From Programs to Executions: An Odyssey in Language Translation

(with examples in Scheme)

Keith D. Cooper

Rice University
Houston, Texas

Fall 2004

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* $\mathcal{R}^n$)
- 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

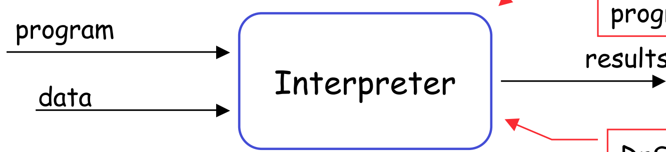
DrScheme *implements* the programming language Scheme

Implementing Programming Languages

Two principal ways to "implement" a language

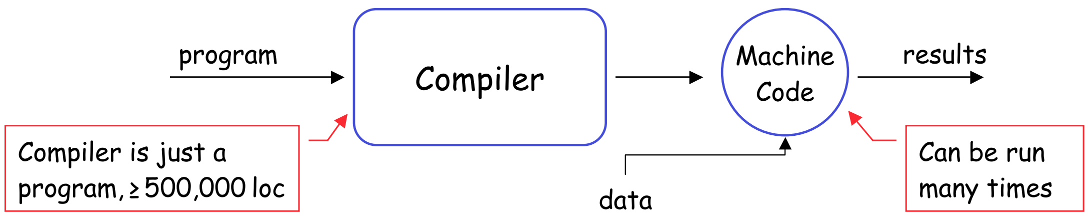
- Interpreter: program x inputs → results

Interpreter is just a program in some language



DrScheme contains about 60,000 lines of C code

- Compiler: program → program



Inside an Interpreter

- Represent the program in some internal form
  - (+ 3 4 5) ⇒ (cons + (cons 3 (cons 4 (cons 5 nil))))
- Traverse that data structure and produce answers
  - (+ 3 4 5) ⇒ 12

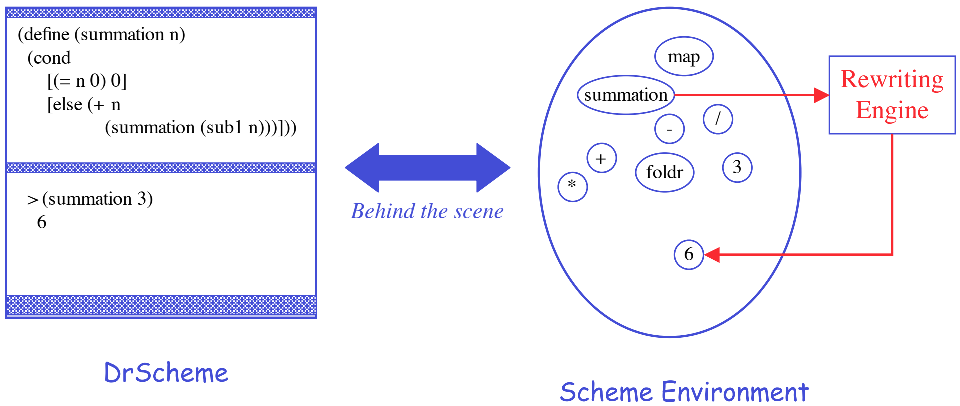
How many names are there in Scheme?

How many lists?

