Crash Course Session 1—Recursion, Iteration, Lists

CS 1102S—Data Structures and Algorithms

Martin Henz

14 January, 2010

Languages vs Implementation

Programming language

Programming languages are the languages in which a programmer writes the instructions that the computer will ultimately execute. *Encyclopedia Britannica*

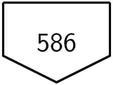
Programming system

Set of tools that help achieving this execution.

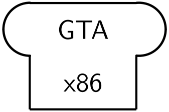
Same language, different tools

For the same language, different tools are available for different purposes.

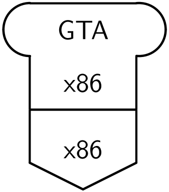
T-Diagrams



x86 Processor



Program “GTA” (x86 code)

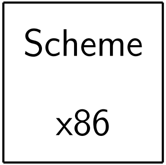


“GTA” running on x86

Interpreter

- Interpreter is program that executes another program
- The interpreter's *source language* is the language in which the interpreter is written
- The interpreter's *target language* is the language in which the programs are written which the interpreter can execute

Interpreters

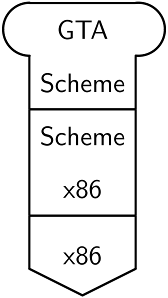


Scheme

x86

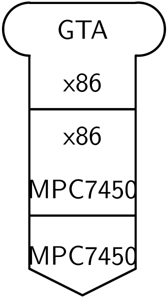
Interpreter for Scheme (x86 machine code)

Interpreting a Program



Scheme program “GTA”
running on x86 using interpretation

Hardware Emulation

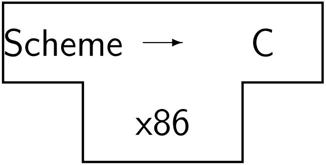


"GTA" x86 executable running on a PowerPC using hardware emulation

Translators

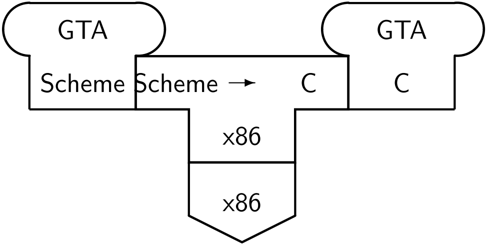
- Translator translates from one language—the *from-language*—to another language—the *to-language*
- Compiler translates from “high-level” language to “low-level” language
- De-compiler translates from “low-level” language to “high-level” language

T-Diagram of Translator



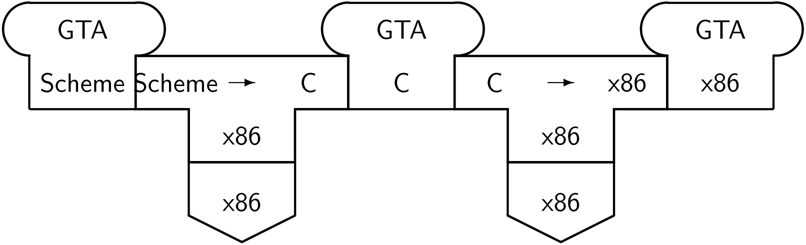
Scheme-to-C compiler written in x86 machine code

Compilation



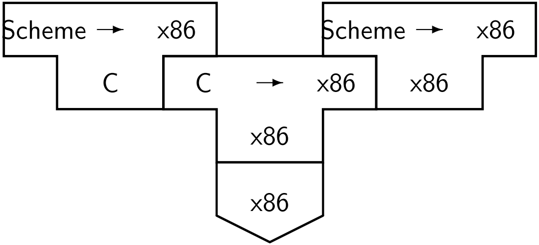
Compiling “GTA” from Scheme to C

Two-stage Compilation



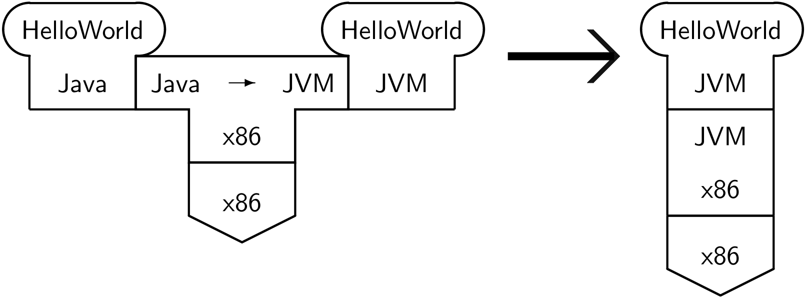
Compiling “GTA” from Scheme to C to x86 machine code

Compiling a Compiler



Compiling a Scheme-to-x86 compiler from C to x86 machine code

Typical Execution of Java Programs



Compiling “HelloWorld” from Java to JVM code, and running the JVM code on a JVM running on an x86

Factorial Function

```
(define (factorial i)
  (if (<= i 0)
    1
    (* i (factorial (- i 1)))))
```

Factorial In Java

```
public static int factorial(int i) {
        if (i <= 1) {
                return 1;
        } else {
            return i * factorial(i - 1);
        }
}
```

Iteration vs Recursion in Scheme

Iteration

A (recursive) Scheme function is *iterative*, if the recursive call is always the last thing to do in its body.

Is Factorial Iterative?

