# CS212
## Java Practicum

**Fall 2005**
**Lecture 2**
**SaM**



## Announcements

- http://www.cs.cornell.edu/courses/cs212/
- Part1 due on Friday
- Partner list sign-up for Part2+
- CMS sign-up (if you're not on already)



## What is SaM? Why SaM?

- **From last lecture**:
  - computer stores data and instructions in memory
  - ***fetch-and-decode cycle***:
    - retrieve instruction
    - execute it
    - repeat
  - JVM is "average" of computers
  - has own instruction set (bytecode)
- ***SaM***:
  - **a simple stack machine** (with a **heap**)
  - simulation of JVM
  - see SaM on CS212 for full instruction set
  - gives us legible instruction set
    - you will write compiler to generate ***Samcode***
    - BTW, what's a compiler? (last panel...)



## Samcode Instructions

- **Low-level instructions**:
  - push and pop values in memory
  - mnemonics for bit patters
- **Structure**:

```
opcode
opcode operand
```

- **Examples**:

```
PUSHIMM 10
STOP
```

## Using SaM

- **Areas**:
  - Main Window
  - Program Code
  - Stack
  - Heap
  - Registers
  - Console
  - Menus
  - Commands
- **Simplest use**:
  - Create text file with Samcode
  - Open file
  - Run
- **Example** (see previous slide)

5

## Structure of Samcode File

- ASCII Text!
- Write instructions on new lines
- One instruction per line
- `//` indicates single-line comments, which are ignored
- Program ends with **`STOP`**
- Program must leave only one item on Stack

6

## Focus on Stack

- ***Call Stack***
  - function calls function calls ...
  - when last function done, go back, then back, then ...
  - how to picture this structure?
- ***Frame***
  - each function's portion of Stack
  - variables, data, administrative info
- Cells and addresses
  - **start at 0!**
- Helpful picture?

7

## Useful Registers

- ***Frame Based Register*** (***FBR***)
  - administrative information
  - keeps track of current frame (and thus, function)
- ***Stack Pointer*** (***SP***)
  - use register
  - store location of next free cell in stack
- Helpful picture?

8

# Some Instructions

- **ALU**:
  - **ADD** (**SUB**, **TIMES**, ....)
  - **AND** (**OR**, **XOR**, **NOT**, ...) (**0** is false; all else is true)
  - **EQUAL** (**LESS**, ...)
  - Generally follows ***below op top*** (see documentation)
- **Stack Manipulation**:
  - **PUSHIMM** ***c*** (**PUSHIMMF** ***f***, ...)
  - **DUP** (**SWAP**, ...)
  - **PUSHABS** ***k***, **STOREABS** ***k***
  - **PUSHOFF** ***k***, **STOREOFF** ***k***
- **Register**: **ADDSP** ***n***
- **Control**: **STOP**
- Many others!
  - see on-line documentation
  - see **Chapter 1**



# Some Examples

- Notation:
  - Prefix: (1 – 2) – 3
  - Postfix: 1 2 – 3 –
- Logical: ~(4 <= 5)
  - Samcode rem: ***below op top***
- Samcode?



# Program Storage?

- Main memory model:
  - store programs as data
  - so, instructions have bit patterns
- Where are they in SaM?
  - Samcode read into an array
  - array stores instruction objects
- Want more? See documentation and source code
  - **SaM→Individual Files→Core→Instructions**
  - See next page for example
- How to load your own instructions?
  - recompile everything (a pain)
  - use SaM's ***instruction loader***



# Example

```
package edu.cornell.cs.sam.core.instructions;
import edu.cornell.cs.sam.core.*;

public class SAM_ADD extends SamInstruction {
    public void exec() throws SystemException {
        int type1 = mem.getType(cpu.get(SP) - 2);
        int type2 = mem.getType(cpu.get(SP) - 1);
        mem.push(higherPrecedence(type1, type2), mem.pop() + mem.pop());
        cpu.inc(PC);
    }
}
```

