Writing a Design Document & How Recursive Functions are Implemented

Week 8
CS 212 - Spring 2008

Announcements

• Project Part 3
  • Online by tomorrow afternoon for both Compiler & GBA
  • Design Document is due on Thursday, March 27
  • Part 3 code is due near Thursday, April 10

Software Life Cycle (Again)

• Problem specification
• Program design
• Choosing algorithms & data structures
• Coding & debugging
• Testing & verification
• Support & maintenance

Program Design

• Specify a set of components that will solve the specified problem
  • Component?
    • Depends on the language we’re using
    • Typically: function, procedure, class, template, package, or interface
  • Design is a Divide & Conquer operation
    • Break problem into smaller and simpler subproblems

Specifying a Component

• Interface
  • How this component is invoked
• Preconditions
  • The conditions that must be true for this unit to work correctly
• Postconditions
  • The conditions that will be true when the component finishes (assuming the preconditions are met)

Top-Down Design

• Based on subdividing by task (i.e., by function)
  • Example is from “Modern Software Development using Java” by Tymann & Schneider

Banking Simulator

- Arrivals
- Departures
- Transactions
- Print Results
- Input Transaction
- Validate
- Process Deposit
- Process Withdrawal
Criticism of Top-Down (Function-Oriented) Design

- What if we want to add a new type of transaction?
  - Say, make-loan-payment
  - Which components are affected?

<table>
<thead>
<tr>
<th>Arrivals</th>
<th>Departures</th>
<th>Transactions</th>
<th>Print Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Transaction</td>
<td>Validate</td>
<td>Process Deposit</td>
<td>Process Withdrawal</td>
</tr>
</tbody>
</table>

Object-Oriented Design

- Decompose problem via entities (i.e., objects) instead of by function

  - For the banking simulation, we might have
    - Customers
    - Waiting lines
    - Tellers
    - Transactions

  - We then determine the methods for each class

```
Class WaitingLine
  putAtEnd(c) Put new customer at end of line
  c = getFirstCustomer() Remove & return 1st customer in line
  isEmpty() True iff line is empty
  isFull() True iff line is full

Class Teller
  isBusy() True iff teller is busy
  serve(c) Teller begins serving customer c

Class Customer
  depart() The customer leaves the bank
  t = getTransaction() Return customer's desired transaction

Class Transaction
  t = transactionType() Return the type of this transaction
  a = transactionAmount() Return dollar amount of transaction
```

What to Put in Your Design Document

- Specify each class
  - For each class, specify the class's methods
  - For each method, specify
    - Its arguments (i.e., its interface)
    - Its preconditions (i.e., what the method does)

- Specify how the classes interact
  - Diagrams can be useful here, but aren't required
    - UML (Unified Modeling Language) can be used, but informal diagrams are OK, too

- Expected length of design document
  - One page ≡ probably too short
  - Ten pages ≡ definitely too long

Recursive Functions
Positive Integer Powers

- $a^n = a \cdot a \cdot a \cdot \ldots \cdot a$ (n times)

- Alternate description:
  - $a^0 = 1$
  - $a^{n+1} = a \cdot a^n$

```java
static int power(int a, int n) {
    if (n == 0) return 1;
    else return a*power(a, n-1);
}
```

A Smarter Version

- Power computation:
  - $a^0 = 1$
  - If $n$ is nonzero and even, $a^n = (a^{n/2})^2$
  - If $n$ is odd, $a^n = a \cdot (a^{n/2})^2$
  - Java note: If $x$ and $y$ are integers, $x/y$ returns the integer part of the quotient

- Example:
  - $a^5 = a \cdot a^{5/2} = a \cdot (a^{2})^2 = a \cdot ((a^2/2)^2)^2 = a \cdot a^{2}$
  - Note: this requires 3 multiplications rather than 5

- What if $n$ were larger?
  - Savings would be more significant
  - This is much faster than the straightforward computation
  - Straightforward computation: $n$ multiplications
  - Smarter computation: $\log(n)$ multiplications

```java
static int power(int a, int n) {
    if (n == 0) return 1;
    int halfPower = power(a, n/2);
    if (n % 2 == 0) return halfPower*halfPower;
    return halfPower*halfPower*a;
}
```

Smarter Version in Java

- $n = 0$: $a^0 = 1$
- $n$ nonzero and even: $a^n = (a^{n/2})^2$
- $n$ nonzero and odd: $a^n = a \cdot (a^{n/2})^2$

```java
static int power(int a, int n) {
    if (n == 0) return 1;
    int halfPower = power(a, n/2);
    if (n % 2 == 0) return halfPower*halfPower;
    return halfPower*halfPower*a;
}
```

