Plan

PA3 and PA4

- Look at PA3 peer reviews and some code.
- PA3 demos
- Make sure to indicate group(s) you were in and groups you are building off of in README. You must cite other people's code if you use it!
- Cannot just turn in another group's PA3. Must implement some PA4 features.

Regression testing: Demo of how to use the regress.sh script.

Code generation for function/method calls and definitions

- Can do MOST of the code generation before having a symbol table
- Analyze what nodes in the AST are affected
- Examples

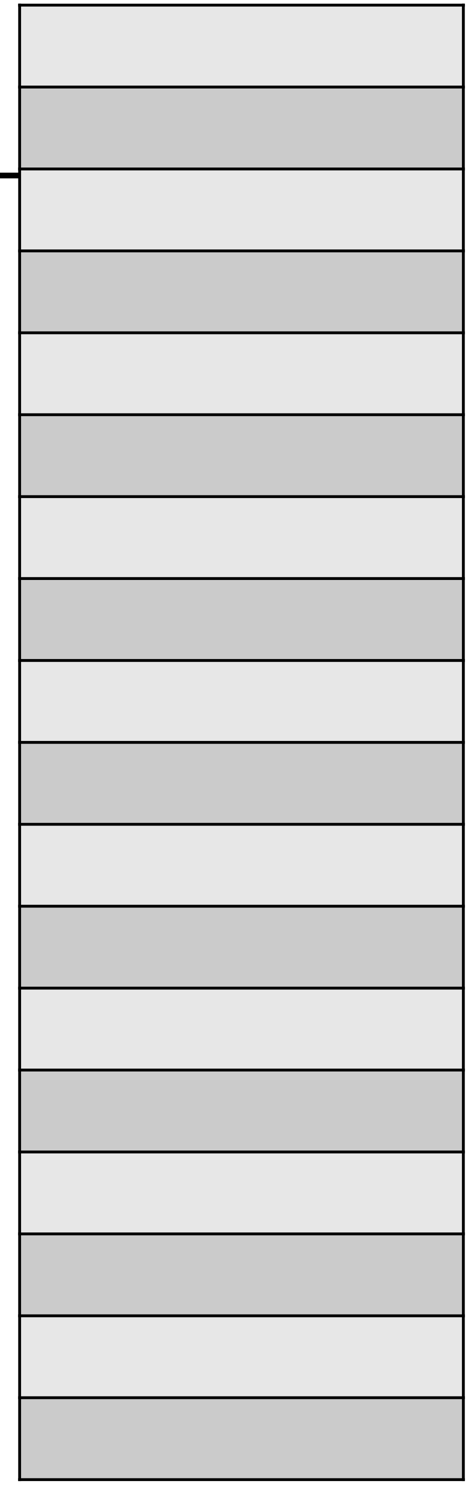
Regress.sh script

Setup

- Create a Test directory with some name (TestPA4Compiler/)
- Copy the regress.sh script into TestPA4Compiler/ and make it executable
- Create a stack project somewhere else: stack new MJCPA4 simple
- Edit the cabal file to include containers and have exec be mjc
- "stack build" and then "stack install"
- Copy mjc binary into TestPA4Compiler/
- Copy MJSIM.jar into same directory.
- Create a WorkingTestCases/ directory in TestPA4Compiler/.
- Put some test cases into WorkingTestCases/.
- Put a copy of the meggy/ Java-only sub directory into TestPA4Compiler/.
- ./regress.sh

RecursiveCount Example in MeggyJava

```
/**
*Recursively put BLUE pixels in (2,0), (1,0), (0,0)
*/
import meggy.Meggy;
class RecursiveCount {
  public static void main(String[] whatever){
    new Foo().count((byte)0);
  }
}
class Foo {
  public void count(byte p) {
    // if haven't reached 2,
      // recursively call count
    // call setPixel at (p,0)
  }
}
```



Recall AVR-GCC Calling Convention

Calling Convention for AVR-GCC

- Pass parameters in registers
    - r24, r25 for parameter 1
    - r22, r23 for parameter 2
    - …
    - r16, r17 for parameter 5
    - …
    - r8, r9 for parameter 9
- Pass return values in register(s), r24, r25
- Call and return instructions implicitly store and use return address on stack
- Push and pop keep track of the stack pointer, which points at next open slot
- Frame pointer is managed internally by each function

Code generation for Function/Method Calls

Already did code generation for

- Meggy.setPixel()
- Meggy.delay()
- Meggy.checkButton()
- Meggy.getPixel()
- How did the above work?