# A Bit About Variable Scope

- Example:
  - is the following legal?

```
int x(int x) { return x++; }
int y(int x) { return x(x); }
```

  - why? why not?
- Scope of variable:
  - region of code in which variable represents something
  - how does Java indicate?
- Local and global variables:
  - each function has its own local variables
  - global variables shared



# Variables and Frames

- **A way to picture variables in frames...**
  - variable gets cell
  - Aside: SaM shows type of cell
- **A bit about recursion....**
- **Samcode**:
  - need to allocate cell
  - then fill cell
  - later retrieve/change contents
  - finally deallocate cell (why?)



# Allocation and Deallocation

- Allocate ***v*** amount of vars: **ADDSP *v***
- Deallocate ***v*** amount of vars: **ADDSP -*v***
- Example:

```
ADDSP 3
ADDSP -1
ADDSP -1
ADDSP -1
STOP
// will get an error mesg, though (why?)
```



# How to access a variable?

- **Addressing of variables**:
  - absolute
  - relative
- ***Absolute***:
  - don't worry about your current frame
  - figure out variable address on stack
  - eg) globals
- ***Relative***:
  - do worry about your current frame
  - figure out variable address with respect to FBR value
  - eg) locals

# Absolute Address

- **Instructions**:
  - To **store** a value ***v*** at location ***i***:
    - **PUSHIMM *v***: Stack[SP] ← ***v***; SP++
    - **STOREABS *i***: Stack[***i***] ← Stack[SP-1]; SP--
  - To **retrieve** a value ***v*** from location ***k***:
    - **PUSHABS *k***; Stack[SP] ← Stack[***k***]; SP++
- **Example:**

```
int rv;
int x;
int y;
x = 10;
y = 20;
rv = x + y;
return rv;
```

```
ADDSP 3
PUSHIMM 10
STOREABS 1
PUSHIMM 20
STOREABS 2
PUSHABS 1
PUSHABS 2
ADD
STOREABS 0
ADDSP -2
STOP
```



# Relative Address

- Instructions:
  - To store a value ***v*** at location ***i***:
    - **PUSHIMM *v***: Stack[SP] ← ***v***; SP++
    - **STOREOFF *i***: Stack[***i***+FBR] ← Stack[SP-1]; SP--
  - To retrieve a value ***v*** from location ***k***:
    - **PUSHOFF *k***: Stack[SP] ← Stack[***k***+FBR]; SP++
- Picture?



# Example

```
ADDSP 1     // rv of program
JSR add     // new frame (jump to "add")
STOREOFF 0  // store rv of "add"
STOP        // done

add:        // code for "add" function
LINK        // store old FBR (0) and set new FBR (2)
ADDSP 3     // allocate space for x, y, rv of add
            // rv of add is at relative address 1
PUSHIMM 10  // push value 10
STOREOFF 2  // store 10 in x's cell
PUSHIMM 20  // push value 20
STOREOFF 3  // store 20 in y's cell

PUSHOFF 2   // retrieve x
PUSHOFF 3   // retrieve y
ADD         // x+y
STOREOFF 1  // store x+y as rv of add
ADDSP -2    // deallocate x, y

SWAP        // exchange rv of add for old FBR
UNLINK      // restore old FBR (0)
SWAP        // exchange rv of add for return address
RST         // return to Samcode just after "JSR add"
```

```
public int add()
   int x, y;
   x = 10;
   y = 20;
   return x+y;
}
```

**NOTE: We will use a different frame structure later!**



# Human Compiling

- ***Compiling***:
  - translate **code** (like Java) to machine **code** (like Samcode)
  - compiler (like **javac**) does the work for you
- ***Human Compiling*** (Part 1 of CS212):
  - you identify simple expressions and statements
  - you convert them into Samcode
  - you test your Samcode problems in SaM
  - we grade your correctness and style