Where are we now?

source code → front end → middle end → back end → target code

errors

IR
Where are we now?

Front end:
- produce intermediate representation (IR) of the program

Middle end:
- transform IR into equivalent, more efficient program

Back end:
- transform IR into target code
Why IR?

Why use an intermediate representation?

- breaks compiler into manageable pieces
  - good software engineering
- allows compiler to make multiple passes over the program
- supports multiple languages and multiple front and back ends
- enables machine-independent optimization
  - general techniques, multiple passes
Properties of IR

Important properties of IR
- ease, cost of generation
- ease, cost of manipulation
- level of abstraction
- freedom of expression
- size of typical procedure

Subtle design decisions in IR have far-reaching effects on speed and effectiveness of the compiler

Level of exposed detail is crucial consideration
Multiple IRs

Compiler often supports multiple IRs

Usually two levels
- high-level IR (language independent, but closer to language)
- low-level IR (machine independent, but closer to machine)

The project: 3 levels
- ASTs, IR trees (Piglet), 3-address code (Kanga)
Why multiple IRs?

Different IRs are better for different analyses and transformations

High-level IR
- good for method inlining and specialization
- good for loop optimizations

Low-level IR
- good for low-level optimizations: code motion, strength reduction, common subexpression elimination
Why does the choice of IR matter?

Could do low-level optimizations on high-level IR, but translation to low-level IR introduces more opportunities to optimize.

Example:

- `a[i+1] + b[i+1]`  
  --> (common subexpression elimination)  
  
  `t = i+1`  
  `a[t] + b[t]`  
  --> (translation to low IR)  
  
  `t = i+1`  
  `*(a + t*4) + *(b + t*4)`  
  --> (common subexpression elimination)  
  
  `t = i+1`  
  `u = t*4`  
  `*(a + u) + *(b + u)`
Choices of IR

Representations talked about in the literature:
- abstract syntax trees (AST)
- linear (operator) form of tree
- directed acyclic graphs (DAG)
- control flow graphs (CFG)
- program dependence graphs (PDG)
- static single assignment form (SSA)
- stack code
- three-address code
- assembly code
- hybrids
Categories

Broadly, IRs fall into three categories

- **Structural**
  - graphically oriented
  - examples: trees, directed acyclic graphs
  - heavily used in source-to-source translators
  - nodes, edges tend to be large

- **Linear**
  - pseudo code for some abstract machine
  - large variation in level of abstraction
  - simple, compact data structures
  - easier to rearrange
  - example: three-address code

- **Hybrids**
  - combination of graphs and linear code
  - attempt to have “best of both worlds”
  - example: control-flow graphs
Abstract syntax trees

Represent program as trees with node for each syntactic construct

Ex: x - 2 * y

Preserves much of the program structure

Not language independent

Can be difficult to manipulate
**DAGs**

An AST with a unique node for each value

Ex:

\[ x := 2 \times y + \sin(2\times x) \]

\[ z := x / 2 \]

Uses less memory than AST

Easy to identify shared values

Difficult to manipulate, esp. to maintain sharing
IR trees

Tree structure very similar to AST

Can be at many different levels of abstraction

Commonly:
- Contains high-level constructs common to many languages
  - expression nodes
  - statement nodes
- But language-independent
Expression nodes

Integers and program variables

Binary operations
- arithmetic
- logical
- comparisons

Unary operations

Array accesses
Statement nodes

Block statements (sequences)
Assignments
Array assignments
If-then, if-then-else
While loops
Function calls
Return
Break, continue, goto
Switch
Expression evaluation (discard result)
High-level IR trees

Preserves high-level structure of the program

Good for optimizations on large-scale structures of the program
- method inlining
- method specialization
- loop optimizations

```c
for (i = 0; i < N; i++) {
    A[i] = 0;
    for (j = 0; j < M; j++)
        A[i] += B[j][i];
}
```

```c
for (i = 0; i < N; i++)
    A[i] = 0;
for (j = 0; j < M; j++)
    for (i = 0; i < N; i++)
        A[i] += B[j][i];
```
Low-level IR trees

Rather than representing loops, if, switch explicitly introduce jump statements:

- jump e
  - unconditionally jump to label e

- cjump a e
  - jump to label e if a is true

Could also distinguish between jump on true and jump on false:

- fjump a e
- tjump a e
Three-address code

Sequence of instructions of the form:

- \( a = b \text{ OP } c \)

Each instruction has at most three addresses \((a, b, c)\)

