Intermediary representations

Source code
Front end produces an intermediate representation

Front end

Target code

Intermediate representation

Desirable properties
- Contains sufficient information
- Easy & inexpensive to generate and manipulate
- Enables multiple independent optimizations
- Support multiple languages and architectures
- Compiler can make multiple passes over program

Advantages

Examples
- Abstract syntax tree (AST)
- Directed acyclic graph (DAG)
- Control flow graph (CFG)
- Three address code
- Stack code
- Linear address code

The IR encodes knowledge that the compiler has
derived about the source program.

The IR encodes knowledge that the compiler has
derived about the source program.

back end transforms the code in IR form into

back end transforms the code in IR form into

native code for the target machine

(native machine)

effective

optimization

optimization

Front end produces an intermediate representation

(native code)

for the program.

(native code)

for the program.

(native code)

for the program.

(native code)

for the program.

(native code)

for the program.
Intermediary representations

Broadly speaking, IRs fall into three categories:

- **Structural**
  - Structural IRs are graphically oriented.
  - Examples: directed acyclic graphs.
  -炒作是

- **Linear**
  - Large variation in level of abstraction.
  - Pseudo-code for some abstract machine.
  - Examples: control-flow graphs.

- **Hybrids**
  - Combination of graphs and linear code.
  - Examples: control-flow graphs.
  - Attempt to take best of each.

**Abstract Syntax Tree (AST)**

An abstract syntax tree (AST) is the procedure's parse tree with the nodes for most non-terminal symbols removed. For ease of manipulation, can use a linearized form of the tree.

```
(1d.x) <>(id, 2x) * (id, 2y) -> (id, x) y
```

This represents "x * 2 * y" in postfix form.

(Operator) form of the tree can use a linearized form.

This represents "x - 2 * y".
A directed acyclic graph (DAG) is an AST with a unique node for each value. The control flow graph (CFG) models the transfers of control in the procedure. Nodes in the graph are basic blocks, edges in the graph represent control flow, loops, if-then-else, case, goto, maximal-length straight-line blocks of code, and maximal-length straight-line blocks of code. The control flow graph (CFG) models the transfers of control in the procedure. A directed acyclic graph (DAG) is an AST with a unique node for each value.
Typical statement types

Three address code

Can represent three address code using quadruples

<table>
<thead>
<tr>
<th>Load</th>
<th>z</th>
<th>Load</th>
<th>y</th>
<th>Op</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Version of three address code for RISC

Advantages

- All other operands are registers
- Only load/store instructions access memory
- Names for intermediate values
- Compact form (direct naming)

Complex expressions like:

With a single operator and at most three names:

The form:

Three address code generally allow statements of
But this isn't the whole story.

Intermediate Representations

Example

Can simplify IR by assuming implicit stack

\[ z = x - 2 \cdot y \]

Disadvantages

- Difficult to reuse values on stack
- Processors operate on registers, not stacks

Advantages

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

Intermediates

- Register assignments
- Overlap information
- Storage layout

Storage map:

- Storage class, offset(s)
- Representation, type

Constant table:

- Lexical nesting depth
- Size, type, location
- Identifiers, procedures

Symbol table:

- But this isn't the whole story
Symbol tables

A symbol table associates values or attributes with names.

What should be in a symbol table? (e.g., types and values)

What information might a compiler need?

Lexical decription

When we ask about a name, we want the closest lexical definition.

How to handle nested lexical scopes?

Implementation

Symbol tables

Implementation

Symbol tables

What levels of decription?

Lexical decription

Lexical decription

Lexical decription

If array, number and size of dimensions

If procedure, number and type of parameters

Tables chained to endoshing scopes

Use one symbol table per scope

One solution

Checks endoshing scopes in order

Name lookup starts in current table if needed.

Insert names in table for current scope

Table lookup starts in current table if needed.

Checks enclosing scopes in order

C MSC /4/3/0

Lecture /1/1 , Page /1 /1

C MSC /4/3/0

Lecture /1/1 , Page /1 /2
Virtual machines

Can interpret IR using "virtual machine"

Examples
- Code for Pascal
- Postscript for display devices

Result
- easy to portable
- much slower
- Java byte code for everywhere
- Just-in-time compilation (JIT)

Just-in-time compilation time becomes execution time

...or just compile entire program

Compile section(s) to native code

And performance critical section(s)

Begin interpreting IR

Registres
- Stack pointer (SP)
- Local stack pointer (LSP)
- Program counter (PC)

Condition codes
- stores result of last conditional instruction

Execution unit
- reads current JVM instruction

Memory
- Code segment (instructions of classes)
- Constant pool (shared constant data)
- Heap (for dynamically allocated memory)
- Stack (for function call frames)
- JVM consists of four parts
- The JVM consists of four parts
- 3. Increment PC (modify FL call, branch)
- 2. Change state of virtual machine
- 1. Reads current JVM instruction

Virtual machines
Java bytecodes

Arithmetic instructions

Constant loading

Locals operations

Stack operations

Direct instructions

Branch instructions
```java
while (true) {
    { pc = new pc + instruction[code[pc]];
        { ...
            if (0 == new pc - code[pc]) break;
            case iadd:
                t1 = pop() + t2; pop();
                push(t1); break;
            case iload: push(local[code[pc] + 1]); pop();
            case istore:
                if (t = 0) new pc = code[pc + 1]; break;
            default: push(local[code[pc] + 1]);
            push(t1); break;
        }
    }
    switch (opcodes[code[pc]]) {
    case 0: new pc = pc + instruction[code[pc]];
        while (true) pc = code[start]
    }
```