Intermediate Code

source

scanner

parser

sem. anal.

IC

opt.

reg. alloc.

code gen.

assembly

front end

back end
Intermediate Code

- language independent
- machine independent
- low level

Goal:

Java
ML
Pascal
C
C++
Tiger
IR
Sparc
MIPS
Pentium
Alpha
Intermediate Representations

- expression trees
- quadruples
  \[ t_1 = t_2 + t_3 \]
- \( \lambda \)-calculus
- spineless tagless \( \lambda \) machine
- Java Byte Code
abstract class Exp

CONST (int value)
NAME (Label label)
TEMP (Temp.Temp temp)
BINOP (int binop, Exp left, Exp right)
MEM (Exp exp)
CALL (Exp func, ExpList args)
ESEQ (Stm stmt, Exp exp)
Expression Trees

abstract class Stmt

MOVE (Exp dst, Exp src)

EXP (Exp exp)

JUMP (Exp exp, Temp, Label, List targets)

CJUMP (int relop, Exp left, Exp right, Label iftrue, Label iffalse)

SEQ (Stmt left, Stmt right)

LABEL (Label label)

UEXP (Exp exp) for PC
Examples on stack frame

```
MEM
  |
  BINOP
    /
   PLUS TEMP fp CONST k

MEM(BINOP(PLUS, TEMP fp, CONST k))
```

Abbrev.

```
MEM
  +
    |
    TEMP fp
    |
    CONST k

MEM+(TEMP fp, CONST k))
```
Notes

MEM(e) ... fetch or store
ESEQ(s,e) ... eval s, compute e
MOVE(TEMP t, e)
MOVE(MEM(e1), e2)

EXP(e) ... discard result

JUMP(c, labs)

JUMP(c)

= JUMP(NAME l, new label(list(l, nil))

-5 = 0-5
not b = 1 xor b
Interface with Semant

each method in Semant, Semant produces pair

(Translate, Exp, Types, Type)

- in Translate build trees
- in Semant, Semant call tree-building functions
- tree/* ... tree data types
Kinds of Expressions

\[ x := e + 5; \]

\[ x := 5; e; y := 42; \]

if \( e \) then 1 else 2

- IC for \( e \) depends on use

- Tree, Exp
- Tree, Stm
- Tree, CJUMP(\( e, y, t, f \))
package Translate;

public abstract class Exp {
    abstract Tree.Exp unEx ();
    abstract Tree.Stm unNx ();
    abstract Tree.Stm unCx (Temp.Label +
                          Temp.Label f);
}

Ex    ... expression
Nx    ... no result
Cx    ... conditional
Example

\[ a > b | c < d \]

we produce a `Translate. Exp` whose `uncx` method is

```java
Tree.Stm uncx (Label t, Label f) {
    Label z = new Label();
    return new SEQ(new CJUMP(CJUMP.GT, a, b, t, z),
                    new SEQ(new Label(z),
                            new CJUMP(CJUMP.LT, c, d, t, f)));
}
```
class Exp extends Exp {
    Tree.Exp exp;
    Exp (Tree.Exp e) {exp = e;}
    Tree.Exp unExp() {return exp;}
    Tree.Exp unNxt() {
        return new Tree.EXP (exp);
    }
    Tree.Exp unCxt(Label t, Label f) {
        ...?
    }
}
Translate.Exp

class Nx extends Exp {
    Tree, Stmt stmt;
    Nx (Tree, Stmt s) { stmt = s; }
    Tree, Exp unExpr() {  ....?.... }
    Tree, Stmt unNx() { return stmt; }
    Tree, Stmt unCx(Label t, Label f) 
    {  ....?.... }
}

abstract class Cx extends Exp {
    Tree, Exp onEx () {
        Temp r = new Temp(0);
        Label t = new Label(1);
        Label f = new Label(1);

        return new Tree.ESEQ ( 
            new Tree.SEQ (new Tree.MOVE ( 
                new Tree.TEMP(r),
                new Tree.CONST(1)) ,
            new Tree.SEQ (un Cx(t,f),
            new Tree.SEQ (new Tree.LABEL(f),
            new Tree.SEQ ( 
                new Tree.MOVE ( 
                    new Tree.TEMP(r),
                    new Tree.CONST(0)) ,
            new Tree.LABEL(t))),
            new Tree.TEMP(r))) ;

    }
}
Translate Exp

class Rel(Cx extends Cx {...})

class IfThenElseExp extends Exp {...}
Simple Variables

- Variable \( x \) declared in current frame

\[
\text{MEM } (+ \text{(TEMP } \text{fp, CONST } k))
\]

- Following static links

\[
\begin{align*}
\text{MEM } (+ \text{(CONST } k_n, \\
\text{MEM } (+ \text{(CONST } k_{n-1}, \\
\vdots \\
\text{MEM } (+ \text{(CONST } k_1, \text{TEMP } \text{fp})) \\
\}}))
\end{align*}
\]

\[
+ \text{(CONST } k_n)
\]
Simple Variables

- Use function
  
  ```
  public Exp simpleVar (Access, Level)
  ```

  in Translate.

