Pros and cons of interpretation

+ simple conceptually, easy to implement
+ fast turnaround time
+ good programming environments
+ easy to support fancy language features
+ clear reference implementation of language semantics

- 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

Alternate implementation strategy: compilation

Divide interpreter work into two parts:
- compile-time
- run-time

Compile-time does preprocessing
- perform some computations at compile-time once
- produce an equivalent program that gets run many times

Only advantage over interpreters: faster running programs

Compile-time processing

Decide layout of run-time data values
- use direct reference at precomputed offsets,
  not hash table lookups or searches along env. chains

Decide where variable contents will be stored
- registers
- stack frame slots at precomputed offsets
- global memory

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

Compilation plan

First, translate typechecked ASTs into
linear sequence of simple statements
called **intermediate code**
- a program in an **intermediate language** (IL)
- source-language, target-language independent

Then, translate intermediate code into target code

Two-step process helps separate concerns
- intermediate code generation from ASTs focuses on
  breaking down source-language contructs into simple
  and explicit pieces
- target code generation from intermediate code focuses on
  constraints of particular target machines

Can write many target code generators (back-ends),
many language-specific front-ends sharing same IL

Can implement optimizer for IL, shared by front- and back-ends

MiniJava's intermediate language

Want intermediate language to have only simple, explicit operations, without "helpful" features

- humans won't write IL programs!
- C-like is good

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, sequences
- no booleans
- no class types, no implicit class fields
- sequences have 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 "grammar" (part 1 of 2)

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

GlobalVarDecl ::= Type ID [= Value] ;

Type     ::= int | double | * Type
           | < Type > | { {Type ID}/, } | code

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 "grammar" (part 2 of 2)

```
Expr     ::= LHSExpr
           | Unop Expr
           | Expr Binop Expr
           | Callee ( {Expr}/, )
           | alloc 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 (part 1 of 3)

In `IL` subdirectory:

```
ILProgram: {ILGlobalVarDecl} {ILFunDecl}

ILGlobalVarDecl: ILType String
  ILInitializedGlobalVarDecl: ILValue

ILType
  ILIntType
  ILDoubleType
  ILPtrType: ILType
  ILSequenceType: ILType
  ILRecordType: {ILType String}
  ILCodeType

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

MiniJava's IL classes (part 2 of 3)

```
ILFunDecl: ILType String {ILFormalVarDecl}
           {ILVarDecl} {ILStmt}

ILVarDecl: ILType String
   ILFormalVarDecl

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

ILLabel: String

ILGlobalVar: String

ILVar: ILVarDecl
```

MiniJava's IL classes (part 3 of 3)

```
ILExpr
   ILAssignableExpr
      ILVarExpr: ILVar
      ILPtrAccessExpr: ILExpr
      ILFieldAccessExpr: ILExpr ILType String
         ILSequenceFieldAccessExpr: ILExpr
   ILUnopExpr: ILExpr
      IL{Int,Double}NegateExpr, ILLogicalNegateExpr,
      ILIntToDoubleExpr
   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
   ILFunCallExpr: ILType {ILExpr}
      ILDirectFunCallExpr: String
      ILIndirectFunCallExpr: ILExpr
      ILRuntimeCallExpr: String
```

Intermediate code generation in MiniJava

Choose representations for source-level data types
- translate each `ResolvedType` into `ILType`(s)

Recursively traverse ASTs, creating corresponding IL program
- `Expr` ASTs create `ILExpr` ASTs (& sometimes `ILStmt`s)
- `Stmt` ASTs create `ILStmt` ASTs
- `MethodDecl` ASTs create `ILFunDecl` ASTs
- `ClassDecl` ASTs create `ILGlobalVarDecl` ASTs
- `Program` ASTs create `ILProgram` ASTs

Traversal parallels typechecking and evaluation traversals

ICG operations on (source) ASTs named `lower`

IL AST classes in `IL` subdirectory

Data type representation (part 1)

What IL type to use for each source type?
- (what operations are we going to need on them?)

