Code generation

High level languages
- Java
  - stack code
  - allocate registers to top of stack
- object-oriented
  - method invocation
  - member layout
- functional
  - higher order functions
  - function calls

Code generator generation
- tree pattern matching
- tree parsing
- peephole

Compiling Java

Class files
- structure for describing program
- machine-independent stream of bytes
- verified when loaded

Issues
- stack reduces ordering
- virtual methods reduce inlining
- multiple threads limit transformations
- verify bytecode to ensure safety

Converting into real code
- analyze stack to determine size
- represent stack as temporary variables
- try to avoid excessive copying
- allocate variables to registers

Compiling stack code

General algorithm
- determine local storage
  - max locals + max stack + max temps
- form basic blocks
- find stack height for instruction
- translate instructions

Naive approach
- map each local stack location to a frame location
- translate each instruction
- move locations between memory and registers

Register allocation approach
- map top of stack, fast locals to registers
- fixed approach maps registers for entire method
- basic block approach maps registers for basic blocks

Object-oriented (OO) languages

Objects
- a collection of data
- functions (methods) for operating on data

Classes
- collection of objects with same attributes
- organizes space of objects
- allows shared implementation of objects

Implementation
- class record
  - pointers to methods (method table)
  - storage for class data
- object record
  - pointer to class type (tag)
  - storage for local data
- location → offset in object record/method table

Class hierarchy

Inheritance
- class may inherit data/methods from another class
- ancestor class bestows attributes (superclass)
- descendant class inherits attributes (subclass)
- subclass should work whenever superclass is expected
- subclass may override methods from superclass (dynamic methods)
- multiple ancestors → multiple inheritance

Impact
- class of object not completely known at compile-time
  (since object of type subclass is allowed whenever class is allowed)
- need to test tags at runtime
- could result in non-constant data/method pointer offset

Can we eliminate overhead of data/method lookups?

Data layout optimization

Single inheritance
- ensure constant offset for fields through prelinking
- when class B inherits from class A
  - lay out fields of A at beginning of B in same order
  - place new fields of B afterwards
- field accessed as constant offset from object record

Multiple inheritance
- ensure constant offset for fields
- assign slots for field via graph coloring
  (may leave gaps between slots)
- descriptor table
  - eliminate gaps through indirection
  - assign unique descriptor slot via coloring
  - descriptor stores offsets for field
- field accessed as constant offset plus indirection
Method lookup optimization

Single inheritance
- arrange method tables entries via prefixing
- override methods by overwriting slot
- ensure constant offset for methods
- methods are executed through:
  1. fetch pointer to class record from object
  2. get function pointer at offset in method table
  3. invoke method through function pointer

Multiple inheritance
- assign slots via graph coloring
- overwrite slots as needed

Additional optimizations
- type propagation to prove class type - convert method lookup into function call
- inlining - merge code into call site, eliminates call overhead

Multiple inheritance example

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>f1()</td>
<td>a</td>
</tr>
<tr>
<td>B</td>
<td>f2()</td>
<td>b</td>
</tr>
<tr>
<td>C</td>
<td>f3()</td>
<td>c</td>
</tr>
<tr>
<td>D</td>
<td>f4()</td>
<td>d</td>
</tr>
<tr>
<td>E</td>
<td>f5()</td>
<td>f</td>
</tr>
<tr>
<td>F</td>
<td>f6()</td>
<td>g</td>
</tr>
</tbody>
</table>

Inheritance example

```java
class A extends Obj { int a; f10; }
class B extends A { int b,c; f20; } x;
x.a = x.f1();
```

Code for random

1. check x's pointer to class record
2. if (x->class == A)
   (a) call x->method[0]
   (b) assign value to x.field[0]
3. else if (x->class == B)
   (a) call x->method[1]
   (b) assign value to x.field[2]

Code for prefix

1. call x->method[0]
2. assign value to x.field[0]

Functional programming languages

Functional programming
- tries to avoid side effects (e.g., assignment)
- encourages equational reasoning
- calculate solutions to equation (e.g., λ-calculus)

Features
- emphasis on function calls, recursion
- higher order functions
  (functions used as arguments, result)
- nested functions with lexical scope

Examples

```lisp
(define FACT
 (lambda (n)
  (cond ((equal? n 1) 1)
        (lambda (x)
          [t (mult n (FACT (sub m 1)))]
          (add n x)))))
```

Compilation techniques

Higher order functions
- represent function pointers as closures
- record containing pointer to function and method to access non-local variables
- simple closure -> function & static link
- must allocate activation records on heap
- analysis to determine when variables escape
  (may be referred to by inner-nested functions)

Function calls
- tail recursion -> result of call is the return value of the parent procedure
- convert tail recursion from function call to goto
- can transform all function calls into tail recursion by adding argument for continuation (current state represented as closure)
- may also include functions

Code generator generators

Automating the process
- would like a description-based tool
- machine description + in description give code generator (eg)
- resulting code should produce great code
- resulting code should run quickly

Two major schools
- tree pattern matching
- instruction matching
Code generator generators

The big picture

```
IR optimizer code generator

machine & IR descriptions code generator tables
```

This scheme should look familiar

Tree pattern matching

Assume that the program is represented as a set of trees.

**Tree rewriting schemes** (EURE)

- machine description is
  1. mapping of subtree into single node
  2. associated code (to be emitted)

- *example pattern*:
  - $v_i \leftarrow a b$
  - \{load $r1, a; load r2, b; add r1, r1, r2\}

- paradigm is
  - find a pattern to match subtree
  - replace rhs pattern with lhs node
  - emit the associated code

Tree parsing schemes

Use LR parses

- encode pattern matching into parsing problem
  - use well understood technology
  - write grammar to describe target machine

- reductions exist code
  - attributed-style specification
  - lots of contextual knowledge available

- grammars are very ambiguous
  - reduce/reduce $\Rightarrow$ pick longest reduction
  - shift/reduce $\Rightarrow$ shift

- linear time scheme

Instruction matching

Assume program is represented in low-level intermediate representation (IR).

**Peephole optimization**

- find logically adjacent instructions that can be combined
  - use a very small context (3-10 instructions)
  - combining $i_1$ and $i_2 \Rightarrow$ faster $i_3$

- work at register-transfer language (RTL) level
  - machine description in RTL
  - low-level IR description in RTL

- using pattern matching, synthesize more complex instructions

- useful for implementing many machine-dependent optimizations

Instruction matching

Generating "peephole" code generators

- provide a one-to-one translation for IR
- add patterns to improve code
  (more complex instructions and addressing modes)

**Training generator**

- feed a set of representative programs to the trainer and let it build a table by exhaustive search
- one time expense \(\text{and it is expensive}\)
- use a linear time pattern matcher run from the tables produced by the trainer

**Typical machines**

- RT/PC w/o floating point - 70-100 instructions
- MC68020 - millions of possible instructions