Along the way

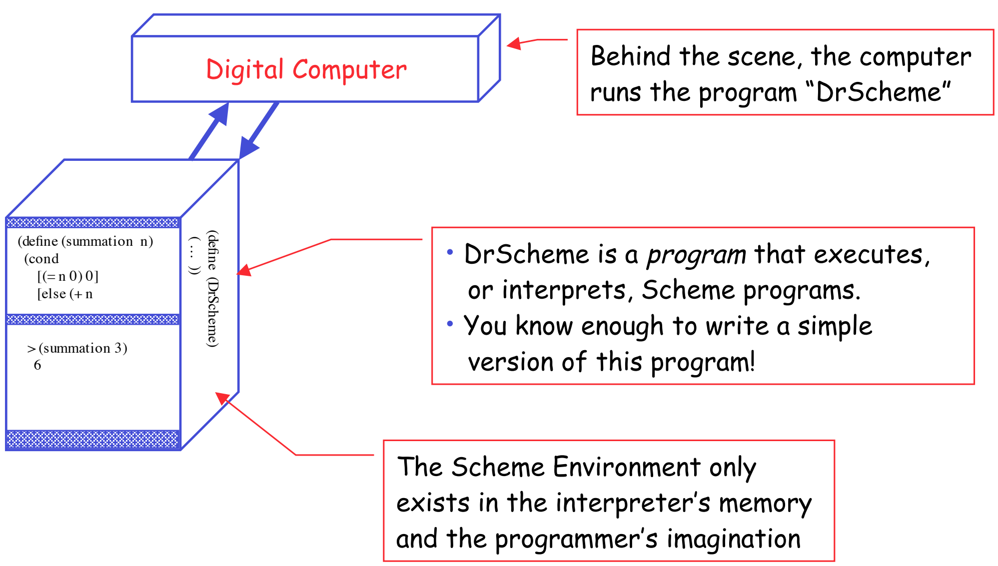
- Manages the name space
  - Variables, arguments/parameters, symbols, free variables
- Manages storage (the computer's memory)
- Manages communication with outside world
  - Programmer or user, external files, other programs …

The Conceptual View

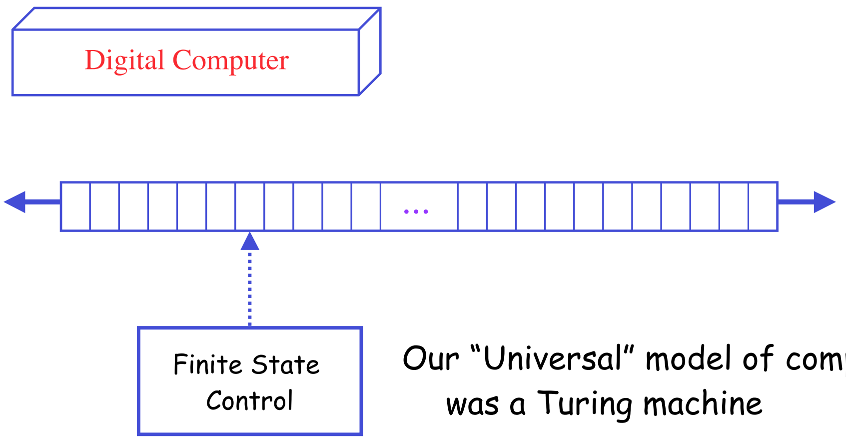


1. You enter your code in the definitions window
2. You enter an expression in the interactions window
3. DrScheme rewrites until it has a solution

What Really Happens?



What does this "computer" look like?



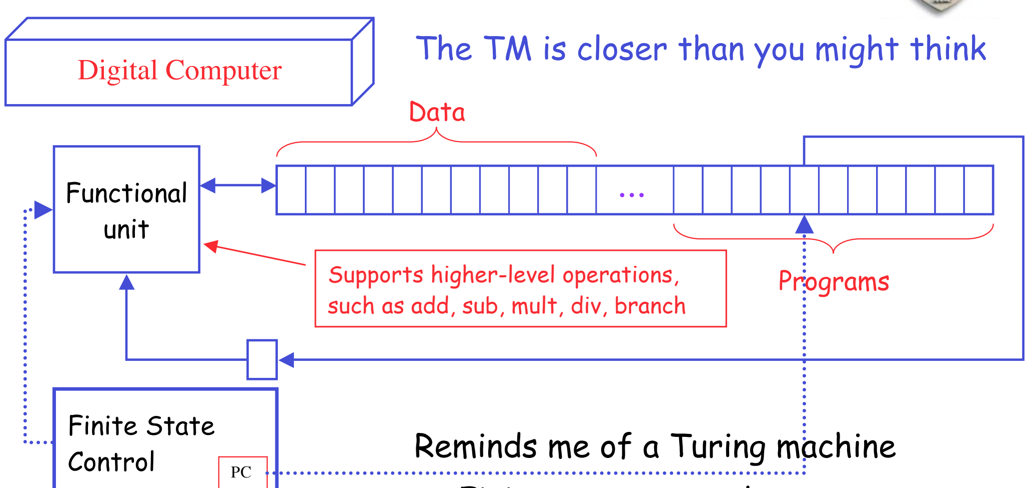
Our "Universal" model of computation was a Turing machine

- Finite state control
- Infinite storage tape

Is this how a "digital computer" works?