`int`:

`boolean`:

`double`:

Data type representation (part 2)

What IL type to use for each source type?
- (what operations are we going to need on them?)

Example:
```
class B {
   int i;
   D j;
}
```

instance of class `B`:

Inheritance

How to lay out subclasses?
- subclass inherits features of superclass
- subclass can be assigned to variable of superclass's type
  ⇒ subclass layout must "match" superclass's layout

Example:
```
class B {
   int i;
   D j;
}
class C extends B {
   int x;
   F y;
}
```

instance of class `C`:

Methods

How to translate a method?

Use a function
- name is "mangled": name of class + name of method
- make `this` an explicit argument

Example:
```
class B {
   ...
   int m(int i, double d) { ... body ... }
}
```

`B`'s method `m` translates to
```
int B_m(*{...B...} this, int i, double d) {
   ... translation of body ... }
```

Method invocation

How to implement method invocation?

Example:
```
class B {
   int m(...) { ... }
   E n(...) { ... }
}
class C extends B {
   int m(...) { ... }  // override
   F p(...) { ... }
}

B b1 = new B();
C c2 = new C();
B b2 = c2;
b1.m(...);
b1.n(...);
c2.m(...);
c2.n(...);
c2.p(...);
b2.m(...);
b2.n(...);
```

Methods via function pointers in instances

Simple idea:

- store code pointer for each *new* method in each instance
  - reuse member for *overriding* methods
- initialize with right method for that name for that object
- do "instance variable lookup" to get code pointer to invoke

Example:

```
class B {
   int i;
   int m(...) { ... }
   E n(...) { ... }
}
class C extends B {
   int j;
   int m(...) { ... }  // override
   F p(...) { ... }
}
```

instance of class `B`:

```
  *{ int i, *code m, *code n }
```

instance of class `C`:

```
  *{ int i, *code m, *code n, int j, *code p }
```

Manipulating method function pointers

Example:

```
B b1 = new B();
C c2 = new C();
B b2 = c2;
b1.m(3, 4.5);
b2.m(3, 4.5);
```

Translation:

```
*.. b1 = alloc {...B...};
b1->i = 0; b1->m = &B_m; b1->n = &B_n;

*.. c2 = alloc {...C...};
c2->i = 0; c2->m = &C_m; c2->n = &B_n;
c2->j = 0; c2->p = &C_p;

*.. b2 = c2;

(*(b1->m))(b1, 3, 4.5);

(*(b2->m))(b2, 3, 4.5);
```

Shared method function pointers

Observation:
  all direct instances of a class store the same
  method function pointer values

Idea:
  factor out common values into a single record
  shared by all direct instances of a class

- often called a **virtual function table**

+ smaller objects, faster object creation

– slower method invocations

`B`'s virtual function table (a global initialized variable):

```
{*code m, *code n} B_vtbl = {m=&B_m, n=&B_n};
```

Example:

```
B b1 = new B();
b1.m(3, 4.5);
```

Translation:

```
*.. b1 = alloc { int i, *{...B_vtbl...} vtbl };
b1->i = 0; b1->vtbl = &B_vtbl;
(*((b1->vtbl)->m))(b1, 3, 4.5);
```

Inheritance and virtual function tables

Virtual function table of subclass extends that of superclass with new methods, replaces function pointer values of overridden methods

Example:

```
class B {
   int i;
   int m(...) { ... }
   E n(...) { ... }
}
class C extends B {
   int j;
   int m(...) { ... }  // override
   F p(...) { ... }
}
```

Virtual function tables:

```
{*code m, *code n} B_vtbl =
   {m=&B_m, n=&B_n};
{*code m, *code n, *code p} C_vtbl =
   {m=&B_m, n=&C_n, p=&C_p};
```

Manipulating shared method function pointers

Example:

```
B b1 = new B();
C c2 = new C();
B b2 = c2;
b1.m(3, 4.5);
b2.m(3, 4.5);
```

Translation:

```
*.. b1 =
   alloc {int i, *{...B_vtbl...} vtbl};
b1->i = 0; b1->vtbl = &B_vtbl;

*.. c2 =
   alloc {int i, *{...C_vtbl...} vtbl, int j};
c2->i = 0; c2->vtbl = &C_vtbl; c2->j = 0;

*.. b2 = c2;

(*((b1->vtbl)->m))(b1, 3, 4.5);

(*((b2->vtbl)->m))(b2, 3, 4.5);
```

Data type representation (part 3)

What IL type to use for each source type?