Also called *quadruples* \((a, b, c, \text{ OP})\) or *quads* or *tuples*

Ex:

- \( a = (b+c) \ast (-e) \)
- \( t1 = b+c \)
  - \( t2 = -e \)
  - \( a = t1 \ast t2 \)
Tuple operations

Operands of tuples are:

- program variables
- integer constants
- *temporary variables*
  - new locations used to store intermediate values
  - contrast with operand stack
3AC instructions

Assignment
- Binary operations
  - $a = b \text{ OP } c$
- Unary
  - $a = \text{ OP } b$
- Copy
  - $a = b$
- Load/store
  - $a = *b$, $a = b$
3AC instructions

Control-flow instructions:
- label L
- jump L
- cjump a L

Calls:
- param a
- call f
- a = call f
3AC representation

Compile-time space can be a serious issue
- more so in the past
- but, very important for analyses that require the whole program

Compact forms of 3AC:
- quadruples
- triples
- indirect triples

Trade-off: compactness vs. ease of manipulation
Today: speed (and locality) more important than size
Quadruples

\[ x - 2 * y \]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>t1</td>
<td>y</td>
</tr>
<tr>
<td>2</td>
<td>loadi</td>
<td>t2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>mul</td>
<td>t3</td>
<td>t2</td>
</tr>
<tr>
<td>4</td>
<td>load</td>
<td>t4</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>sub</td>
<td>t5</td>
<td>t4</td>
</tr>
</tbody>
</table>

simple record structure with four fields

easy to reorder

explicit names
x - 2 * y

(1) load     y
(2) loadi    2
(3) mul      (1) (2)
(4) load     x
(5) sub      (4) (3)

use table index as implicit name of result
requires only three fields in record
harder to reorder
Indirect triples

\[ x - 2 \times y \]

(1) load y
(2) loadi 2
(3) mul (100) (101)
(4) load x
(5) sub (103) (102)

use *pointer* to triple as implicit name
uses more space than triples
easier to reorder
Stack machine

Simplifies IR by assuming implicit syntax

Procedures use *operand stack* for temporary values

Ex:
\[ z = x - 2 \times y \]

- load x
- push 2
- load y
- mul
- sub
- store z

Advantages:
- Compact form
- Introduced names are implicit, not explicit
- Simple to generate and execute

Disadvantages:
- Processors operate on registers, not stacks
- Difficult to reuse values on stack
- Difficult to manipulate
Examples

- P-code for Pascal
- Java bytecode
- CLR bytecode
Virtual machines

Can interpret IR using virtual machines

Examples:
- P-code for Pascal
- Java bytecode
- PostScript

Interpretation is easy and portable. But slow.

Run-time compilation.
Control flow graphs (CFG)

Models transfer of control in the procedure
- nodes in the graph are *basic blocks* (maximal-length straight-line sequences of code)
- edges in the graph represent control flow (loops, if-then-else, case, goto)

Ex:
```
n = 0
x = 1
while (n < 10) {
    n = n + 1
    x = x * n
}
print(x)
```
Useful for *data flow analyses*

- used by most intraprocedural optimizations
  - common subexpression elimination
  - loop invariant code motion
  - strength reduction
- alias analysis
- pointer analysis
- liveness analysis
  - used by register allocator
In control flow graphs, representation of each basic block is typically a 3AC representation.
Static single assignment (SSA)

- Every variable is assigned only once
- Simplifies analysis and optimization
- Translation to SSA: subscript each variable, use $\phi$-functions at join points

Ex:

```plaintext
n = 0
x = 1
while (n < 10) {
    n = n + 1
    x = x * n
}
prient(x)
```

```
\[
\begin{align*}
\phi(n_0, n_2) & = n_1 \\
\phi(x_0, x_2) & = x_1
\end{align*}
\]
```

```
\[
\begin{align*}
n_0 & = 0 \\
x_0 & = 1 \\
n_2 & = n_1 + 1 \\
x_2 & = x_1 \times n_2
\end{align*}
\]
```
Phi functions

\(\phi\) function merges multiple values incoming from different edges

One argument per incoming edge

Place \(\phi\)s at join points in the program where a variable with more than one value flows in
Undoing SSA

\[
n_0 = 0 \\
x_0 = 1 \\
n_1 = \phi(n_0, n_2) \\
x_1 = \phi(x_0, x_2) \\
n_1 < 10 ?
\]

\[
n_2 = n_1 + 1 \\
x_2 = x_1 \times n_2
\]

\[
\text{print}(x_1)
\]

