Project 11: Target Code Generation

File you will create: `Emit.java`

Code in Main:

```
emit := new Emit ();
emit.emitAll ();
```

Run through IR statements.
For each, generate SPARC instructions.
Print on "stdout".

Files

New File:

```
Emit.java
```

Slight modifications:

```
Main.java
makefile
```

Use, if necessary:

```
Lexer.class
Parser.class
Checker.class
Generator.class
```

Scripts, for testing:

```
run / runAll / go
```

New Scripts:

```
run2 / runAll2 / go2
pc
```

Other files:

```
Ast.java
IR.java
```

...etc...

Test Files:

```
simple.pcat
simple.out.bak
simple.error.bak
```

`simple.s` $\left.\right\}$ *You will produce these*
`simple`

```
simple.givenInput
simple.givenOutput1
simple.givenOutput2
```

“go2” script

```
go2 simple
```

```
cat simple.pcat
rm simple.s
java Main < simple.pcat > simple.s
cat simple.s
gcc simple.s -o simple
simple
```

“pc” script (PCAT Compile)

```
pc simple
```

Same as go2, except no “cat”s.
Compile-assemble-go

run / runAll

Same as other projects, with new test files.

run2 / runAll2

```
run2 simple
```

Compile and execute using “pc”
Supply simple.givenInput
Compare output to simple.givenOutput1 / simple.givenOutput2
Print differences

Code Generation Algorithm #1

For each IR statement...
Generate several SPARC instructions
... to do the job.

Grading Criterion:
Output must match .out.bak and .err.bak files.

Code Generation Algorithm #1

For each IR statement...
Generate several SPARC instructions
... to do the job.

Grading Criterion:
Output must match .out.bak and .err.bak files.

You may implement a more complex code generation algorithm.
... AFTER you get code generation algorithm #1 working!!!

Grading Criterion:
The executable must have "functional equivalence".
Use the run2 and runAll2 tests!

Approach

Start with printIR in IR.java
Leave in the "print" statements.
Prefix them with "!___"

```
System.out.print ("BLAH-BLAH-BLAH");
    ↓
System.out.print ("!___BLAH-BLAH-BLAH");
```

Approach

Start with printIR in IR.java
Leave in the "print" statements.
Prefix them with "!___"

```
System.out.print ("BLAH-BLAH-BLAH");
   ↓
System.out.print ("!___BLAH-BLAH-BLAH");
```

OPcomment

OPiadd

$\left.\right\}$ Sequence of IR instructions

OPimul

OPcomment

Approach

Start with printIR in IR.java
Leave in the "print" statements.
Prefix them with "!___"

```
System.out.print ("BLAH-BLAH-BLAH");
   ↓
System.out.print ("!___BLAH-BLAH-BLAH");
```

OPcomment
```
! ASSIGNMENT STMT...
```
OPiadd
```
!   t1 := x + y
```
OPimul
```
!   z := t1 * 5
```
OPcomment
```
! IF STMT...
```

Approach

Start with printIR in IR.java
Leave in the "print" statements.
Prefix them with "!___"

```
System.out.print ("BLAH-BLAH-BLAH");
    ↓
System.out.print ("!___BLAH-BLAH-BLAH");
```

OPcomment
```
   ! ASSIGNMENT STMT...
```
OPiadd
```
   !   t1 := x + y
             XXXXXXXX
             XXXXXXXX
             XXXXXXXX
```
$\}$ *SPARC code*

OPimul
```
   !   z := t1 * 5
             XXXXXXXX
             XXXXXXXX
             XXXXXXXX
```
$\}$ *SPARC code*

OPcomment
```
   ! IF STMT...
```

Boilerplate

A method called emitBoilerplate *?*

```
! PCAT Compiler Version 1.0
                .global   .div
                .global   .rem
                .data
temp:           .double   0
                .text
strNL:          .asciz    "\n"
strInt:         .asciz    "%d"
strFlt:         .asciz    "%g"
strTrue:        .asciz    "TRUE"
strFalse:       .asciz    "FALSE"

message1:       .asciz    "Error: Allocation failed!\n"
                ...etc. for other 4 runtime error messages...

runtimeError1:  XXXXX
                call      printf
                XXXXX
                XXXXX
                call      exit
                ...etc. for other 4 runtime errors...
```

```
writeFlt:        XXXXX
                 XXXXX
                 ret
                 restore
writeBool:       XXXXX
                 XXXXX
                 ret
                 restore
                 .data
display0:        .word    0
display1:        .word    0
                 ...etc...
display8:        .word    0
                 .text
float1:          .single  0r12.34
float2:          .single  0r3.1415
                 ...etc...
str1:            .asciz   "Hello, world!"
str2:            .asciz   "This test is..."
                 ...etc...
```

Support Routines
Normally, the support routines (printf, strcpy, etc.) would be linked in, as necessary, from separately compiled library routines.)