- (what operations are we going to need on them?)

Nested records without implicit pointers, as in C

```
struct S1 {
   int x;
   struct S2 {
      double y;
      S3* z;
   } s2;
   int w;
} s1;
```

Data type representation (part 4)

What IL type to use for each source type?

- (what operations are we going to need on them?)

Unions, as in C

```
union U {
   int x;
   double y;
   S3* z;
   int w;
} u;
```

Data type representation (part 5)

What IL type to use for each source type?

- (what operations are we going to need on them?)

Arrays: *T*`[]`:

Data type representation (part 6)

What IL type to use for each source type?
- (what operations are we going to need on them?)

Multidimensional arrays: *T*`[][]`...
- array of arrays?
- rectangular matrix?

Data type representation (part 7)

What IL type to use for each source type?
- (what operations are we going to need on them?)

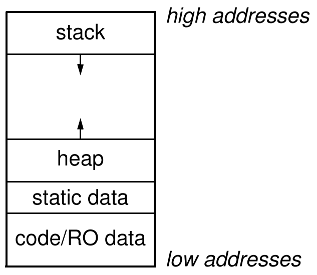
Strings
- null-terminated arrays of characters, as in C
- length-prefixed array of characters, as in Java
  - how big should length field be?

Data placement

Where to allocate space for each variable/data structure?

Key issue: what is the **lifetime** (**dynamic extent**)
of a variable/data structure?
- whole execution of program (global variables)
  $\Rightarrow$ **static** allocation
- execution of a procedure activation (formals, local vars)
  $\Rightarrow$ **stack** allocation
- variable (dynamically-allocated data)
  $\Rightarrow$ **heap** allocation

Parts of run-time memory



Code/RO data area
- read-only data & machine instruction area
- shared across processes running same program

Static data area
- place for read/write variables at fixed location in memory
- can start out initialized, or zeroed

Heap
- place for dynamically allocated/freed data
- can expand upwards through `sbrk` system call

Stack
- place for stack-allocated/freed data
- expands/contracts downwards automatically

Static allocation

Statically allocate variables/data structures with global lifetime

- global variables in C, `static` class variables in Java
- `static` local variables in C, all locals in Fortran
- compile-time constant strings, records, arrays, etc.
- machine code

Compiler uses symbolic addresses
Linker assigns exact addresses, patches compiled code

`ILGlobalVarDecl` to declare statically allocated variable
`ILFunDecl` to declare function
`ILGlobalAddressExpr` to compute address of
statically allocated variable or function

Stack allocation

Stack-allocate variables/data structures with **LIFO** lifetime

- last-in first-out (stack discipline):
  data structure doesn't outlive previously allocated
  data structures on same stack

Activation records usually allocated on a stack

- a stack-allocated a.r. called a **stack frame**
- frame includes formals, locals

Fast to allocate & deallocate storage
Good memory locality

Multiple threads: each thread gets its own stack

- where in memory to place thread stacks?

`ILVarDecl` to declare stack allocated variable
`ILVarExpr` to reference stack allocated variable

- both with respect to some `ILFunDecl`

Problems with stack allocation

Stack allocation works only
for data that doesn't outlive containing stack frame

Violated if first-class functions allowed

```
(int(*)(int)) curried(int x) {
   int nested(int y) { return x+y; }
   return &nested;
}

(int(*)(int)) f = curried(3);
(int(*)(int)) g = curried(4);

int a = f(5);
int b = g(6);

// what are a and b?
```