What does this "computer" look like?

The TM is closer than you might think



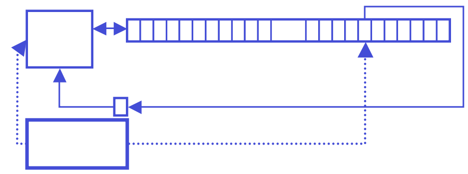
Reminds me of a Turing machine

- Finite state control
- Large (finite) storage
- We added a "functional unit"

What commands does the “computer” run?

Computer’s *instruction set*

- Low-level, imperative commands
  - Arithmetic operations
  - Control operations
  - Location-oriented programming
- We call these operations “assembly-language” or “machine code” (you would find them in a Windows .exe)



Arithmetic Operations
add x, y => z
sub x, y => z
mult x, y => z
div x, y => z

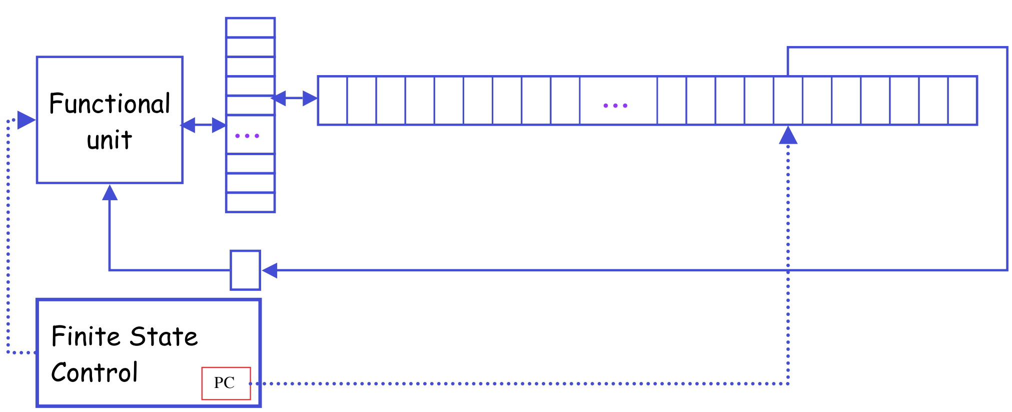
Control Operations
branch x -> y
jump -> y
call -> y
return

- No cons, first, define, …
- All those functions can be implemented with these ops
- Church/Turing thesis says these ops are enough …

One final complication

Memory is slower than functional or control units

- Fast, named, data memory near the processor — “*registers*”
- Load & store ops move data between memory & registers
- Other ops now refer to registers for data (args & results)



Programming with Machine Operations

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

This might become, after storage assignment & translation

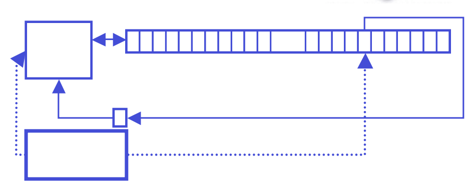
1: n
2: result

100:	loadi	1	$\Rightarrow r_1$
101:	load	r_1	$\Rightarrow r_2$
102:	copy	r_1	$\Rightarrow r_3$
103:	add	$r_3\ r_2$	$\Rightarrow r_3$
104:	sub	$r_2\ r_1$	$\Rightarrow r_2$
105:	eq?	$r_2\ r_1$	$\Rightarrow r_{10}$
106:	branch	r_{10}	$\rightarrow$ 108
107:	jump		$\rightarrow$ 103
108:	loadi	2	$\Rightarrow r_{11}$
109:	store	r_2	$\Rightarrow r_{11}$
110:	stop		

Assembly programming & the design of assemblers are taught in COMP/ELEC 320

Understanding How a Computer Works

One valuable tool is simulation
- Write a program that has the same behavior
- Models behavior of system
- Gives insight into its workings