Remove \(\phi\)-functions by introducing copies in previous blocks
Represent control-flow as one of a given set of patterns:
- basic block
- if-then-else
- if-then
- while
- do
- sequence
- other (represent as a control-flow graph)
Assembly language

Production compilers often use an assembly-language based IR

Assembly language instructions but with infinite registers
- sometimes in SSA form also

Very close to the machine

Suitable for machine-specific optimizations
Many kinds of IR used in practice
Best choice depends on application
No widespread agreement on the subject
Compiler may need several different IRs
Choose IR at right level of detail
Keep manipulation costs in mind
Today, we’ll talk about Piglet, the IR trees to be used in the next project

We’ll describe the translation of ASTs into IR trees

- Implementation: use a visitor!
Piglet simplifications

Piglet has a few simplifications vs. how “real” compilers work:
- Piglet does not support global or static data
- Piglet does not provide instructions for accessing the stack, just the heap – temporaries are used for local variables
- Piglet has a notion of procedures built-in, rather than just using global labels
No global data

Static or global data is supported in most “real” IR

Used for:
- global variables and constants
- string literals
- vtables (dispatch tables)
- exception tables (mapping program counter to try-block)
- type information (to implement casts, instanceof)
- jump tables (to implement switch)

The only global data in MiniJava is vtables
We’ll work around this later
Piglet procedures

add [ 3 ]
BEGIN
RETURN PLUS TEMP 1 TEMP 2
END

procedure named with add
takes 3 parameters (this, m, n)
returns m + n
Piglet for expressions

Represents a computation that returns a single value

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT i</td>
<td>An integer constant i</td>
</tr>
<tr>
<td>LABEL n</td>
<td>Symbolic constant n (a code label)</td>
</tr>
<tr>
<td>TEMP i</td>
<td>Temporary i (a “register”)</td>
</tr>
<tr>
<td>BIN e1 e2</td>
<td>Application of a binary operator to integer operands</td>
</tr>
<tr>
<td>HALLOCATE e</td>
<td>Allocate e bytes on the heap</td>
</tr>
<tr>
<td>CALL f e1 ... en</td>
<td>Procedure call</td>
</tr>
<tr>
<td>ESEQ s e</td>
<td>Expression sequence; evaluate s for side effects</td>
</tr>
</tbody>
</table>
# Differences with “real” IR trees

<table>
<thead>
<tr>
<th>NAME n</th>
<th>A global name</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL n</td>
<td>Symbolic constant n (a code label)  use NAME instead</td>
</tr>
<tr>
<td>MEM e</td>
<td>Access the contents of the address e</td>
</tr>
<tr>
<td>HALLOCATE-e</td>
<td>Allocate e bytes on the heap  use CALL to malloc instead</td>
</tr>
</tbody>
</table>
**Piglet for statements**

Represents a computation that returns no value

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE (TEMP t) e</td>
<td>Evaluate e and put result in a temporary t</td>
</tr>
<tr>
<td>HLOAD (TEMP t) e1 (INT i)</td>
<td>Evaluate e1 to an address a, load *(a+i) and put result in t</td>
</tr>
<tr>
<td>HSTORE e1 (INT i) e2</td>
<td>Evaluate e1 to an address a, store e2 in *(a+i)</td>
</tr>
<tr>
<td>EXP e</td>
<td>Evaluate e and discard result</td>
</tr>
<tr>
<td>JUMP (LABEL n)</td>
<td>Jump to code address n</td>
</tr>
<tr>
<td>CJUMP e1 (LABEL n)</td>
<td>Evaluate e1, jump to n if true; otherwise fall through</td>
</tr>
<tr>
<td>BEGIN s1 ... sn END</td>
<td>Evaluate s1, then s2, ..., then sn</td>
</tr>
<tr>
<td>LABEL n</td>
<td>Define a constant value of name n as current code address.</td>
</tr>
<tr>
<td>PRINT e</td>
<td>Evaluate integer e, print result</td>
</tr>
<tr>
<td>ERROR</td>
<td>Report an error and exit</td>
</tr>
<tr>
<td>NOP</td>
<td>Do nothing</td>
</tr>
</tbody>
</table>
### Differences with “real” IR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE (MEM e1) e2</td>
<td>Evaluate e and discard result</td>
</tr>
<tr>
<td>HLOAD (TEMP t) e1 (INT i)</td>
<td>Evaluate e1 to an address a, load *(a+i) and put result in t</td>
</tr>
<tr>
<td>HSTORE e1 (INT i) e2</td>
<td>Evaluate e1 to an address a, store e2 in *(a+i)</td>
</tr>
<tr>
<td>PRINT e</td>
<td>Evaluate integer e, print result</td>
</tr>
<tr>
<td>ERROR</td>
<td>Report an error and exit</td>
</tr>
</tbody>
</table>
Representation

You’ll be generating Piglet by building Piglet IR trees in memory.