Problems with stack allocation

Violated if inner classes allowed

```
Inner curried(int x) {
   class Inner {
      int nested(int y) { return x+y; }
   };
   return new Inner();
}

Inner f = curried(3);
Inner g = curried(4);

int a = f.nested(5);
int b = g.nested(6);

// what are a and b?
```

Problems with stack allocation

Violated if pointers to locals allowed

```
int* addr(int x) { return &x; }

int* p = addr(3);
int* q = addr(4);

int a = (*p) + 5;
int b = (*p) + 6;

// what are a and b?
```

Heap allocation

Heap-allocate variables/data structures with unknown lifetime
- `new`/`malloc` to allocate space
- `delete`/`free`/garbage collection to deallocate space

Heap-allocate activation records (environments at least) of first-class functions, functions containing inner classes

Put locals with address taken into heap-allocated environment, or make illegal, or make undefined

Relatively expensive to manage

Can have dangling references, storage leaks if don't `free` right
- use automatic garbage collection in place of manual `free` to avoid these problems

`ILAllocateExpr`, `ILAllocateSequenceExpr` to allocate heap memory

Garbage collection implicitly frees heap memory

Parameter passing

When passing arguments, need to support right semantics

An issue: when is argument expression evaluated?
- before call, or if & when needed by callee?

Another issue: what happens if formal assigned in callee?
- effect visible to caller? if so, when?
- what effect in face of aliasing among arguments, lexically visible variables?

Different choices lead to different representations for passed arguments and different code to access formals

Some parameter passing modes

Parameter passing options:
- call-by-value, call-by-sharing
- call-by-reference, call-by-value-result, call-by-result
- call-by-name, call-by-need
- ...

Language-specific front-ends translate their parameter-passing modes into modes offered by IL
- ILs typically support only very simple modes

Call-by-value

If formal is assigned, caller's value remains unaffected

```
class C {
   int a;
   void m(int x, int y) {
      x = x + 1;
      y = y + a;
   }
   void n() {
      a = 2;
      m(a, a);
      System.out.println(a);
   }
}
```

Implement by passing copy of argument value

- trivial for scalars: ints, booleans, etc.
- inefficient for aggregates: arrays, records, strings, ...

Call-by-sharing

If implicitly reference aggregate data via pointer
(e.g. Java, Lisp, Smalltalk, ML, ...)
then call-by-sharing is call-by-value applied to implicit pointer

- "call-by-pointer-value"

```
class C {
   int[] a = new int[10];
   void m(int[] x, int[] y) {
      x[0] = x[0] + 1;
      y[0] = y[0] + a[0];
      x = new int[20];
   }
   void n() {
      a[0] = 2;
      m(a, a);
      System.out.println(a);
   }
}
```

- efficient, even for big aggregates
- re-assignments of formal don't affect caller
- updates to target of formal visible to caller immediately

Call-by-reference

If formal is assigned, actual value is changed in caller

- change occurs immediately

```
class C {
   int a;
   void m(int& x, int& y) {
      x = x + 1;
      y = y + a;
   }
   void n() {
      a = 2;
      m(a, a);
      System.out.println(a);
   }
}
```

Implement by passing pointer to actual

- efficient for big data structures
- references to formal do extra dereference, implicitly

Call-by-value-result: do assign-in, assign-out

- subtle differences if same actual passed to multiple formals

Call-by-result

Write-only formals, to return extra results;
no incoming actual value expected

- "out parameters"
- formals cannot be read in callee,
  actuals don't need to be initialized in caller

```
class C {
   int a;
   void m(int&out x, int&out y) {
      x = 1;
      y = a + 1;
   }
   void n() {
      a = 2;
      int b;
      m(b, b);
      System.out.println(b);
   }
}
```

Can implement as in call-by-reference or call-by-value-result

Call-by-name, call-by-need

Variations on **lazy evaluation**

- only evaluate argument expression if & when needed by callee function

Supports very cool programming tricks

Hard to implement efficiently in traditional compiler

- introduce nested functions for each lazily evaluated expression!