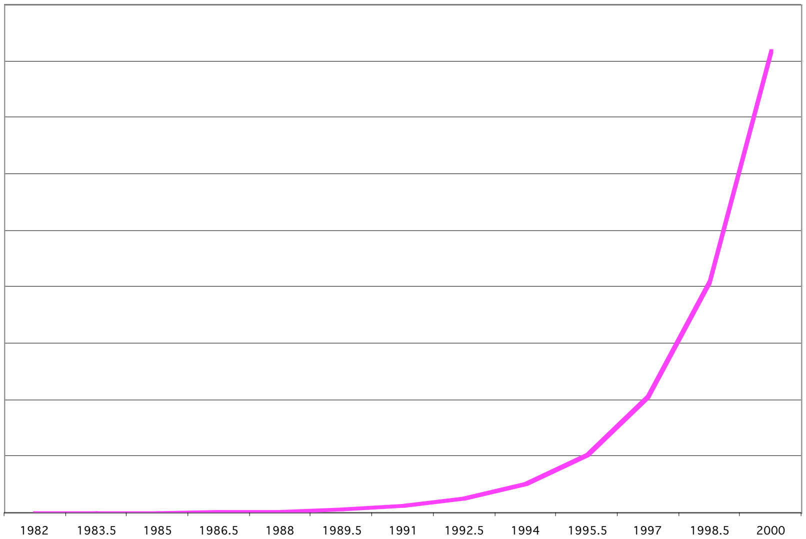


Writing a simulator for a simple computer is a common exercise for 2nd or 3rd year CS students

Simulation is used in many ways
- Design of new systems
- Conduct experiments that are expensive or dangerous
- Train pilots in cases where loss is expensive

Simulating a computer shows us a lot about how it works

Computers Keep Getting Faster



Processor power versus time, 1980 to 2000

Plot of Moore's Law

1980-2000

How did this happen?

They Are Running Faster

The clock frequency of processors has risen

- 1983: 10 MHz 68020 provided about 1 MIP
- 2004: 1 GHz PowerPC provides 1000 MIPS + 4 GFLOP
- 2004: Pentiums in 3 to 4 GHz range

High-end chips are heading toward 2 processors per chip

⇒ 10 cycles/operation

⇒ 1 cycle/operation

Finite-state control has gotten better!

All this power has a downside, however

- Power consumption ∝ frequency²
- Heat ∝ power
  - Hence, Intel's announcement of multiple cores rather than higher frequencies for the next generation of processors … it's about heat!
- Computation needs operands, needs memory

They Are Also Running More Operations

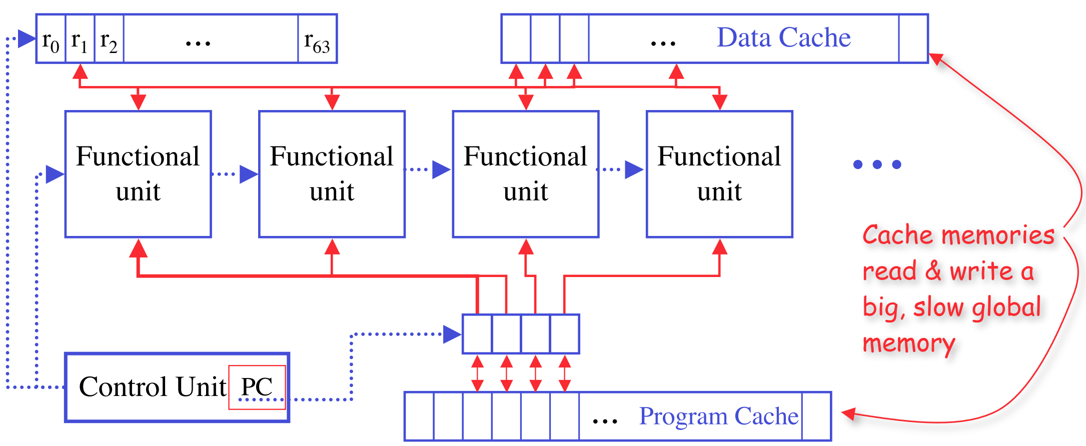
Programs contain parallelism

- Operations that can execute at the same time
- Can occur at the fine-grained level or on a larger scale