Display Regs
(generate as many as "MaxLexicalLevel")

Float List

String List

Consider generating code for `OPiadd`...

```
w := y + 348
```

Operands could be...

- **VarDecl**
- **Formal**
- **IntegerConst**
- **RealConst**

Need to get them into registers before doing the operation (addition).

Consider generating code for `OPiadd`...

```
        w := y + 348
```

Operands could be...

- VarDecl
- Formal
- IntegerConst
- RealConst

Need to get them into registers before doing the operation (addition).

```
void getIntoAnyReg (Ast.Node p, String reg, String reg2)
```

p	Points to the operand node.
reg	Target register. Examples: "`%o4`", "`%f3`"
reg2	Work register: Must be an integer register. Possibly the same as "reg". Example: "`%o5`"

Operands could be...

- VarDecl *(either local or non-local)*
- Formal *(either local or non-local)*
- IntegerConst
- RealConst

IntegerConst

```
    set   348,%o4
```

RealConst

```
    set   float4,%o5
    ld    [%o5],%f3
```

VarDecl / Formal

```
    ld    [%fp+-8],%o4
```

Local
LexLevel = current level
or:
LexLevel = –1 (temps)

```
    set   display4,%o5
    ld    [%o5],%o5
    ld    [%o5+-12],%o4
```

Non-Local
(Otherwise)

Dealing with the Destination

IR Instruction:

```
w := y + 348
```

The destination will be...

VarDecl *(either local or non-local)*
Formal *(either local or non-local)*

Dealing with the Destination

IR Instruction:

```
w := y + 348
```

The destination will be...

VarDecl *(either local or non-local)*
Formal *(either local or non-local)*

```
void storeFromAnyReg (Ast.Node p, String reg, String reg2)
```

p	**Points to the result, either a VarDecl or Formal.**
reg	**Generate code to move contents of "reg" into memory.**
reg2	**Work register: Must be an integer register. Always different from "reg".**

Example

To translate OPiadd

```
   x := y + z

        inst.op
        inst.arg1
        inst.arg2
        inst.result
```

Example

To translate OPiadd

```
   x := y + z

        inst.op
        inst.arg1
        inst.arg2
        inst.result
```

```
System.out.println ("\tadd\t%o0,%o1,%o1");
```

Tabs before and after op-codes.

Example

To translate OPiadd

```
x := y + z

    inst.op
    inst.arg1
    inst.arg2
    inst.result
```

```
getIntoAnyReg (          ,      ,      );
getIntoAnyReg (          ,      ,      );
System.out.println ("\tadd\t%o0,%o1,%o1");
storeFromAnyReg (          ,      ,      );
```

Tabs before and after op-codes.

Example

To translate OPiadd

```
x := y + z

    inst.op
    inst.arg1
    inst.arg2
    inst.result
```

```
getIntoAnyReg (          , "%o0",      );
getIntoAnyReg (          , "%o1",      );
System.out.println ("\tadd\t%o0,%o1,%o1");
storeFromAnyReg (          , "%o1",      );
```

Tabs before and after op-codes.

Example

To translate OPiadd

```
x := y + z

     inst.op
     inst.arg1
     inst.arg2
     inst.result
```

```
getIntoAnyReg (inst.arg1, "%o0",       );
getIntoAnyReg (inst.arg2, "%o1",       );
System.out.println ("\tadd\t%o0,%o1,%o1");
storeFromAnyReg (inst.result, "%o1",       );
```

Tabs before and after op-codes.