Run piglet.jj through JTB to generate the IR tree classes.

Study jtb.out.jj to see how the trees are generated from the code.
Types of translation

$E[e]$  
- translate AST for expression $e$ into an IR expression tree

$S[e]$  
- translate expression/statement $e$ into an IR statement tree

$T[e] \rightarrow L$  
translate expression $e$ into a conditional jump—if true goto $L$, if false, fall through to the next statement

$F[e] \rightarrow L$  
similar to $T$, but jump on false
Types of translation

$E, S, T, F$ are mutually recursive
Temporaries

Use *temporaries* (TEMP) to represent local variables.

Need to fix the indexes of the temporaries representing *this* and the formal parameters.

TEMP(0) – *this*

TEMP(1), ..., TEMP(n) – the *n* formal parameters

TEMP(k), for *k* > *n*, used for intermediate results, other variables

Need to keep a map (symbol table!) from local variable name to TEMP index
Fresh variables

In each translation below, except where noted, temporaries are fresh

i.e., use the next free index (symbol table!)
Local variables

\[ E[x] = \text{TEMP}(k) \]

where \( k = \text{getLocalIndex}("x") \)

getLocalIndex uses a hash table (one per method) to record local variable indexes.

Preload with formal parameters:

- \( E[\text{this}] = \text{TEMP}(0) \)
- \( E[a_i] = \text{TEMP}(i) \)
$E[n] = \text{INT}(n)$

$E[\text{false}] = \text{INT}(0)$

$E[\text{true}] = \text{INT}(1)$
**Binary expressions**

\[
E[ e_1 + e_2 ] = \text{PLUS } E[ e_1 ] E[ e_2 ]
\]

Most other operators are similar: +, -, *, /, %, <, >, <=, >=, ==, !=

&& and || are different. Will discuss later.
Statements

Using a statement as an expression:

\[ S[ e ] = \text{EXP} \ E[ e ] \]

Evaluate \( e \) and discard the result.
\[ S[\text{if } (e) \ s] = \]
BEGIN
\[ F[\ e] \ \text{end} \]
\[ S[\ s] \]
LABEL end
END
If

\[ S[\text{if (e) } s_1 \text{ else } s_2 ] = \]

BEGIN

\[ F[ e ] L \]

\[ S[ s_1 ] \]

JUMP LABEL end

LABEL L

LABEL L

\[ S[ s_2 ] \]

LABEL end

END
While

\[ S[\text{while } (e) s] = \]
BEGIN
  JUMP LABEL bottom
  LABEL top
  \[ S[s] \]
  LABEL bottom
  \[ T[e] \]
END
While (alternate)

\[ S[\text{while (e) s}] = \]
BEGIN
  LABEL top
  \( F[e] \) end
  \( S[s] \)
  JUMP LABEL top
  LABEL end
END
Conditional jumps

\[ T[e] L = \]
CJUMP \( E[e] L \)

\[ F[e] L = \]
BEGIN
  CJUMP \( E[e] \) end
  JUMP LABEL L
  LABEL end
END
Jumps

\[ T[\text{true}] \; L = \text{JUMP} \; \text{LABEL} \; L \]

\[ T[\text{false}] \; L = \text{NOP} \]

\[ F[\text{true}] \; L = \text{NOP} \]

\[ F[\text{false}] \; L = \text{JUMP} \; \text{LABEL} \; L \]
Jumps

\[ T[!e]L = F[e]L \]

\[ F[!e]L = T[e]L \]
### && and ||

<table>
<thead>
<tr>
<th>Expression</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[ e_1 &amp;&amp; e_2 ]$ L =</td>
<td>$F[ e_1 ]$ end</td>
</tr>
<tr>
<td></td>
<td>$F[ e_2 ]$ end</td>
</tr>
<tr>
<td></td>
<td>JUMP LABEL L</td>
</tr>
<tr>
<td></td>
<td>LABEL end</td>
</tr>
<tr>
<td>$F[ e_1 &amp;&amp; e_2 ]$ L =</td>
<td>$F[ e_1 ]$ L</td>
</tr>
<tr>
<td></td>
<td>$F[ e_2 ]$ L</td>
</tr>
<tr>
<td>$T[ e_1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T[ e_2 ]$ L</td>
</tr>
<tr>
<td>$F[ e_1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T[ e_2 ]$ end</td>
</tr>
<tr>
<td></td>
<td>JUMP LABEL L</td>
</tr>
<tr>
<td></td>
<td>LABEL end</td>
</tr>
</tbody>
</table>
&& and ||

