Midterm exam:

Good job!

Generally, results displayed a good mastery of concepts.

Most common problems:

- expressions are not allowed at the top-level in programs (problem 6.a)

- sentences may have more than one derivation in an unambiguous grammar (problem 5)

Any questions?
Next phase of compiler construction: IR transformation

IR is bridge between:
- Parse trees
  - language specific
  - high level
- Machine code
  - architecture specific
  - low level

IR is more general than either
- Can describe many languages
- Can describe many architectures

IR is intermediate level
- Lower level constructs (closer to machine level)
- Many specifics abstracted

Like parse trees,
- IR made of \textit{exprs} and \textit{stmts}

Control structures translated into labeled \textit{jumps}
- Most other \textit{stmts} \textit{moves}
  - 3-operand form typical
  - target = left op right
- Also have function calls
IR statements:

  MOVE expr,expr
  JUMP expr,<const,label> list,label
  CJUMP expr,true label,false label
  LABEL
  CALL expr,(expr)
  SEQ stmt,stmt
  ESEQ stmt, expr

IR exps have no side-effects

Each operand becomes either:
  memory (can have single +/-)
  register

Only integer constants in IR
 Chars, integers, pointers

IR transformation “flattens” expressions

Turns complex expressions into list of simple IR expressions

For example,
  a = b + ((c=3)*d)
becomes
  move c,3
  move t1,c,*,d
  move a,b,+,t1
Also need to add casts in place of type annotations

Casts not in source language, but exists as parseExpr in compiler

example
    char c; int i;
    c += i;
really means
    c = (char)((int)c + i);

In general, assignment needs to add casts

Cast rhs of assignment to type of lhs if different
Actually build CastExpr
    Translation then performed on CastExpr

Side effects in assignment complicate translation

    int * f(int);
    *(f(x)) += 3;

This must call f only once
Need to create temp for result

    t1 = f(x);
    *t1 = *t1 + 3;
most translation straightforward expressions
  most ops translate directly
  need to make casts explicit
  assignment ops and ++,- trickier
  calls, assignments become stmts
  ESEQ for these
  stmt then expr
  used for these cases

Statement translation mostly control
2 forms of jumps
Conditional (cjump)
  Has boolean expression
  True label
  False label
Computed (jump; like switch)
  Has value expression
  Has list of paired constant value,label
  Has default label

Computed jump (cjjump):
  Computes value
  Compares to each value in list
  If any match, jump to label
  Otherwise jump to default
Consider:

```java
switch (expr) {
    case v1 : s1; ... ; case vn : sn; default : s
}
```

Translation:

```java
L:
    cjump <expr>, 
    <v1>=L1; ... ; <vn>=Ln,D
L1:
    <s1>
    ...
Ln:
    <sn>
D:
    <s>
```

What if case expressions aren't values?

Structural analysis requires cases to be expressions involving constants

*Constant folding* evaluates such expressions to values at compile time

`ParseExpr.valueOf()` does constant folding (returns null if expr not foldable)

Consider translating

```java
while (expr) stmt
```

Translation becomes:

```java
L1:
    cjump <expr>,L2,L3
L2:
    <stmt>
    jump 1,{},L1
L3:
```

```java
C S 202-19
computed jumps
16```

```java
C S 202-19
Constant folding
17```

```java
C S 202-19
translating while
18```
consider for(init; cond; incr)body
translates into

init
top_label
body
cont_label
incr
cjump cond top_label break_label
break_label

display statements as list
actually implemented as single tree
uses seq statement

seq pairs 2 statements as 1
either child can be seq
eventually move to left child only

s1
s2
s3
becomes

seq

seq
s3

s1
s2
translate on FunctionDef
produces entry label
translates body
produces exit label

label for exit generated early on
details of entry/exit filled in later

may need to use temp
\[ f(x) = 3 \]
becomes
\[ \text{ESEQ} \left( \text{SEQ} \left( \text{MOVE} \ t, f(x), \text{MOVE} \ \text{MEM}(t), 3 \right); \ \text{MEM}(t) \right) \]

but what about the \( f(x) \)
also becomes eseq
\( f(x) \) becomes call stmt
what is return
special return register
maps to return on target machine
\( f(x) \) becomes
\[ \text{ESEQ} \left( \text{CALL}, f(x), \text{REG} \right) \]
want translation target independent
what about this return register
2 choices
can put independent place holder
can include some machine dependence
will do latter
dependence all defined in interface
main routine instantiates appropriate
all specifics handled there

---

translating storage expressions
2 real forms of storage
MEM
REG
don’t know yet where vars will be
TEMP may bind to either
temp param subclass of temp
Register allocation will bind temps
(assignment 11)

---

declarators hold expression
generally temp
translate returns this expression
param declarators usually param temp
temp hopefully become REG
may become MEM (+fp.offset))
variables marked if & applied
already return MEM(+fp.offset))

---
Array access becomes (informally):
MEM(+(&arr,"(index,sizeof(element))")

Use Type.typeSize() method to determine sizeof elements

Actually build + and * IRBinExprs

consider a == b
what is translation?
what does it mean?
   depends on context
x = (a == b) sets x to 0 or 1
if (a==b) changes control flow
how to represent both?

actually 2 functions on parseExpr
   translate returns exprs so far
   translateBoolean returns conditional branch

translateBoolean takes 2 args
   one describes true label
   one describes false label
usually,
call translateBoolean on boolean expr
call translate on value expr
sometimes do opposite
\[ x = (a = b) \]
if \( b \)
value expression as boolean easier
turn into $x \neq 0$
translateBoolean that

comparisons generate simple booleans
not swaps true, false labels
what about $\&\&$, $||$
need to combine booleans

consider $a==b \&\& c==d$

becomes

```
CJUMP a==b,L1,F
LABEL L1
CJUMP c==d,T,F
```

where $T$ is eventual true label
$F$ is eventual false label
overview of processing
- call generateIR on each Declarator
- allocate storage for vars
- translate functions
  - all code translation from here

translate on statement
takes 3 args
- exit_label where return branches allocated by FunctionDef
- cont_label where continue branches may be null, from loop
- break_label where break branches may be null from enclosing switch/loop

ForStmt.translate(X,C,B)
- `top, cont, break = new IRLabel()`
- `result = SEQ(init.translateEffect(),top);`
- `result = SEQ(result, body.translate(X,cont, break))`
- `result = SEQ(result,cont)`
- `result = SEQ(result,incr. translateEffect())`
- `result = SEQ(result, cond.translateBoolean(top, break))`
- `result = SEQ(result, break)`
translateBoolean
   takes 2 labels
   generates CJUMP
   maybe more, with SEQ

translateEffect
   generates just the side effects
   remember translate of assign
   ESEQ(Move x,y; x)
   sometimes want just Move
   don’t care about value of it
   for loop init, incr
   Expr statement

translateEffect simple
   if side effect directly
      translate returns ESEQ
      return stmt part only
   if no side effect directly
      call translateEffect on children, if any
      combine answers
   if no children
      return IRNop
Assignment 7:

implement IR translate method in
  WhileStmt, IfStmt, BreakStmt, ForStmt
  ArrayExpr
finish IR translation on
  BinExpr, UnaryExpr

Look for COMPLETE ME
Use predefined translations as examples

Warning: codebase designed for implementing Lake

Extraneous details wrt Pond

Don’t worry about extraneous details
  (anything not in Pond spec., other than cast)

Running lakec.Lake:
  -ir now dumps IR
  -p stops after parsing

Will stop after parsing if any errors

Due Tuesday after break at midnight.