Incompatible with side-effects

⇒ only in purely functional languages, e.g. Haskell, Miranda

Main ICG operations

`ILProgram Program.lower();`

- translate the whole program into an `ILProgram`

`void ClassDecl.lower(ILProgram);`

- translate method decls
- declare the class's method record (vtbl)

```
void MethodDecl.lower(ILProgram,
                      ClassSymbolTable);
```

- translate into IL fun decl, add to IL program

`void Stmt.lower(ILFunDecl);`

- translate into IL statement(s), add to IL fun decl

`ILExpr Expr.evaluate(ILFunDecl);`

- translate into IL expr, return it

`ILType ResolvedType.lower();`

- return corresponding IL type

An example ICG operation

```
class IntLiteralExpr extends Expr {
   int value;

   ILExpr lower(ILFunDecl fun) {
      return new ILIntConstantExpr(value);
   }
}
```

An example ICG operation

```
class AddExpr extends Expr {
   Expr arg1;
   Expr arg2;

   ILExpr lower(ILFunDecl fun) {
      ILExpr arg1_expr = arg1.lower(fun);
      ILExpr arg2_expr = arg2.lower(fun);
      return new ILIntAddExpr(arg1_expr, arg2_expr);
   }
}
```

An example overloaded ICG operation

```
class EqualExpr extends Expr {
   Expr arg1;
   Expr arg2;

   ILExpr lower(ILFunDecl fun) {
      ILExpr arg1_expr = arg1.lower(fun);
      ILExpr arg2_expr = arg2.lower(fun);
      if (arg1.getResultType().isIntType() &&
          arg2.getResultType().isIntType()) {
         return new ILIntEqualExpr(arg1_expr,
                                   arg2_expr);
      } else if (arg1.getResType().isBoolType() &&
                 arg2.getResType().isBoolType()) {
         return new ILIntEqualExpr(arg1_expr,
                                   arg2_expr);
      } else {
         throw new InternalCompilerError(...);
      }
   }
}
```

An example ICG operation

```
class VarDeclStmt extends Stmt {
   String name;
   Type type;
   VarInterface var_iface;  // set during typechecking

   void lower(ILFunDecl fun) {
      fun.declareLocal(var_iface);
   }
}
```

`declareLocal` declares a new local variable in the IL function

ICG of variable references

```
class VarExpr extends AssignableExpr {
   String name;
   VarInterface var_iface;  // set during typechecking

   ILExpr lower(ILFunDecl fun) {
      return var_iface.generateRead(fun);
   }
   void lowerAssign(ILFunDecl fun, Expr rhs) {
      ILExpr rhs_expr = rhs.lower(fun);
      var_iface.generateAssignment(rhs_expr, fun);
   }
}

class AssignStmt extends Stmt {
   ILAssignableExpr lhs;
   Expr rhs;

   void lower(ILFunDecl fun) {
      lhs.lowerAssign(fun, rhs);
   }
}
```

ICG of local variable references

```
abstract class VarInterface {
   String name;
   abstract ILExpr generateRead(ILFunDecl fun);
   abstract void generateAssignment(ILExpr rhs,
                                    ILFunDecl fun);
}

class LocalVarInterface extends VarInterface {
   ILExpr generateRead(ILFunDecl fun) {
      ILVar var = fun.lookupVar(this);
      return new ILVarExpr(var);
   }
   void generateAssignment(ILExpr rhs_expr,
                           ILFunDecl fun) {
      ILVar var = fun.lookupVar(this);
      fun.addStmt(
         new ILAssignStmt(new ILVarExpr(var),
                          rhs_expr));
   }
}
```

ICG of instance variable references

```
class InstanceVarInterface extends VarInterface {
   ClassSymbolTable class_st;
   ILExpr generateRead(ILFunDecl fun) {
      ILExpr rcvr_expr =
        new ILVarExpr(fun.lookupThis());
      ILType class_type =
         ILType.classILType(class_st);
      ILRecordMember var_member =
         class_type.getRecordMember(name);
      ILAssignableExpr field_access =
         new ILFieldAccessExpr(rcvr_expr,
                               class_type,
                               var_member);
      return field_access;
   }
   void generateAssignment(ILExpr rhs_expr,
                           ILFunDecl fun) {
      ... same as above ...
      ILAssignableExpr field_access = ...;
      fun.addStmt(
         new ILAssignStmt(field_access, rhs_expr));
   }
}
```