$$E[ e_1 && e_2 ] =$$

BEGIN

$$F[ e_1 && e_2 ]$$ L

MOVE (TEMP result) (INT 1)

JUMP LABEL end

LABEL L

LABEL L

MOVE (TEMP result) (INT 0)

LABEL end

END

TEMP result
Example

n = 0;
while (n < 10) {
    n = n + 1;
}

--> 
MOVE TEMP(n) CONST(0)
JUMP LABEL bottom
LABEL top
MOVE TEMP(n) (PLUS TEMP(n) INT(1))
LABEL bottom
CJUMP (LT TEMP(n) INT(10)) top
Switch (naive)

\[
S[\text{switch (x) \{ case c_1: s_1 \ldots case c_n: s_n \} }] = \text{BEGIN}
\]

\[
\text{MOVE (TEMP tx) } E[\text{x}]
\]

\[
\text{CJUMP (NE (TEMP tx) (INT c_1)) (LABEL L_2)}
\]

\[
S[\text{s_1}]
\]

LABEL L_2

\[
\ldots
\]

LABEL L_n

\[
\text{CJUMP (NE (TEMP tx) (INT c_n)) (LABEL end)}
\]

\[
S[\text{s_n}]
\]

LABEL end

\text{END}
Switch (jump table)

\[ S[\text{switch (x) \{ case c_1: s_1 \ldots case c_n: s_n \}}] = \text{BEGIN} \]

\begin{align*}
\text{MOVE (TEMP tx) } & E[ x ] \\
\text{HLOAD (TEMP tl) } & \text{JumpTable (TEMP tx)} \\
\text{JUMP (TEMP tl)} \\
\text{LABEL L_1} \\
S[ s_1 ] \\
\ldots \\
\text{LABEL L_n} \\
S[ s_n ] \\
\text{LABEL end} \\
\text{END}
\end{align*}
Array representation

Arrays are stored in the heap.

For the project, array layout is:

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
</tr>
<tr>
<td>a[1]</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>a[length-1]</td>
</tr>
</tbody>
</table>

Length of the array is stored in the first word of the object.
Every slot is 4 bytes long. (Only int[] is supported.)
We don’t need an object header since arrays don’t have methods.
Array length

Load the first word of the array

$E⟦ a.length ⟧ =

BEGIN

MOVE (TEMP ta) $E⟦ a ⟧$

NullCheck(ta)

HLOAD (TEMP result) (TEMP ta)

END

TEMP result
New array

\[ E[\text{new int}[\text{len}]] = \]

BEGIN

\begin{align*}
\text{MOVE} & \ (\text{TEMP} \ tlen) \ E[\text{len}] \\
\text{MOVE} & \ (\text{TEMP} \ tsz) \ (\text{MULT} \ (\text{PLUS} \ (\text{TEMP} \ tlen) \ (\text{INT} \ 1)) \ (\text{INT} \ 4)) \\
\text{MOVE} & \ (\text{TEMP} \ tnew) \ (\text{ALLOCATE} \ (\text{TEMP} \ tsz)) \\
\text{HSTORE} & \ (\text{TEMP} \ tnew) \ (\text{INT} \ tlen) \\
\text{ZeroBuffer} & \ ((\text{PLUS} \ (\text{TEMP} \ tnew) \ (\text{INT} \ 4)), \ (\text{TEMP} \ tlen))
\end{align*}

END

TEMP tnew

Allocates \((\text{len}+1)\times4\) bytes. Stores the length. Zeros the rest.
Zero buffer

ZeroBuffer(addr, len) =

BEGIN

  MOVE (TEMP a) addr
  MOVE (TEMP b) (PLUS (TEMP a) (MULT len (INT 4)))
  JUMP LABEL end

LABEL top

LABEL top

HSTORE (TEMP a) (INT 0)

MOVE (TEMP a) (PLUS (TEMP a) (INT 4))

LABEL end

MOVE (TEMP t) (LE (TEMP a) (TEMP b))