Computers can exploit parallelism

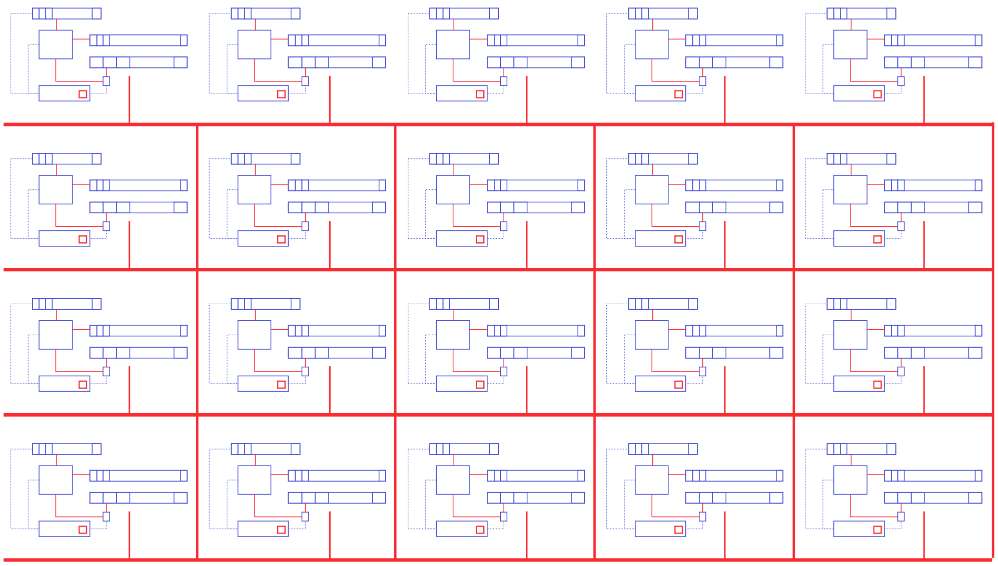
- Increase operations per cycle
- Use more hardware rather than faster hardware
- Many options
  - Single chip processor with many functional units
  - Custom-built machine with many individual nodes (computers)
  - Networks of workstations and/or PCs

Single Chip, Many Functional Units



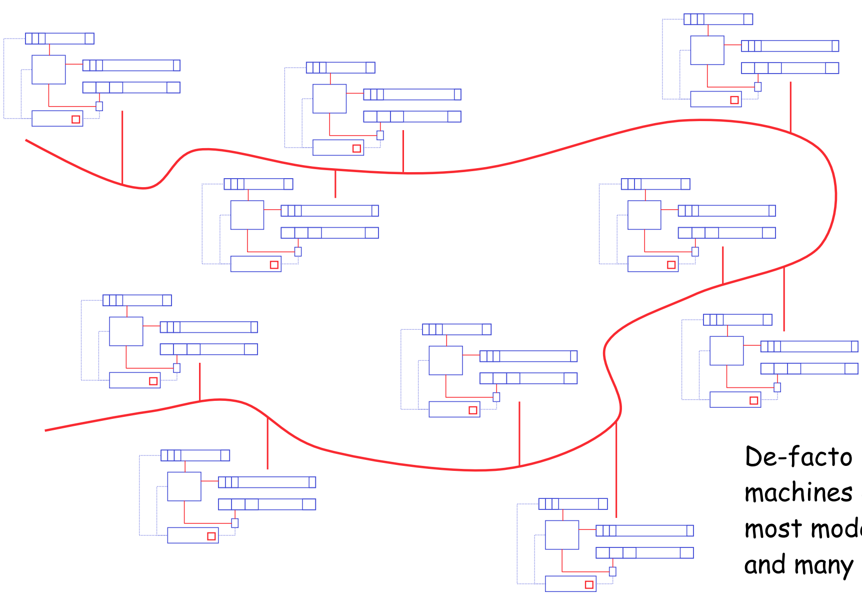
This takes advantage of instruction-level parallelism

Many Processors, Dedicated Interconnect



Multiprocessor Computer

Network of Workstations



De-facto parallel machines exist in most modern offices and many homes!