But first, a Whitespace program

Example

Example class Fac {
    public int ComputeFac(int num) {
        int numAux;
        if (num < 1) {
            numAux = 1;
        } else {
            numAux = num * this.ComputeFac (num-1);
        }
        return numAux;
    }
}

Interpretation tradeoffs: reprise

• Simple conceptually, easy to implement
  – fast turnaround time
  – good programming environments
  – easy to support fancy language features
• Slow to execute
  – data structure for value vs. direct value
  – variable lookup vs. registers or direct access
  – EVAL overhead vs. direct machine instructions
  – no optimizations across AST nodes

Compile-time Processing

• Decide representation of run-time data values
• Decide where data will be stored
  – registers
  – format of stack frames
  – global memory
  – format of in-memory data structures (e.g. records, arrays)
• Generate machine code to do basic operations
  – just like interpreting expression, except generate code that will evaluate it later
• Do optimizations across instructions if desired
### Compile-time vs Run-time

<table>
<thead>
<tr>
<th>Compile-time</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure</td>
<td>Activation record/stack frame</td>
</tr>
<tr>
<td>Scope, symbol table</td>
<td>Environment (contents of stack frame)</td>
</tr>
<tr>
<td>Variable</td>
<td>Memory location or register</td>
</tr>
<tr>
<td>Lexically-enclosing scope</td>
<td>Static link</td>
</tr>
<tr>
<td>Calling procedure</td>
<td>Dynamic link</td>
</tr>
</tbody>
</table>

### Compilation Plan

- Translate ASTs into linear sequence of simple statements called intermediate code (IL or IR)
  - Source-language, target-language independent
- Translate IL into target code
- Intermediate code generation focuses on simple representations of source constructs
- Target code generation focuses on constraints of particular target machines
- Different front ends and back ends can share IL
- IL can be optimized independently of each

### MiniJava’s Intermediate Language

- Want intermediate language to have simple, explicit operations (humans don’t write IL programs)
- Use simple declaration primitives
  - global functions, global variables
  - no classes, no implicit method lookup, no nesting
- Use simple data types
  - ints, doubles, explicit pointers, records, arrays
  - no boolean
  - no class types, no implicit class fields
  - arrays are naked sequences; no implicit length or bounds checks
- Use explicit gotos instead of control structures
- Make all implicit checks explicit (e.g. array bounds checks)
- Implement method lookup via explicit data structures and code

### MiniJava’s IL (1)

**Program** ::= `{GlobalVarDecl} {FunDecl}

**GlobalVarDecl** ::= Type ID \[=\] Value ;

**Type** ::= int | double | *\[\] Type

**Value** ::= Int | Double | &ID

**FunDecl** ::= Type ID \(\{\text{Type ID}/,\}\) {VarDecl} {Stmt}

**VarDecl** ::= Type ID ;

**Stmt** ::= Expr ;

<table>
<thead>
<tr>
<th>Unop</th>
<th>Binop</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>-int</td>
<td>-double</td>
<td>Expr</td>
</tr>
<tr>
<td>not</td>
<td>int2double</td>
<td>Expr</td>
</tr>
<tr>
<td>&lt;(\leq)(\geq)&gt;(=)!|=(\neq) . (\text{int/\text{double}})</td>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>(\ll\text{unsigned})</td>
<td>ID ({\text{Expr}})</td>
<td>String</td>
</tr>
</tbody>
</table>

### MiniJava’s IL (2)

**Expr** ::= LHSExpr | Unop Expr

| Expr Binop Expr |
| Callee \(\{\text{Expr}\}/,\) |
| new Type \(\{\text{Expr}\}\) |
| Int | Double | & ID

**LHSExpr** ::= ID | * Expr

| Expr \(\rightarrow\) ID \(\{\text{Expr}\}\) |

**Unop** ::= -int | -double | not | int2double

**Binop** ::= \(+\)\|\(-\)\|\(*\)\|\(/\)

| ID \(\{\text{Expr}\}\) \(\{\text{Expr}\}\) | \(\text{int/\text{double}}\) |
| \(\ll\text{unsigned}\) | Expr |

**Callee** ::= ID \(\{\text{Expr}\}\) | String

### MiniJava’s IL Classes (1 of 6)

**ILProgram** : {ILGlobalVarDecl} {ILFunDecl}

**ILGlobalVarDecl** : ILType String

| ILInitializedGlobalVarDecl : ILValue

**ILType**

| ILIntType
| ILDoubleType
| ILPtrType : ILType
| ILSequenceType : ILType
| ILRecordType : ILType String
| ILCodeType
MiniJava's IL Classes (2 of 6)

- **ILValue**
  - **ILIntValue**: int
  - **ILDoubleValue**: double
  - **ILGlobalAddressValue**: ILGlobalVar
  - **ILLabelAddressValue**: ILLabel
  - **ILSequenceValue**: {ILValue}
  - **ILRecordValue**: {ILValue String}

- **ILFunDecl**: ILType String {ILFormalVarDecl} {ILStmt}
- **ILVarDecl**: ILType String

MiniJava's IL Classes (3 of 6)

- **ILStmt**
  - **ILExprStmt**: ILExpr
  - **ILAssignStmt**: ILAssignableExpr
  - **ILConditionalBranchStmt**: ILExpr ILLabel
  - **ILConditionalBranchFalseStmt**: ILLabel
  - **ILConditionalBranchTrueStmt**: ILLabel
  - **ILGotoStmt**: ILLabel
  - **ILLabelStmt**: ILLabel
  - **ILThrowExceptionStmt**: String
  - **ILReturnStmt**: ILExpr

