Intermediate Representations

IR in compilers

- Internal representation of input program by compilers
  - Computation expressed in the input program
  - Results of program analysis
    - Control-flow graphs, data-flow graphs, dependence graphs
  - Symbol tables
    - Book-keeping information for translation (eg., types and addresses of variables and subroutines)
- External format of IR
  - Needs to be serialized
  - Allows independent passes over IR

Intermediate Representations

- Decisions in *IR* design affect the speed and efficiency of the compiler
- Some important *IR* properties
  - Ease of generation
  - Ease of manipulation
  - Procedure size
  - Freedom of expression
  - Level of abstraction
- The importance of different properties varies between compilers
- Selecting an appropriate *IR* for a compiler is critical

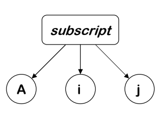
Types of Intermediate Representations

Three major categories

- Structural
  - Graphically oriented
  - Heavily used in source-to-source translators, program correctness tools
  - Tend to be large
  - Examples: Trees, DAGs
- Linear
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange
  - Examples: 3 address code, Stack machine code
- Hybrid
  - Combination of graphs and linear code
  - Example: control-flow graph

Level of Abstraction

- The level of detail exposed in an *IR* influences the profitability and feasibility of different optimizations.
- Two different representations of an array reference:



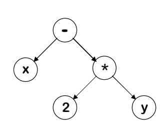
High level AST: Good for memory disambiguation

```
loadI 1        => r1
sub   rj, r1   => r2
loadI 10       => r3
mult  r2, r3   => r4
sub   ri, r1   => r5
add   r4, r5   => r6
loadI @A       => r7
Add   r7, r6   => r8
load  r8       => rAij
```

Low level linear code: Good for address calculation

Abstract Syntax Tree

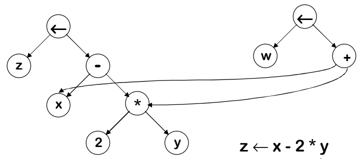
An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed



x - 2 * y

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



$z \leftarrow x - 2 * y$
$w \leftarrow x + 2 * y$

- Makes sharing explicit
- Encodes redundancy
- Same expression twice means that the compiler might arrange to evaluate it just once!

7

Stack Machine Code

Originally used for stack-based computers, now Java and C#

- Example:

  x - 2 * y =>

```
push x
push 2
push y
multiply
subtract
```

Advantages

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code
- Useful where code is transmitted over slow communication links (e.g., *the net*)

8

Three Address Code

Several different representations of three address code

- In general, three address code has statements of the form:

  x ← y <u>op</u> z

  With 1 operator (<u>op</u>) and, at most, 3 names (x, y, & z)

Example:

$z \leftarrow x - 2 * y$ => $t_1 \leftarrow 2 * y$
$z \leftarrow x - t_1$

Advantages:

- Resembles many machines
- Introduces a new set of names (the temp results)
- Compact form

9

Three Address Code: Quadruples

Naïve representation of three address code

- Table of k * 4 small integers
- Simple record structure
- Easy to reorder
- Explicit names

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

RISC assembly code

load	1	Y	
loadi	2	2	
mult	3	2	1
load	4	X	
sub	5	4	2

Quadruples

10

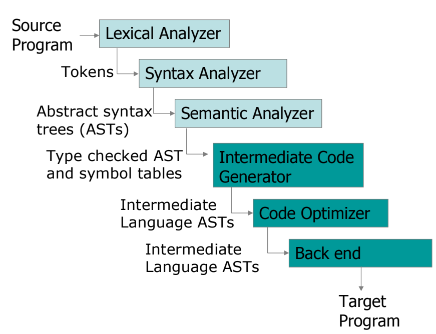
Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

(1)	load	y	
(2)	load	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

11

Implementation of MiniJava Compiler



12

Symbol Tables

- After ASTs have been constructed, the compiler must check whether the input program is type-correct. During this type checking, a compiler checks whether the use of names (such as variables, functions, type names) is consistent with their definition in the program.
- Consequently, it is necessary to remember declarations so that we can detect inconsistencies and misuses during type checking. This is the task of a *symbol table*.

13

Symbol Table Entries

- What information do we need to put in an entry for a variable in a Symbol Table?

14

Symbol Table Entries

- What information do we need to put in an entry for a variable in a Symbol Table?
- Some obvious choices:
  - Name
  - Type
  - Array? (then dimension information)
  - Line Number (used in reporting errors)
  - Scope (so we know when to deactivate it)
  - Initialized? (for compile-time error checking)
  - Memory Position (for compiling to Assembly)
  - Others if we we're interpreting the code

15

Symbol Table Design

- Several data structures can be used for a symbol table.
  - Arrays
  - Linked Lists
  - Binary Tree
  - Hash Table
  - Hybrids
- Which are the best choices? Consider:
  - Memory used
  - Cost to Insert()
  - Cost to LookUp()

16

Symbol Table Design

- Most compilers use
  - Hash table
    - Hash is often a simple function of symbol string
  - Each Hash Bucket has a linked list to resolve conflicts
- Our MiniJava compiler uses such a system

17

The Rest of the Story…

Representing the code is only part of an *IR*

There are other necessary components:

- Symbol table (already discussed)
- Constant table
  - Representation, type
  - Storage class, offset
- Storage map
  - Overall storage layout
  - Overlap information
  - Virtual register assignments
- Others?

18