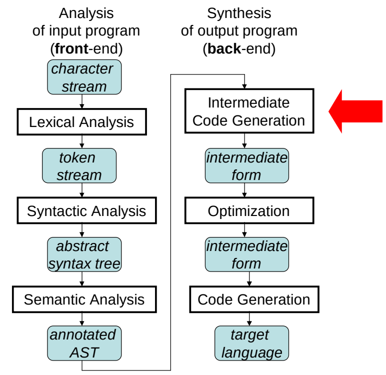
Intermediate Code Generation

Part 1. Overview

With the fully analyzed program expressed as an annotated AST, it's time to translate it into code

Compiler Passes



Compilation Overview

First, translate typechecked ASTs into linear sequence of simple statements called **intermediate code**
- a program in an **intermediate language** (IL) [also **IR**]
- 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 constructs into simple and explicit pieces
- target code generation from intermediate code 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 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, 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
```

IL Example

```
class Fac {
   public int ComputeFac(int num) {
      int numAux = 0;
      if (num < 1)
         numAux = 1;
      else
         numAux = num * this.ComputeFac(num-1);
      return numAux;
   }
}
```

Source

```
class Fac {
 int ComputeFac(int num) {
   int numAux = 0;
   if ((num) < (1)) numAux = 1;
   else numAux = (num) * ((this).ComputeFac((num) - (1)));
   return numAux;
 }
}
```

PrintAST


```
class Fac {
 int ComputeFac(int num) {
   int numAux = 0;
   if ((num) < (1)) numAux = 1;
   else numAux = (num) * ((this).ComputeFac((num) - (1)));
   return numAux;
 }
}
```

PrintAST

```
int Fac.ComputeFac(*{*? <vtbl>} this, int num.26210109) {
 int numAux.10730689;
 int .T.1;
 *{*? <vtbl>} .T.0;
 numAux.10730689 := 0;
 iffalse num.26210109 <.int 1 goto L0.Fac.ComputeFac;
 numAux.10730689 := 1;
 goto L1.Fac.ComputeFac;
 label L0.Fac.ComputeFac;
 .T.0 := this;
 iftrue .T.0 !=.int 0 goto L2.Fac.ComputeFac;
 throw new Exception("dereferencing a null pointer");
 label L2.Fac.ComputeFac;
 .T.1 := (*.T.0-><vtbl>:*{*? ComputeFac}->ComputeFac:*code)(.T.0, num.26210109-.int 1):int;
 numAux.10730689 := num.26210109 *.int .T.1;
 label L1.Fac.ComputeFac;
 return numAux.10730689;
}
```

PrintIL

Code Generation Considerations

Our generated program will 'run' in an environment that is determined by many factors:

- Hardware
- Operating System
- System Software Conventions
  - Calling conventions
  - Exception handling conventions
  - Data storage conventions

Code Generation Considerations
(Continued)

This means that there are many decisions to be made are part of our Intermediate Code Generation design. Two man factors influence our design:

- Performance
  - Compiler performance
  - Run time performance
- Compatibility
  - Inter-operability

ICG Design Tasks

Decide layout of run-time data values
- use direct reference at precomputed offsets, not e.g. hash table lookups

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

Decide on IL sequences for complex operations
- FORs, Calls, Returns, etc.

Intermediate Code Generation
Part 2. Storage Layout

Run-time storage layout:

- Plan how and where to keep data at run-time
- 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

- Aggregate scalars together
- Different compilers make different decisions
- Decisions are sometimes machine dependent
  - Note that through the discussion of the front-end, we never mentioned the target machine
  - We didn't in interpretation, either
  - But now it's going to start to come up constantly
  - Necessarily, some of what we will say will be "typical", not universal.

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][2]
a[2][1]
a[2][2]
a[3][1]
a[3][2]

Implications of Array Layout

- Which is better if row-major? col-major?