No! `(* i (factorial (- i 1)))`
In Java: `i * factorial(i - 1);`

Iterative Factorial Function In Scheme

```
(define (iterfactorial i acc)
  (if (<= i 1)
    acc
    (iterfactorial (− i 1) (∗ acc i))))

 (define (iterativefactorial i)
   (iterfactorial i 1))
```

Iterative Factorial Function In Java?

```
private static int iterFactorialTry(int i,int acc){
  if (i <= 1) {
    return acc;
  } else {
    return iterFactorialTry(i-1,acc*i);
  }
}
public static int iterativeFactorialTry(int i) {
        return iterFactorialTry(i,1);
}
```

The Sad Truth about Java

Java has no iterative recursion!

Every function call requires space on a Java runtime stack.

Recursion is always recursive!

A recursive function in Java will never use constant space.

Loops to the rescue!

Loop constucts

Java contains loop constructs such as while and for.

Iteration in Java

Iteration can only be achieved using loops in Java.

Iterative Factorial Function In Java

```
public static int iterativeFactorial(int i) {
  int acc = 1;
  while (i > 1) {
    acc = acc * i;
    i = i - 1;
  }
  return acc;
}
```

Lists in Scheme and Java

Lists in Scheme

Built-in, using cons, car, cdr, '() , null?.

Lists in Java

There is a List interface in Java, see http://java.sun.com/j2se/1.5.0/docs/api/java/util/List.html

Start with List of Integers

Here, we study a restricted form of lists first: IntList .

Lists in Scheme

```
(cons 1 2)            ;; a pair
’()                   ;; an empty list
(cons 7 ’())          ;; a list with one integer
(cons 4 (cons 9 ’())) ;; a list with two integers
```

Builtin Operations on Lists in Scheme

```
'()                    ;; an empty list
(cons 1 2)             ;; a pair
(car alist)            ;; first component (head)
(cdr alist)            ;; second component (tail)
(null? alist)          ;; whether list is empty
```

Operations on Lists of Integers in Java

```
public static IntList nil = new IntList()
public static IntList cons(int i, IntList list)
public static int car(IntList list)
public static IntList cdr(IntList list)
public static boolean isNil(IntList list)
public static void print(IntList list)
```

`IntList`

These functions are available in the library (class) `IntList`.

Some Cheating (for convenience)

```
public static IntList intList(int [] elements)
```

Length in Scheme

```
(define (length xs)
  (if (null? xs)
    0
    (+ 1 (length (cdr xs)))))
```

Length in Java

```
public static int length(IntList aList) {
  if (IntList.isNil(aList)) {
    return 0;
  } else {
    return 1 + length(IntList.cdr(aList));
  }
}
```

Iterative Length in Scheme

```
(define (iterlength alist acc)
  (if (null? alist)
    acc
    (iterlength (cdr alist) (+ acc 1))))

(define (iterativelength alist)
  (iterlength alist 0))
```

Iterative Length in Java?

```
public static int iterLengthTry(IntList aList,
                                int acc) {
  if (IntList.isNil(aList)) {
    return acc;
  } else {
    return iterLengthTry(IntList.cdr(aList),acc+1);
  }
}
public static int iterativeLengthTry(IntList aLst){
  return iterLengthTry(aLst,0);
}
```

Iterative Length in Java!

```
public static int iterativeLength(IntList aList) {
  int acc = 0;
  while (! IntList.isNil(aList)) {
    aList = IntList.cdr(aList);
    acc++;
  }
  return acc;
}
```

Append in Scheme

```
(define (append alist anotherlist)
  (if (null? alist)
    anotherlist
    (cons (car alist)
          (append (cdr alist) anotherlist))))
```

Append in Java

```
public static IntList append(IntList aList, IntList
  if (IntList.isNil(aList)) {
    return anotherList;
  } else {
    return
      IntList.cons(IntList.car(aList),
                   append(IntList.cdr(aList),
                          anotherList));
  }
}
```

Naive Reverse in Scheme

```
(define (naivereverse alist)
  (if (null? alist)
     ’()
    (append (naivereverse (cdr alist))
                             (cons (car alist) ’())))))
```

Naive Reverse in Java

```
public static IntList naiveReverse(IntList aList) {
  if (IntList.isNil(aList)) {
     return IntList.nil;
  } else {
    return append(naiveReverse(IntList.cdr(aList)),
                  IntList.cons(IntList.car(aList),
                               IntList.nil));
  }
}
```

Square All in Scheme

```
(define (squareall alist)
  (if (null? alist)
    '()
    (cons (* (car alist) (car alist))
          (squareall (cdr alist)))))
```

Square All in Java

```
public static IntList squareAll(IntList aList) {
  if (IntList.isNil(aList)) {
    return IntList.nil;
  } else {
    return
      IntList.cons(IntList.car(aList)
                   * IntList.car(aList),
                   squareAll(IntList.cdr(aList)));
  }
}
```

Sum in Scheme

```
(define (sum alist)
  (if (null? alist)
    0
    (+ (car alist) (sum (cdr alist)))))
```

Sum in Java

```
public static int sum(IntList aList) {
  if (IntList.isNil(aList)) {
    return 0;
  } else {
    return IntList.car(aList)
           + sum(IntList.cdr(aList));
  }
}
```

Next Session

- More built-in types
- Loops
- Arrays