The real code

```
class InstanceVarInterface extends VarInterface {
   ...
   ILExpr generateRead(ILFunDecl fun) {
      ...
      ILAssignableExpr field_access =
         new ILFieldAccessExpr(rcvr_expr,
                               class_type,
                               var_member);
      ILVar result =
         fun.newTempVar(var_member.getType());
      fun.addStmt(
         new ILAssignStmt(new ILVarExpr(result),
                          field_access));
      return new ILVarExpr(result);
   }
}
```

ICG of if statements

What IL code to generate for an if statement?

if (*testExpr*) *thenStmt* else *elseStmt*

ICG of if statements

```
class IfStmt extends Stmt {
   Expr test;
   Stmt then_stmt;
   Stmt else_stmt;

   void lower(ILFunDecl fun) {
      ILExpr test_expr = test.lower(fun);
      ILLabel false_label = fun.newLabel();
      fun.addStmt(
         new ILCondBranchFalseStmt(test_expr,
                                   false_label));
      then_stmt.lower(fun);
      ILLabel done_label = fun.newLabel();
      fun.addStmt(new ILGotoStmt(done_label));
      fun.addStmt(new ILLabelStmt(false_label));
      else_stmt.lower(fun);
      fun.addStmt(new ILLabelStmt(done_label));
   }
}
```

ICG of print statements

What IL code to generate for a print statement?

```
System.out.println(expr);
```

No IL operations exist that do printing (or any kind of I/O)!

Runtime libraries

Can provide some functionality of compiled program in **external runtime libraries**

- libraries written in any language, compiled separately
- libraries can contain functions, data declarations

Compiled code includes calls to functions & references to data declared libraries

Final application links together compiled code and runtime libraries

Often can implement functionality either through compiled code or through calls to library functions

- tradeoffs?

ICG of print statements

```
class PrintlnStmt extends Stmt {
   Expr arg;

   void lower(ILFunDecl fun) {
      ILExpr arg_expr = arg.lower(fun);
      ILExpr call_expr =
         new ILRuntimeCallExpr("println_int",
                               arg_expr);
      fun.addStmt(new ILExprStmt(call_expr));
   }
}
```

ICG of new expressions

What IL code to generate for a `new` expression?

```
class C extends B {
   inst var decls
   method decls
}
... new C() ...
```

ICG of new expressions

```
class NewExpr extends Expr {
   String class_name;

   ILExpr lower(ILFunDecl fun) {
      generate code to:
         allocate instance record, assign ptr to temp var
         initialize vtbl field with class's method record
         initialize inst vars to default values
      return read of temp var
   }
}
```

An example ICG operation

```
class MethodCallExpr extends Expr {
   String class_name;

   ILExpr lower(ILFunDecl fun) {
      generate code to:
         evaluate receiver, put in temp var
         evaluate arg exprs
         test whether receiver temp var is null
         load vtbl member of receiver temp var
         load called method member of vtbl
         call fun ptr, passing receiver temp var and arg exprs,
            put result in temp var
      return read of result temp var
   }
}
```

IGC of array operations

What IL code to generate for array operations?

```
new type[expr]
arrayExpr.length
arrayExpr[indexExpr]
arrayExpr[indexExpr] = ...
```

IGC of doubles

How to implement overloaded operators?

```
class AddExpr extends Expr {
   Expr arg1;
   Expr arg2;

   ILExpr lower(ILFunDecl fun) {
      ILExpr arg1_expr = arg1.lower(fun);
      ILExpr arg2_expr = arg2.lower(fun);
      ... generate different ILExpr, depending on whether
      args are both doubles or both ints ...
   }
}
```

How to implement implicit conversions from int to double?

- assignments of int values to double contexts
  - extend existing `Expr.convertIfNeeded` operation
- mixed int & double binary operations