```
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

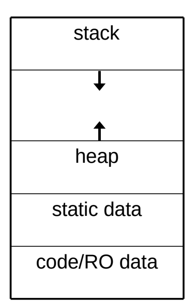
String representation

- 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

Parts of run-time memory



- Code/Read-only data area
  - Shared across processes running same program
- Static data area
  - Can start out initialized or zeroed
- Heap
  - Can expand upwards through (e.g. `sbrk`) system call
- Stack
  - Expands/contracts downwards automatically

Static allocation

- Statically allocate variables/data structures with global lifetime
  - Machine code
  - Compile-time constant scalars, strings, arrays, etc.
  - Global variables
  - `static` locals in C, all variables in FORTRAN
- Compiler uses symbolic addresses
- Linker assigns exact address, patches compiled code

Stack allocation

- Stack-allocate variables/data structures with LIFO lifetime
  - Data doesn't outlive previously allocated data on the same stack
- Stack-allocate procedure activation records
  - A stack-allocated activation record = a *stack frame*
  - Frame includes formals, locals, temps
  - And housekeeping: static link, dynamic link, …
- Fast to allocate and deallocate storage
- Good memory locality

Stack allocation II

- What about variables local to nested scopes within one procedure?

```
procedure P() {
  int x;
  for(int i=0; i<10; i++){
      double x;
      …
  }
  for(int j=0; j<10; j++){
      double y;
      …
  }
}
```


Stack allocation: constraints I

- No references to stack-allocated data allowed after returns
- This is violated by general first-class functions

```
proc foo(x:int): proctype(int):int;
  proc bar(y:int):int;
  begin
    return x + y;
  end bar;
begin
  return bar;
end foo;

var f:proctype(int):int;
var g:proctype(int):int;

f := foo(3);    g := foo(4);
output := f(5); output := g(6);
```


Stack allocation: constraints II

- Also violated if pointers to locals are allowed

```
proc foo (x:int): *int;
  var y:int;
begin
  y := x * 2;
  return &y;
end foo;

var w,z:*int;

z := foo(3);
w := foo(4);

output := *z;
output := *w;
```


Heap allocation

- For data with unknown lifetime
  - `new/malloc` to allocate space
  - `delete/free/`garbage collection to deallocate
- Heap-allocate activation records of first-class functions
- Relatively expensive to manage
- Can have dangling reference, storage leaks
  - Garbage collection reduces (but may not eliminate) these classes of errors

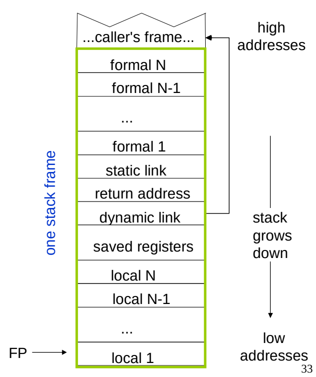
Stack frame layout

- Need space for
  - Formals
  - Locals
  - Various housekeeping data
    - Dynamic link (pointer to caller's stack frame)
    - Static link (pointer to lexically enclosing stack frame)
    - Return address, saved registers, …
- Dedicate registers to support stack access
  - FP - frame pointer: ptr to start of stack frame (fixed)
  - SP - stack pointer: ptr to end of stack (can move)

Key property

- All data in stack frame is at a *fixed, statically computed* offset from the FP
- This makes it easy to generate fast code to access the data in the stack frame
  - And even lexically enclosing stack frames
- Can compute these offsets solely from the symbol tables
  - Based also on the chosen layout approach

Stack Layout



Accessing locals

- If a local is in the same stack frame then
```
t := *(fp + local_offset)
```
- If in lexically-enclosing stack frame
```
t := *(fp + static_link_offset)
t := *(t + local_offset)
```
- If farther away
```
t := *(fp + static_link_offset)
t := *(t + static_link_offset)
…
t := *(t + local_offset)
```