CJUMP (TEMP t) (LABEL top)

END
Array load

\[ E[a[i]] = \]

BEGIN

  MOVE (TEMP ta) \[ E[a] \]

  MOVE (TEMP ti) \[ E[i] \]

  NullCheck((TEMP ta))

  BoundsCheck((TEMP ta), (TEMP ti))

  HLOAD (TEMP result)

    (PLUS (TEMP ta)
     (PLUS (TEMP ti) (INT 1)) (INT 4))) (INT 0)

END

TEMP result
Bounds checks

BoundsCheck(a, i) =

BEGIN

HLOAD (TEMP tlen) a (INT 0)
MOVE (TEMP t) (LT i (INT 0))
CJUMP (TEMP t) error
MOVE (TEMP t) (GE i (TEMP tlen))
CJUMP (TEMP t) error
JUMP end
LABEL error
ERROR
LABEL end
END
Null checks

NullCheck(e) =
BEGIN
    MOVE (TEMP t) (EQ e (INT 0))
    CJUMP (TEMP t) error
    JUMP end
    LABEL error
    ERROR
    LABEL end
END
Array store

Similar to array load, but use HSTORE.
Object representation

Similar to array representation, but first word is object header

<table>
<thead>
<tr>
<th>header</th>
</tr>
</thead>
<tbody>
<tr>
<td>field1</td>
</tr>
<tr>
<td>field2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>fieldn</td>
</tr>
</tbody>
</table>

Superclass fields come first
Field access

\[ E[ e_0.f_k ] = \]

BEGIN

  MOVE (TEMP a) \( E[ e_0 ] \)

  \textit{NullCheck}(TEMP a)

  \textit{HLOAD}(TEMP \text{ result})(TEMP a)(\text{INT } k+1)

END

TEMP result

This assumes field \( f_k \) is at slot \( k+1 \) in the object (+1 is for the object header)
New object

Similar to new array, but size is fixed.

Add method to class table entry for C to compute size of an instance of C

Need to create vtable and store in first word of object.

- This should be a global (one per class), but for the project you can just create one vtable per instance.
Size of object

Need to allocate space for the header and for the fields

\[
\text{sizeof}(C) = 4 + 4 \times \text{Number Of Fields Of}(C)
\]

\[
\text{Number Of Fields Of}(C) = \# \text{ fields declared in C} + \# \text{ fields inherited}
\]
Dispatch table

- Let $m_1 \ldots m_n$ be the methods of $C$, declared or inherited (and not overridden)
- The order must be the same for all subclasses of $C.$

\[ VTable(C) = \]

BEGIN

\[
\begin{align*}
MOVE \ (TEMP \ len) \ n \\
HALLOCATE \ (TEMP \ tbl) \ (MUL \ (TEMP \ len) \ (INT \ 4)) \\
HSTORE \ (TEMP \ tbl) \ (INT \ 0) \ (LABEL \ m_1) \\
\ldots \\
HSTORE \ (TEMP \ tbl) \ (INT \ 4(n-1)) \ (LABEL \ m_n)
\end{align*}
\]

END

TEMP tbl
New object

\[ E[\text{new } C()] = \]

BEGIN

MOVE (TEMP tlen) \textit{NumberOfFieldsOf}(C)

MOVE (TEMP tsz) (MULT (PLUS (TEMP tlen) (INT 1)) (INT 4))

MOVE (TEMP tnew) (HALLOCATE (TEMP tsz))

HSTORE (TEMP tnew) \textit{VTable}(C)

\textit{ZeroBuffer}((PLUS (TEMP tnew) (INT 4)), (TEMP tlen))

END

TEMP tnew
Call and return

No explicit representation of argument passing, stack frame setup, etc.

\[ E[f(e_1, \ldots, e_n)] = \text{CALL (LABEL f)} \ E[e_1] \ldots \ E[e_n] \]

\[ S[\text{return } e] = \text{RETURN } E[e] \]
Method call

\[ E[e_0.m_k(e_1, \ldots, e_n)] = \]

BEGIN

MOVE (TEMP a) \[ E[e_0] \]

NullCheck(TEMP a)

HLOAD (TEMP tbl) (TEMP a) (INT 0)

HLOAD (TEMP m) (TEMP tbl) (INT k)

END

CALL (TEMP m) (TEMP a) \[ E[e_1] \ldots E[e_n] \]

This assumes method \( m_k \) is at slot \( k \) in the dispatch table for the static type of \( e_0 \)
Questions?