- Semant passes
  
  - access of x (from Translate.alloc)
  
  - level in which x is used

- Translate uses method `exp()`

  from class Frame, Access, e.g.,

  ```
  a.exp (new TEMP (frame.FP(1))) =
  ```

  ```
  MEM (BINOP (PLUS,
           TEMP (frame.FP(1)),
           CONST (k)))
  ```
Array Variables

**Pascal:**

```pascal
var a, b : array [1..12] of integer
begin
  a := b
end
copies array
```

**C:**

```c
#define a[12], b[12]

a = b;
?
illegal — no array variables, only array constants and pointers variables:

#define a[12], *b

b = a;
?
```
Array Variables

Tiger:

let

type intArray = array of int
var a := intArray[12] of 0
var b := intArray[12] of 7
in
  a := b
end

- a and b point to same array
- same semantics as Java and ML
Structured L-Values

L-value: LHS of assignment
R-value: everywhere else

C: structs as L-values
Pascal: arrays and records

Requires MEM operator to be extended with size:

MEM(Exp exp, int size)

Tiger has only scalar L-values
Subscripting, Field Selection

Pascal:

\[ \text{L-value should not be (e.g., } a[i]) \]

\[ \text{MEM} \]

\[ \text{+} \]

\[ \text{TEMP fp} \quad \text{CONST k} \]

but only

\[ \text{+} \]

\[ \text{TEMP fp} \quad \text{CONST k} \]

Might need to add offset/index.
Might need to coerce to R-value.
Subscripting

Pascal

If a is in MEM\(e\), then \(a[i][j]\) is

```
MEM
  /
 +
/   \
MEM  *
  /
 e     i
  /
    \
  CONST w
```
L-values and MEM nodes

Tiger:

We get away with L-values being MEM nodes.

- only scalar L-values
- MEM is fetch/store
String Rep

Pascal: fixed size

C: null-terminated

Tiger: length + data

- call run-time function stringEqual
- lay out constants
**Conditionals**

- Comparisons translated into CJUMP
  
  E.g.; \( x < 5 \) is translated as
  
  \( Cx(5) \) where
  
  \( s_i(t,f) = CJUMP(LT,x,\text{CONST}(5),t,f) \)

- Use class RelCx.

- 8 and I have been translated into if-expressions in the abstract syntax

- if-then-else is missing
IfThenElseExp

Given: if e1 then e2 else e3

- Treat e1 as a Cx expression.
- Treating e2 and e3 simply as Ex expressions is inefficient if the if-expression is a boolean exp

Therefore, we need IfThenElseExp with appropriate unEx, unNt, unCx methods.
Example

if \( x < 5 \) then \( a > b \) else 0

should result in something like
class IfThenElseExp extends Exp {
    Exp cond, a, b;
    Label t = new Label();
    Label f = new Label();
    Label join = new Label();
    IfThenElseExp (Exp cc, Exp aa, Exp bb)
        cond = cc; a = aa; b = bb;
    }

Tree.Stm unCx (Label tt, Label ff);
Tree.Exp unEx () {...}
Tree.Stm unNx () {...}
}
Record & Array Creation

\[ \{(f_1 = e_1, f_2 = e_2, \ldots, f_n = e_n)\} \]

should result in (p. 172)
Record & Array Creation

For arrays, simply call initArray.

- malloc, initArray, stringEqual
- are C or ASM functions
- don't have static links
- are OS-specific
  (C compiler may add underscore)

- Use Frame.externalCall()
Memory Allocation

- we use malloc
- we don't deallocate
- Garbage Collector added in lab 9

GC could be either

- a mark-and-sweep
- or a copying collector.
While Loops

while cond do body
is translated to

test:
  if not(cond) goto done
body
  goto test
done:

Label done is needed for breaks inside body.

Better:

goto test
start:
  body
test:
  if cond goto start
done:

Can be patched up later (chap. 8,
For Loops

Given: for i := lo to hi do body

Don't use:

```
let var i := lo
  var limit := hi
in
  while i <= limit
    do (body; i := i + 1)
end
```

Better:

```
let var i := lo
  var limit := hi
in
  if lo <= hi then
    while i
      do (body;
          if i >= limit then break
          i := i + 1)
  end
```
Function Definition

1. announce function
2. label
3. adjust sp (allocate frame)
4. save/move arguments
5. save callee-save registers
6. body
7. move result into RV
8. load callee-save registers
9. reset sp (deallocate frame)
10. return
11. announce end of function

Frame, proc entry, exit
Fragments

Data fragments (string constants)

Procedure fragments
  - frame
  - body (result of procEntryExit)