At compile-time…

- …need to calculate
  - Difference in nesting depth of use and definition
  - Offset of local in defining stack frame
  - Offsets of static links in intervening frames

Intermediate Code Generation
Part 3. Calling Conventions

Calling conventions

- Define responsibilities of caller and callee
  - To make sure the stack frame is properly set up and torn down
- Some things can only be done by the caller
- Other things can only be done by the callee
- Some can be done by either
- So, we need a protocol

Typical calling sequence

- Caller
  - Evaluate actual args
    - Order?
  - Push onto stack
    - Order?
    - Alternative: First k args in registers
  - Push callee's static link
    - Or in register? Before or after stack arguments?
  - Execute call instruction
    - Hardware puts return address in a register
- Callee
  - Save return address on stack
  - Save caller's frame pointer (dynamic link) on stack
  - Save any other registers that might be needed by caller
  - Allocates space for locals, other data
    `sp := sp – size_of_locals – other_data`
    - Locals stored in what order?
  - Set up new frame pointer (`fp := sp`)
  - Start executing callee's code

Typical return sequence

- Callee
  - Deallocate space for local, other data
    `sp := sp + size_of_locals + other_data`
  - Restore caller's frame pointer, return address & other regs, all without losing addresses of stuff still needed in stack
  - Execute return instruction
- Caller
  - Deallocate space for callee's static link, args
    - `sp := fp`
  - Continue execution in caller after call

Accessing procedures

similar to accessing locals

- Call to procedure declared in same scope:

```
static_link := fp
call p
```

- Call to procedure in lexically-enclosing scope:

```
static_link := *(fp + static_link_offset)
call p
```

- If farther away

```
t := *(fp + static_link_offset)
t := *(t  + static_link_offset)
…
static_link := *(t + static_link_offset)
call p
```


Some questions

- Return values?
- Local, variable-sized, arrays

```
proc P(int n) {
  var x array[1 .. n] of int;
  var y array[-5 .. 2*n] of array[1 .. n] int;
  …
}
```

- Max length of dynamic-link chain?
- Max length of static-link chain?

Accessing locals

- Same stack frame then

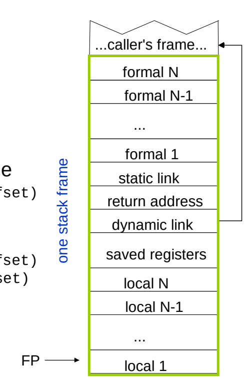
```
t := *(fp + local_offset)
```

- Lexically-enclosing stack frame

```
t := *(fp + static_link_offset)
t := *(t + local_offset)
```

- If farther away

```
t := *(fp + static_link_offset)
t := *(t + static_link_offset)
…
t := *(t + local_offset)
```



Exercise: apply to this example

```
module M;
  var x:int;
  proc P(y:int);
    proc Q(y:int);
       var qx:int;
      begin R(x+y);end Q;
    proc R(z:int);
       var rx,ry:int;
      begin P(x+y+z);end R;
  begin Q(x+y); R(42); P(0); end P;
begin
  x := 1;
  P(2);
end M.
```


Parameter Passing

When passing arguments, need to support the 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
- ...

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
- assignments of formal to a different aggregate (e.g. `x = ...`) don't affect caller
- updates to contents of aggregate (e.g. `x[...] = ...`) 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
- Incompatible with side-effects implies only in purely functional languages, e.g. Haskell, Miranda

Call by value vs. call by name

```
begin
  integer n;
  procedure p(k: integer);
    begin
    print(k);
    n := n+1;
    print(k);
    end;
  n := 0;
  p(n+10);
  end;
```


Original Call-by-name

Algol 60 report: "Substitute actual for formal, evaluate."

