Operations*
add r1, r2 => r3	load r1 => r2	branch r1 -> r2
sub r1, r2 => r3	store r1 => r2	branchi r1 -> L2
mult r1, r2 => r3	loadi c1 => r2	call -> L1
div r1, r2 => r3	copy r1 => r2	return