The Backend (continued)

David Notkin
Autumn 2008

But first, a Whitespace program

An example of a program that counts from 1 to 10, outputting the current value as it goes.

Sorry, but it doesn't have any comments (although it could).

[Space][Space][Space][Tab][LF]	
[LF][Space][Space][Space][Tab][Space][Space] [Space][Space][Tab][Tab][LF]	Set a Label at this point
[Space][LF][Space]	Duplicate the top stack item
[Tab][LF][Space][Tab]	Output the current value
[Space][Space][Space][Tab][Space][Tab][Space][LF]	Put 10 (newline) on the stack...
[Tab][LF][Space][Space]	...and output the newline
[Space][Space][Space][Tab][LF]	Put a 1 on the stack
[Tab][Space][Space][Space]	Addition. This increments our current value.
[Space][LF][Space]	Duplicate that value so we can test it
[Space][Space][Space][Tab][Space][Tab][Tab][LF]	Push 11 onto the stack
[Tab][Space][Space][Tab]	Subtraction. So if we've reached the end, we have a zero on the stack.
[LF][Tab][Space][Space][Tab][Space][Space][Space][Tab][Space][Tab][LF]	Put a 1 on the stack
[LF][Space][LF][Space][Tab][Space][Space][Space][Space][Tab][Tab][LF]	Jump to the start
[LF][Space][Space][Space][Tab][Space][Space][Space][Tab][Space][Tab][LF]	Set the end label
[Space][LF][LF]	Discard our accumulator, to be tidy
[LF][LF][LF]	Finish

Example

- Activation records
- Environment
- Dynamic link
- Static link

```
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

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

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 booleans
  – 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
              | Type [] | { {Type ID}/, } | fun
Value ::= Int | Double | &ID
              | [ {Value}/, ] | { {ID = Value}/, }
FunDecl ::= Type ID ( {Type ID}/,)
              { {VarDecl} {Stmt} }
VarDecl ::= Type ID ;
Stmt ::= Expr ; | LHSExpr = Expr ;
          | iffalse Expr goto Label ;
          | iftrue Expr goto Label ;
          | goto Label ; | label Label ;
          | throw new Exception( String ) ;
          | return Expr ;
```


MiniJava's IL (2)

```
Expr ::= LHSExpr    | Unop Expr
        | Expr Binop Expr
        | Callee ( {Expr}/, )
        | new Type [ [Expr ]]
        | Int    | Double    | & ID
LHSExpr ::= ID    | * Expr
        | Expr -> ID [[ Expr ] ]
Unop ::= -.int  |  -.double  |  not  |  int2double
Binop ::= (+|-|*|/).(int|double)
        | (<|<=|>=|>|==|!=).(int|double)
        | <.unsigned
Callee ::= ID | ( * 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}
           {ILVarDecl} {ILStmt}
ILVarDecl: ILType String
  ILFormalVarDecl
```


MiniJava's IL Classes (3 of 6)

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

ILLabel: String

ILGlobalVar: String
```


MiniJava's IL Classes (4 of 6)

```
ILVar: ILVarDecl

ILExpr
  ILAssignableExpr
  ILVarExpr: ILVar
  ILPtrAccessExpr: ILExpr
  ILFieldAccessExpr: ILExpr ILType String
  ILSequenceFieldAccessExpr: ILExpr
ILUnopExpr: ILExpr
  IL{Int,Double}NegativeExpr,
    ILLOgicalNegateExpr, ILIntToDoubleExpr
```


MiniJava's IL Classes (5 of 6)

```
ILBinopExpr: ILExpr ILExpr
  IL{Int,Double}{Add,Sub,Mul,Div,
                 Equal,NotEqual,
                 LessThan,LessThanOrEqual
                 GreaterThanOrEqual,
                 GreaterThan}Expr,
  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

Integer	Use hardware representation (2, 4, and/or 8 bytes of memory, maybe aligned)
Bool	1 byte or word
Char	1-2 bytes or word
Pointer	Use hardware representation (2, 4, or 8 bytes, maybe two words if segmented machine)

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?

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


Layout of arrays

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

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


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

```
a : array [3] of
       array [2] of
         record;
           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:array [1000, 2000] of int;

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

for j:= 1 to 2000 do
  for i:= 1 to 1000 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

```
a : array of
  record;
    i : int;
    c : char;
  end;
```


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 ≈ 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