Consequences:

```
procedure CALC (a,b,c,i); real a,b,c; integer i;
   begin i:= 1; a:=0; b:=1;
   loop: a := a+c;
         b := b*c;
         if i = 10 then go to finish;
         i := i+1; go to loop;
 finish: end;

CALC(sum, product, b*(b-j), j);
```


Original Call-by-name

```
procedure CALC (a,b,c,i); real a,b,c; integer i;
      begin j:= 1; sum:=0; product:=1;
      loop: sum := sum+(b*(b-j));
            product := product*(b*(b-j));
            if j = 10 then go to finish;
            j := j+1; go to loop;
    finish: end;

   CALC(sum, product, b*(b-j), j);
```

sum := $\sum_{j=1..10}$ b*(b-j)

product := $\prod_{j=1..10}$ b*(b-j)

Fortran example

```
SUBROUTINE S(EL, K)
K = 2
EL = 0
RETURN
END
A(1) = A(2) = 1
I=1
CALL S (A(I),I))
```

Algol 60 Example

```
procedure S (el, k);
   integer el, k;
   begin
        k := 2;
        el := 0
   end;
A[1] := A[2] := 1;
i := 1;
S (A[i], i);
```


What happens here?

```
begin
  integer n;
  procedure p(k: integer);
     begin
     print(n);
     end;
  n := 5;
  p(n/0);
  end;
```


Intermediate Code Generation

Part 4. MiniJava IL Definition

MiniJava's Intermediate Language
(Reprise)

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, 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 "grammar" (1 of 2)

```
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 "grammar" (2 of 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
```

IL Example

```
class Fac {
  public int ComputeFac(int num) {
    int numAux = 0;
    if (num < 1)
      numAux = 1;
    else
      numAux = num * this.ComputeFac(num-1);
    return numAux;
  }
}
```

Source

```
class Fac {
 int ComputeFac(int num) {
  int numAux = 0;
  if ((num) < (1)) numAux = 1;
  else numAux = (num) * ((this).ComputeFac((num) - (1)));
  return numAux;
 }
}
```

PrintAST


```
class Fac {
 int ComputeFac(int num) {
  int numAux = 0;
  if ((num) < (1)) numAux = 1;
  else numAux = (num) * ((this).ComputeFac((num) - (1)));
  return numAux;
 }
}
```

PrintAST

```
int Fac.ComputeFac(*{*? <vtbl>} this, int num.26210109) {
 int numAux.10730689;
 int .T.1;
 *{*? <vtbl>} .T.0;
 numAux.10730689 := 0;
 iffalse num.26210109 <.int 1 goto L0.Fac.ComputeFac;
 numAux.10730689 := 1;
 goto L1.Fac.ComputeFac;
 label L0.Fac.ComputeFac;
 .T.0 := this;
 iftrue .T.0 !=.int 0 goto L2.Fac.ComputeFac;
 throw new Exception("dereferencing a null pointer");
 label L2.Fac.ComputeFac;
 .T.1 := (*.T.0-><vtbl>:*{*? ComputeFac}->ComputeFac:*code)(.T.0, num.26210109-.int 1):int;
 numAux.10730689 := num.26210109 *.int .T.1;
 label L1.Fac.ComputeFac;
 return numAux.10730689;
}
```

PrintIL

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 program
- `Expr` ASTs create `ILExpr` ASTs
- `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 (1)

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

int:

boolean:

double:

Data Type Representations (2)

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

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

Instance of Class B

Inheritance

How to lay out subclasses
- Subclass inherits from superclass
- Subclass can be assigned to a variable of superclass type implying subclass layout must "match" superclass layout

```
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 do we implement method invocation?

