Intermediate Representations

• Machine independent translation
  – Keep independent of target architecture for as long as possible

• Intermediate forms
  – 3 address code (triples, quads)
  – Expression trees

• Some examples of Tiger intermediate code
Why an intermediate representation?

• Need representation closer to actual instructions for ease of translation

• Compiler
  – *Front end* does lexical analysis, parsing, semantic analysis, translation to IR
  – *Back end* does optimization of IR, translation to machine instructions

• Try to keep machine dependences out of IR for as long as possible
How IR is useful?

Java
ML
Pascal
C
C++

IR

Sparc
MIPS
Pentium
Alpha
SPIM code
Three Address Code

• *Result, operand, operand, operator*
  
  – \( x := y \ \text{op} \ z \), where \( \text{op} \) is a binary operator and \( x, y, z \) can be variables, constants or compiler-generated temporaries (intermediate results)

• Can write this as a shorthand
  
  – \( <\text{op}, \text{arg1}, \text{arg2}, \text{result}> \) -- quadruples
  
  – Let line number of instruction stand for the result
    
    • \( <\text{op}, \text{arg1}, \text{arg2}> \) -- triples (saves space)
Three Address Code

• Set of statements allowed
  – Assignment  \( x := y \ op \ z \)
  – Copy stmts  \( x := y \)
  – Goto L
  – if \( x \) relop \( y \) goto L
  – Indexed assignments  \( x := y[j] \) or \( s[j] := z \)
  – Address and pointer assignments (for C)
    \( x := &y, x := *p; \ *x := y \)
  – Parm \( x \); call \( p, n; \) return \( y \); (for calls)
How this works?

\[ z := 2 == 3 \text{ or } 3 == 4; \]

\[ t_1 := 2 == 3 \]
\[ t_2 := 3 == 4 \]
\[ t_3 := t_1 \text{ or } t_2 \]
\[ z := t_3 \]

Production rules:

\[ S ::= id := E \]
\[ E ::= T \ R \]
\[ R ::= or \ T \ R | \epsilon \]
\[ T ::= D \ Relop \ D \]

Example derivation:

\[ S \rightarrow id := E \]
\[ E \rightarrow T \ R \]
\[ R \rightarrow or \ T \ R | \epsilon \]
\[ T \rightarrow D \ Relop \ D \]
\[ D \rightarrow 2 \]
\[ D \rightarrow 3 \]
\[ Relop \rightarrow == \]
\[ R \rightarrow or \]
\[ T \rightarrow D \]
\[ D \rightarrow 3 \]
\[ D \rightarrow 4 \]
\[ Relop \rightarrow == \]
Expression Trees (Tiger)

- Simple intermediate representation (IR)
- Convenient to translate into actual machine instructions for several target machines
- Convenient to produce from abstract syntax
- Each construct must have a clear meaning
- Take “big” pieces of abstract syntax and translate them into many small pieces of abstract machine instructions
Tiger IR - Exp’s

- \textit{Const}(j), integer constant j
- \textit{Name}(n), symbolic constant n (to correspond to assembly language label)
- \textit{Temp}(t), temporary t (unlimited in number)
- \textit{Binop}(o, e1,e2), application of binary operator o to operands e1 and e2. (Appel, p 157)
- \textit{Mem}(e), contents of \texttt{wordSize} bytes of memory
Tiger IR- Exp’s

- \textit{Call}(f,l), application of function \(f\) to argument list \(l\)
- \textit{Eseq}(s,e) statement \(s\) evaluated for side effects and then \(e\) is evaluated for a result
Tiger IR - Stm’s

- **Move**(Temp t, e), evaluate e and move it into t
- **Move**(Mem(e1,k),e2), evaluate e1 yielding address a. then evaluate e2 and store result into k bytes of memory starting at a.
- **Exp**(e), evaluate e and discard result
- **Jump**(e,labs), transfer control to address e; labs tells all possible locations that e can evaluate to
Tiger IR - Stm’s

- **Cjump**(o, e1, e2, t, f), evaluate e1, then e2, yielding values of a, b. now compare a to b using relational operator o. if result is true, jump to t, else jump to f.
- **Seq**(s1, s2), statement s1 follows statement s2
- **Label**(n), define the constant value of name n to be the current machine code address
- No abstract instructions for procedure entry or exit
Project Organization: Packages