- **ILLabel**: String
- **ILGlobalVar**: String

MiniJava's IL Classes (4 of 6)

- **ILVar**: ILVarDecl

MiniJava's IL Classes (5 of 6)

- **ILExpr**
  - **ILAssignableExpr**
  - **ILVarExpr**: ILVar
  - **ILPtrAccessExpr**: ILExpr
  - **ILFieldAccessExpr**: ILExpr ILType String
  - **ILSequenceFieldAccessExpr**: ILExpr
  - **ILUnopExpr**: ILExpr
    - **INegativeExpr**
    - **ILLogicalNegateExpr**
    - **ILIntToDoubleExpr**

- **ILBinopExpr**: ILExpr ILExpr
  - **IL{Add,Sub,Mul,Div,Equal,NotEqual,LessThan,LessThanOrEqual,GreaterThanOrEqual,GreaterThan}**
  - **ILUnsignedLessThanExpr**

- **ILAllocateExpr**: ILType
  - **ILAllocateSequenceExpr**: ILExpr

- **ILIntConstantExpr**: int
- **ILDoubleConstantExpr**: double

- **ILGlobalAddressExpr**: ILGlobalVar

MiniJava's IL Classes (6 of 6)

- **ILGlobalAddressExpr**: ILGlobalVar
- **ILFunCallExpr**: ILType {ILExpr}
  - **ILDirectFunCallExpr**: String
  - **ILIndirectFunCallExpr**: ILExpr
  - **ILRuntimeCallExpr**: String

Intermediate Code Generation

- Choose representations for source-level data types
  - translate each ResolvedType into ILType(s)
- Recursively traverse ASTs, creating corresponding IL pgm – parallels typechecking and evaluation traversals
  - **Expr** ASTs create ILExpr ASTs
  - **Stmt** ASTs create ILStmt ASTs
  - **MethodDecl** ASTs create ILFunDecl ASTs
  - **ClassDecl** ASTs create ILGlobalVarDecl ASTs
  - ...
Run-time storage layout

- Representation of
  - int, bool, etc.
  - arrays, records, etc.
  - procedures
- Placement of
  - global variables
  - local variables
  - parameters
  - results

Data layout of scalars
Based on machine representation

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Use hardware representation</td>
</tr>
<tr>
<td></td>
<td>(2, 4, and/or 8 bytes of memory, maybe aligned)</td>
</tr>
<tr>
<td>Bool</td>
<td>1 byte or word</td>
</tr>
<tr>
<td>Char</td>
<td>1-2 bytes or word</td>
</tr>
<tr>
<td>Pointer</td>
<td>Use hardware representation</td>
</tr>
<tr>
<td></td>
<td>(2, 4, or 8 bytes, maybe two words if segmented machine)</td>
</tr>
</tbody>
</table>

Data layout of aggregates:
records, arrays, etc.

- Aggregate scalars together
- Different compilers make different decisions
- Decisions are sometimes machine dependent

Layout of records

- Concatenate layout of fields
- Respect alignment restrictions
- Respect field order, if required by language
  - Why might a language choose to do this or not do this?
  - Respect contiguity?

Layout of arrays

- Repeated layout of element type
  - Respect alignment of element type
- How is the length of the array handled?

Layout of multi-dimensional arrays

- Recursively apply layout rule to subarray first
- This leads to row-major layout
- Alternative: column-major layout
  - Most famous example: FORTRAN

s : array [5] of record;
  i : int;
  c : char;
  end;

r : record
  b : bool;
  i : int;
  m : record
    b : bool;
    c : char;
  end;

j : int;
  end;

  i : int;
  c : char;
  end;

a[1][1] a[1][1]
a[1][2] a[2][1]
a[2][1] a[3][1]
a[2][2] a[2][1]
a[3][1] a[2][2]
a[3][2] a[3][2]
Array Layout: which is better?

\[ a: \text{array} \{1000, 2000\} \text{ of int}; \]

\[ \text{for } i := 1 \text{ to } 1000 \text{ do} \]
\[ \text{for } j := 1 \text{ to } 2000 \text{ do} \]
\[ a[i, j] := 0; \]

\[ \text{for } j := 1 \text{ to } 2000 \text{ do} \]
\[ \text{for } i := 1 \text{ to } 1000 \text{ do} \]
\[ a[i, j] := 0; \]

Dynamically sized arrays

- Arrays whose length is determined at run-time
  - Different values of the same array type can have different lengths
- Can store length implicitly in array
  - Where? How much space?
- Dynamically sized arrays require pointer indirection
  - Each variable must have fixed, statically known size

Dope vectors

- PL/1 handled arrays differently, in particular storage of the length
- It used something called a dope vector, which was a record consisting of
  - A pointer to the array
  - The length of the array
  - Subscript bounds for each dimension
- Arrays could change locations in memory and size quite easily

String representation

- A string is an array of characters
  - So, can use array layout rule for strings
- Pascal, C strings: statically determined length
  - Layout like array with statically determined length
- Other languages: strings have dynamically determined length
  - Layout like array with dynamically determined length
  - Alternative: special end-of-string char (e.g., \0)

Storage allocation strategies

- Given layout of data structure, where in memory to allocate space for each instance?
- Key issue: what is the lifetime (dynamic extent) of a variable/data structure?
  - Whole execution of program (e.g., global variables)
    - Static allocation
  - Execution of a procedure activation (e.g., locals)
    - Stack allocation
  - Variable (dynamically allocated data)
    - Heap allocation