How did we know the types for the actual argument expressions?

How can we know they types for user-defined functions? Return value?

What are the relevant AST nodes for method/function calls?

Outline of Code to Generate at a Function Call

```
# for each actual expression, pop it from the run-time stack into
# appropriate register(s) for parameter pass
       pop r??
       pop r??
       ...

       # call the function
       call classnamefuncname
       → next sequential instruction = return address

       # If we are an expression, then push the return value
       # onto the stack.
       push r25    # only have this if have a 2 byte return value
       push r24
```

call, return, return address

Call and return instructions manipulate the RTS implicitly.

A call instruction:

```
        call  clNmfNm
RA:
```

pushes RA (return address) on the RTS and jumps to clNmfNm.

A return instruction:

```
        ret
```

pops the return address off the RTS and jumps to it.

Stack Pointer vs Frame Pointer

Stack Pointer

– used to evaluate expressions
– moves around while executing a function body
– points at first available open slot in the Run Time Stack

Frame Pointer

– used to address parameters and locals
– does not vary during the execution of a function body
– gets updated at the beginning of a method call and reset and end

Notice that Run Time Stack actually grows Down in memory (in spite of pictures on following slides), so when offsetting off frame pointer use Y+1, Y+2 for this, Y+3 for first parameter if byte, Y+3 and Y+4 if int, etc.

Code Generation at the Method/Function Definitions

Where should the code be generated for method/function definitions?

```
    .text
.global methodname
    .type methodname, @function
methodname:
    # push callers frame pointer
    push r29
    push r28
    # store off parameter(s)
    push r24
    …
    # make callee’s frame pointer copy of stack pointer
    in r28, __SP_L__
    in r29, __SP_H__
```

Code Generation at the Method/Function Definitions

Epilogue

```
# handle return value


# pop parameters off stack


# restore the frame pointer


# return
ret
.size methodname, .-methodname
```

Calling convention

Caller:

1. gather actual params on the RTS

- push receiver

  (receiver = "this" in callee)

- eval and push explicit parameters 2,3,…

2. call

- pop actuals in reg (pair)s

- call fname

  (fname = className+funcName)

3. on return

(1) push return value on stack

Callee:

1. push old FP (r28, r29)

2. make space for frame

multiple push 0-s

3. copy SP → FP

in r28, __SP_L__

in r29, __SP_H__

4. populate frame (Reg → Y+offset)

5. execute body

may push return value

6. may get return value into r24(25)

7. clear frame space (undo 2)

8. pop FP into r28,r29

9. ret

PA4simple.java example: call

new C().setP((byte)3 ,(byte)7,Meggy.Color.BLUE);

← *SP*

← *mFP*

1. caller pushes
actual params
new C() = 0,0
(byte) 3
(byte)7
(byte)BLUE
and pops them into
r18: BLUE r20: 7
r22: 3 r24(25) newC()

2. caller performs
CALL CsetP

RA:

← *SP*

← *mFP*

CsetP: callee

1. *saves mFP on RTS*
2. *makes space for parameters by pushing 0-s*
3. *copies SP to FP*

callee

4. populates stack frame

5. executes body

(6. in case of return pops ret expr into r24(25))

← *SP, FP*
0
0
0
0
0
mFP
RA

mFP ←

← *FP*
0
0
3
7
BLUE
mFP
RA

PA4simple.java example: Returning control to caller

← SP, FP

0

0

3

7

BLUE

mFP

RA

callee

(0. in case of return pops return expr and puts it in r24(25))

1. pops frame off RTS exposing mFP

2. pops mFP into FP exposing RA

3. executes ret, which pops RA and jumps to it

← SP

← mFP

caller

1. resumes execution at RA

(2. If return value pushes it on stack.)