Parse

Absyn

Symbol

Semant

Types

Translate

Temp

Util

Tree

Frame

Semantic Analysis

SPIM

Code Generation

Parsing and Lexing
Package Absyn

• Class Exp

• Class Dec
  – TypeDec, VarDec

• Class Ty
  – ArrayTy, NameTy, RecordTy

• Class FieldList

• Class FieldExpList
Package Tree

- Class Exp
  - BINOP, CALL, CONST, ESEQ, MEM, NAME, TEMP
- Class Stm
  - SEQ, EXP, JUMP, CJUMP, LABEL, MOVE
- ExpList
- StmList
- Print

Essentially, the job of assignment 5 is to translate Absyn trees to sequences of Tree trees.
IR vs AST (Appel, p 103, 157)

package Tree
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 stm, Exp exp)

package Absyn
abstract class Exp
IntExp(int pos, int value)
OpExp(int pos, Exp left, int oper, Exp right)
VarExp(int pos, Var var)
CallExp(int pos, Symbol func, Explist args)
Translating Expressions

- Translation discussed in Chapter 7 talks about 3 sorts of expressions, pp 159ff:
  - **unExp** - a single valued expression
  - **unNx** - an expression that is a statement (yields no value)
  - **unCx** - a conditional expression (that jumps to t or f)

```plaintext
package Translate: abstract class Exp
  class Ex  class Nx  abstract class Cx
```
Translating Expressions

• Use simpler translation scheme (Appel, p 178)
• Translate all expressions as values
  – Can think of translating these using one Translate.Exp class without subclasses with a member Tree.Exp and only an unEx() method
    • unExp translated as usual to an Ex (an expression returning a value)
    • an Nx(s) is translated as Ex(ESEQ(s, CONST(0)))
    • For conditionals, use a value expression that evaluates to 0(false) or 1(true) --more later
ExpTy transExp(Absyn.Exp e)

• Use as a dispatcher method which calls other methods particularized to translation of a specific expression’s Absyn AST
  – In class Translate.Exp, Exp(Tree.Exp) encapsulates its parameter object in a Translate.Exp object which can be stored in the first field of an ExpTy object (the field we had been leaving null in assignment 4)
  – Every ExpTy object has one each Translate.Exp, Types.Type members for intermediate code and type respectively
What about conditional exprs?

/* if input is Absyn.IfExpr with else clause */
1. generate 3 jump labels - one for true, other for false, one for end of if
2. generate a temp to hold the numerical result
3. create a code sequence which first calculates the test as a value expression, compares its value versus 0 and then jumps to true label or false label.
4. associate the true label with the code for the then clause and jump to end of if
5. associate the false label with the code for the else clause and fall through to end of if.

/* you will probably use JUMP, ESEQ, CONST, CJUMP from Tree package in combination to translate an IfExpr*/
Memory

• Any intermediate instructions that contain simple variables will involve accesses to memory through a frame
  – *Frame* interface to be provided
  – Access should be frame_pointer+offset through a Mem instruction

• Array elements will be addressed as base_address + subscript*elementsize
  – base_address is contents of Frame element corresponding to the array
  – elementsize is assumed same for all data
  – subscript is calculated as value of a temp
Memory

- Simple variable (in frame)

- Array element (memory-resident array variable $a[j]$)
Translations

• Arithmatic expressions
  – Unary negation implemented as subtraction from 0
  – Can be translated at first using just CONSTs as operands to check out transExp() driver and a small subset of translator routines
Translations - Loops

• While code

  test:  if not(condition) goto done

  body

  goto test

done:

  – Have to identify break statements with the done label for the closest enclosing loop

  – But this has to happen during transExp() recursive processing of a program expression
Translators - Loops

- For loop - most easily translated by considering as a form of while

  for \( j \) := \( lo \) to \( hi \) do body  \( \) becomes

  let var \( j \) := \( lo \)
  var limit := \( hi \)
  in while \( j \) <= limit
    do (body ; j := j+1)
  end
Translations - Declarations

• Variables
  – Must reserve space for variables on frame
  – May need to emit code for initializations using assignments
Fragments

• Overall program expression is translated into a list of Fragment objects, one per function
  – Also translate String literal pool as a Fragment

• Necessary Fragments package to be supplied