Implementation of Recursive Methods

- Key idea:
  - Use a stack to remember parameters and local variables across recursive calls
  - Each method invocation gets its own stack frame

- A stack frame contains storage for:
  - Local variables of method
  - Parameters of method
  - Return info (return address and return value)
  - Other bookkeeping info

Stack Frame

- A new stack frame is pushed with each recursive call
- The stack frame is popped when the method returns
  - Leaving a return value (if there is one) on top of the Stack

Example: power(2, 5)
**How Do We Keep Track?**

- At any point in execution, many invocations of `power` may be in existence.
  - Many stack frames (all for `power`) may be in Stack.
- Thus there may be several different versions of the variables `a` and `n`.
- How does processor know which location is relevant at a given point in the computation?

**Answer:** Frame Base Register

- When a method is invoked, a frame is created for that method invocation, and FBR is set to point to that frame.
- When the invocation returns, FBR is restored to what it was before the invocation.
- How does machine know what value to restore in the FBR?
  - This is part of the return info in the stack frame.

**Basic Idea for Functions**

- A new frame (on the stack) is created for each function call.
  - We use the FBR (Frame Base Register) to indicate the current frame.
  - When a function returns it should "clean up" its frame.

**What's Kept in a Frame?**

- We already have this principle:
  - When an expression is evaluated, the result is left on top of the stack.

- We know we have to change the FBR for each new frame.
- What do we do with the old FBR?

**What Else is Kept in a Frame?**

- Another principle:
  - Every time a function is called, it has its own local variables.
  - Thus it makes sense to keep a function's local variables in its frame.
- The parameters of a function are also "local variables." They can be kept in the frame, too.

**Is That It? Nothing Else in a Frame?**

- Well, no; there's one more thing.
- We're using assembly language.
  - If we want to jump somewhere then jump back then we must remember where to come back to.
How Do We Jump Back?

- We can store the return address (i.e., a saved PC value) in the frame, too.
- We have provided SAM instructions to store and restore the PC
  - JSR address
    - push PC+1 onto stack; set PC to address
  - Jump to SubRoutine
    - set PC to value on top of stack

- We also have instructions to save and restore the FBR
  - LDNK
    - push value of FBR onto stack; set FBR to SP-1
  - UNLINK
    - set value of FBR to value on top of stack

Creating a Frame

- Responsibility for creating a frame is shared by the caller (calling code) and the callee (the function's code)
  - Caller's responsibilities
    - Push space for return value
    - Push arguments
    - Create new frame (use LDNK = push current FBR and set FBR to SP-1)
    - JSR to callee (push PC+1 and jump to callee)
  - Callee's responsibilities
    - Reserve space for local variables
    - Continue with callee's code

Clearing a Frame (Clean-up)

- Responsibility for clearing a frame is shared by the callee (the function's code) and the caller (calling code)
  - Callee's responsibilities
    - Clear local variables from stack
    - JUMPIND to caller (clear the saved PC and jump back to calling code)
  - Caller's responsibilities
    - Restore the FBR (UNLINK)
    - Clear the arguments from stack
    - Note: return value remains on stack

Access to Frame's Data

- Data stored in the frame are accessed via offset from the FBR
  - Let p be the number of parameters
    - The first local variable
      - STOREOFF 2
    - The second local variable
      - STOREOFF 3
    - The first parameter
      - STOREOFF –p
    - The second parameter
      - STOREOFF –p + 1
    - The return value
      - STOREOFF –p – 1

An Example

```c
int factorial (int n) :
  if n < 2 then return 1;
  else return n * factorial(n-1);
endif
```

```assembly
factorial: PUSHOFF -3
LOAD IMM 2
LESS
JUMP false
JUMPI // Clear the argument
false: PUSHOFF -1
ADDSP 1 // Space for return value
PUSHOFF -1
SUB // Argument is now on stack
LDNK // Create new stack frame
JUMPIND // Call the function
UNLINK // Restore FBR
ADDSP -1 // Clear the argument
TIMES
STOREOFF -2 // Store return value
JUMPIND // Return
```

Example Calling Code

```assembly
program
  ADDSP 1 // Space for return value
  PUSHOFF -3 // The argument
  ADDSP 2 // The second local variable
  ADDSP 3 // The second parameter
  ADDSP -1 // The return value
  STOP
```

- We need this "calling code" to help create factorial's initial frame