Example

To translate OPiadd

```
x := y + z

     inst.op
     inst.arg1
     inst.arg2
     inst.result
```

```
getIntoAnyReg (inst.arg1, "%o0", "%o0");
getIntoAnyReg (inst.arg2, "%o1", "%o1");
System.out.println ("\tadd\t%o0,%o1,%o1");
storeFromAnyReg (inst.result, "%o1", "%o0");
```

Tabs before and after op-codes.

tst/simple.pcat

```
! MAIN...
    mainEntry


! WRITE STMT...
    writeString "Hello"


    writeNewline


! MAIN EXIT...
    mainExit
```

```
program is
  begin
    write ("Hello");
  end;
```

tst/simple.pcat

```
! MAIN...
!   mainEntry


! WRITE STMT...
!   writeString "Hello"


!   writeNewline


! MAIN EXIT...
!   mainExit
```

```
program is
  begin
    write ("Hello");
  end;
```

tst/simple.pcat

```
program is
  begin
    write ("Hello");
  end;
```

```
! MAIN...
!   mainEntry
          .global  main
main:     save     %sp,-96,%sp
          set      display0,%o0
          st       %fp,[%o0]
! WRITE STMT...
!   writeString "Hello"
          sethi    %hi(str1),%o0
          call     printf
          or       %o0,%lo(str1),%o0
!   writeNewline
          sethi    %hi(strNL),%o0
          call     printf
          or       %o0,%lo(strNL),%o0
! MAIN EXIT...
!   mainExit
          ret
          restore
```

Attack Strategy

```
mainEntry
mainExit
writeString
writeNewline
```
} simple

```
writeInt
writeFloat
writeBool
assign
```
} write

```
label
goto
igotoEQ, etc.
```
} goto1, goto2, goto3

```
iadd,fadd,etc.
itof
```
} binary1, binary2, div, neg, itof

Attack Strategy

$$\left.\begin{array}{l}\texttt{call}\\ \texttt{procEntry}\\ \texttt{returnVoid}\\ \texttt{returnExpr}\end{array}\right\}\text{call1, call2, call3}$$

$$\left.\begin{array}{l}\texttt{param}\\ \texttt{formal}\end{array}\right\}\text{param1, param2, param3}$$

$$\left.\begin{array}{l}\texttt{readInt}\\ \texttt{readFloat}\\ \texttt{loadAddress}\end{array}\right\}\text{read1, read2}$$

$$\left.\begin{array}{l}\texttt{alloc}\\ \texttt{loadIndirect}\\ \texttt{store}\end{array}\right\}\text{alloc1, alloc2}$$

Attack Strategy

$$\left.\begin{array}{l}\texttt{error1}\\ \texttt{error2}\\ \texttt{...}\end{array}\right\}\text{Test runtime error handling}$$

$$\left.\begin{array}{l}\texttt{array1}\\ \texttt{array2}\\ \texttt{array3}\end{array}\right\}\text{Test array index calculation}$$

`for` Test looping code

`local` Test non-local accesses

`semError` Test scripts (no code generated)

$$\left.\begin{array}{l}\texttt{fact}\\ \texttt{primes}\\ \texttt{sort}\\ \texttt{yapp}\end{array}\right\}\text{“Real” application programs}$$

(Don’t run often)

`speed` Benchmark program

Boilerplate Code to Handle Errors

```
!
! runtimeError1-5
!
! Branch to one of these labels to print
! an error message and abort.
!
runtimeError1:
             set      message1,%o0
             call     printf
             nop
             call     exit
             mov      1,%o0
runtimeError2:
             set      message2,%o0
             call     printf
             nop
             call     exit
             mov      1,%o0
```

...etc...

Boilerplate Code to Print Boolean Values

```
! writeBool
!
! This routine is passed an integer in %i0/o0.
! It prints "FALSE" if this integer is 0 and
! "TRUE" otherwise.
!
writeBool:
        save    %sp,-128,%sp
        cmp     %i0,%g0
        be      printFalse
        nop
        set     strTrue,%o0
        ba      printEnd
        nop
printFalse:
        set     strFalse,%o0
printEnd:
        call    printf
        nop
        ret
        restore
```

```
strTrue:    .asciz   "TRUE"
strFalse:   .asciz   "FALSE"
```

Boilerplate Code to Print Boolean Values

```
! writeFlt
!
! This routine is passed a single precision
! floating number in %f0.  It prints it by calling
! printf.  It uses registers %f0, %f1.
!
writeFlt:
        save    %sp,-128,%sp
        fstod   %f0,%f0
        set     temp,%l0
        std     %f0,[%l0]
        ldd     [%l0],%o0
        mov     %o1,%o2
        mov     %o0,%o1
        set     strFlt,%o0
        call    printf
        nop
        ret
        restore
```

```
strFlt:    .asciz    "%g"
temp:      .double   0
```