```
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 w/right method for that name for that object
- Do "instance var lookup" to get code pointer to invoke

```
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 Ptrs

- 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

Often called a **virtual function table**, or **vtbl**

+ small 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);
```


Method Inheritance

A subclass inherits all the methods of its superclasses

- its method record includes all fields of its superclass

Virtual function tables 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(...) { ... }
}
```

B's method record value: `{*code m, *code n} B_vtbl =`
`{m=&B_m, n=&B_n};`

C's method record value: `{*code m, *code n, *code p} C_vtbl=`
`{m=&C_m, n=&B_n, p=&C_p};`

Example

- 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 (3)

What IL type to use for each source type?

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

array of T:

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.lower(ILFunDecl);`
- translate into IL expr, return it

`ILType Type.lower();`
`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);
   }
}
```


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.getResultType().isBoolType() &&
               arg2.getResultType().isBoolType()) {
      return new ILBoolEqualExpr(arg1_expr, arg2_expr);
    } else {
        throw new InternalCompilerError(...);
    }
  }
}
```

An Example ICG Operation

```
class VarDeclStmt extends Stmt {
    String name;
    Type type;
    void lower(ILFunDecl fun) {
       fun.declareLocal(type.lower(), name);
    }
}
```

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

ICG of Variable References

```
class VarExpr extends Expr {
   String name;
   VarInterface var_iface;  //set during typechecking
   ILExpr lower(ILFunDecl fun) {
       return var_iface.generateRead(fun);
   }
}
class AssignStmt extends Stmt {
   String lhs;
   Expr rhs;
   VarInterface lhs_iface;  //set during typechecking
   void lower(ILFunDecl fun) {
       ILExpr rhs_expr = rhs.lower(fun);
       lhs_iface.generateAssignment(rhs_expr, fun);
   }
}
```

`generateRead`/`generateAssignment` gen IL code to read/assign the variable
- code depends on the kind of variable (local vs. instance)

ICG of Instance Variable References

```
class InstanceVarInterface extends VarInterface {
   ClassSymbolTable class_st;

   ILExpr generateRead(ILFunDecl fun) {
     ILExpr rcvr_expr =
        new ILVarExpr(fun.lookupVar("this"));
     ILType class_type =
        ILType.classILType(class_st);
     ILRecordMember var_member =
        class_type.getRecordMember(name);
     return new ILFieldAccessExpr(rcvr_expr,
                                  class_type,
                                  var_member);
   }
```


ICG of Instance Variable Reference

```
   .
   .
   void generateAssignment(ILExpr rhs_expr,
                           ILFunDecl fun) {
     ILExpr rcvr_expr =   // same as in generateRead
        new ILVarExpr(fun.lookupVar("this"));
     ILType class_type =
        ILType.classILType(class_st);
     ILRecordMember var_member =
        class_type.getRecordMember(name);
     ILAssignableExpr lhs =
        new ILFieldAccessExpr(rcvr_expr,
                              class_type,
                              var_member);
     fun.addStmt(new ILAssignStmt(lhs, rhs_expr));
   }
 }
```


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 in 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));
   }
}
```

What about printing doubles?

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
       initialize vtbl field with class's method record
       initialize inst vars to default values
     return reference to allocated record
   }
}
```


An Example ICG Operation

```
class MethodCallExpr extends Expr {
   String class_name;
   ILExpr lower(ILFunDecl fun) {
     generate code to:
        evaluate receiver and arg exprs
        test whether receiver is null
        load vtbl member of receiver
        load called method member of vtbl
        call fun ptr, passing receiver and args
     return call expr
   }
}
```

ICG of Array Operations

What IL code to generate for array operations?

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


Storage Layout

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)
  => static allocation
- execution of a procedure activation (formals, local vars)
  => stack allocation
- variable (dynamically-allocated data)
  => heap allocation

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 address

Linker assigns exact address, 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. is called a *stack frame*
- frame includes formals, locals, static link of procedure
- dynamic link = stack frame above

Fast to allocate & deallocate storage

Good memory locality

`ILVarDecl` to declare stack allocated variable

`ILVarExpr` to reference stack allocated variable
- **both with respect to some `ILFunDecl`**

Problems with Stack Allocation (1)

Stack allocation works only when can't have references to stack allocated data after containing function returns

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 (2)

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 (3)

Violated if pointers to locals are 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

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; Garbage collection implicitly frees heap