Compiler Construction

Chapter 4: Intermediate representations

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Outline

Introduction

Intermediate representation (IR)

- Representation of the code
- Derives many facts that have no explicit representation in source code
  - E.g. memory addresses of variables and constants
- A symbol table is also a part of IR
- IR choices depend on the nature of the source and the target languages

Taxonomy of IRs

Structural organization

- Graphical IRs e.g. a parse tree
- Linear IRs e.g. ILOC code
- Hybrid IRs e.g. a control-flow graph (CFG)

Level of abstraction

- Near-source representation e.g. a tree
- Low-level representation e.g. ILOC code

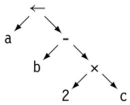
Mode of use

- Definitive IR is the primary representation for the code.
- Derivative IR is built for a specific, temporary purpose.

Naming - namespace

- Names and storage locations

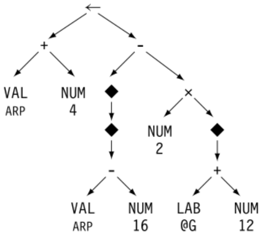
Examples of IRs



(a) AST for $a \leftarrow b - 2 \times c$

Op	Arg$_1$	Arg$_2$	Result
×	2	c	t
-	b	t	a

(b) Quadruples for $a \leftarrow b - 2 \times c$



(c) Low-Level AST

$$
\begin{array}{lll}
t_0 & \leftarrow & r_{arp} - 16 \\
t_1 & \leftarrow & \blacklozenge\ t_0 \\
t_2 & \leftarrow & \blacklozenge\ t_1 \\
t_3 & \leftarrow & @G \\
t_4 & \leftarrow & t_3 + 12 \\
t_5 & \leftarrow & \blacklozenge\ t_4 \\
t_6 & \leftarrow & t_5 \times 2 \\
t_7 & \leftarrow & t_2 - t_6 \\
t_8 & \leftarrow & r_{arp} + 4 \\
\blacklozenge\ t_8 & \leftarrow & t_7
\end{array}
$$

(d) Low-Level Linear Code

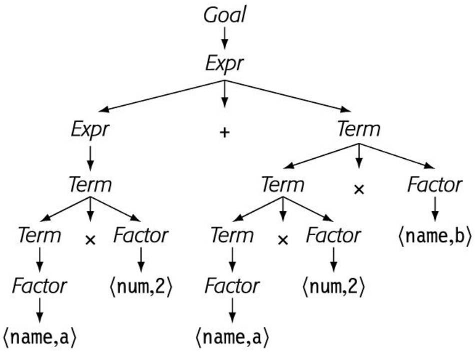
■ **FIGURE 4.1** Different Representations for $a \leftarrow b - 2 \times c$.

Outline

Parse tree

$$
\begin{array}{lcl}
\textit{Goal} & \rightarrow & \textit{Expr} \\
\textit{Expr} & \rightarrow & \textit{Expr} + \textit{Term} \\
 & | & \textit{Expr} - \textit{Term} \\
 & | & \textit{Term} \\
\textit{Term} & \rightarrow & \textit{Term} \times \textit{Factor} \\
 & | & \textit{Term} \div \textit{Factor} \\
 & | & \textit{Factor} \\
\textit{Factor} & \rightarrow & \underline{(}\ \textit{Expr}\ \underline{)} \\
 & | & \texttt{num} \\
 & | & \texttt{name}
\end{array}
$$

(a) Classic Expression Grammar



(b) Parse Tree for a × 2 + a × 2 × b

■ **FIGURE 4.2** Parse Tree for a × 2 + a × 2 × b.

Syntax-related tress

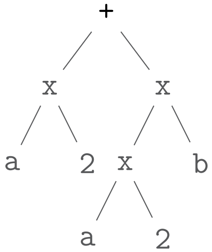
- Parse trees
  - Mostly used in parsing and attribute grammar
- Abstract syntax tree
  - A compact parse tree with no nonterminal node.
- Directed acyclic graphs (DAGs)

Abstract syntax tree (AST)

An AST is a contraction of the parse tree that omits most nodes for nonterminal symbols.

- Source-to-source translation
- Low-level AST is also a source to generate assembly codes

$a \times 2 + a \times 2 \times b$

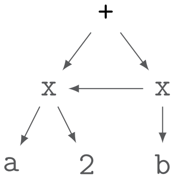


Directed acyclic graph (DAG)

A DAG is an AST with sharing. Identical subtrees are instantiated once, with multiple parents.

- Reduce memory footprint
- Identify redundancies

`a x 2 + a x 2 x b`



Control-flow graph

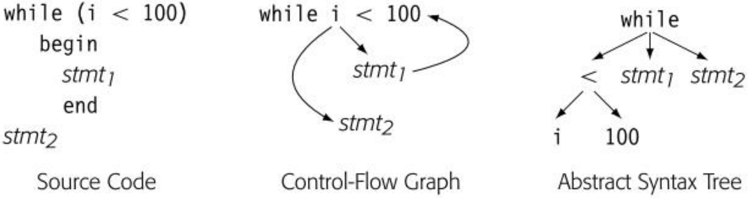
Basic block

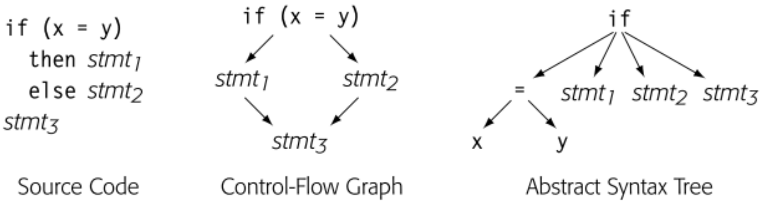
A basic block is a maximal-length sequence of branch-free code. It begins with a labelled operation and ends with a branch, jump, or predicated operation.

Control-flow graph

A control-flow graph has a node for every basic block and an edge for each possible control transfer between blocks.

Control-flow graph





Control-flow graph

Derivative IR

- A graph represents relationships among blocks
- Operation inside a block are represented with another IR, such as AST, DAG, linear IRs
- Use for optimization

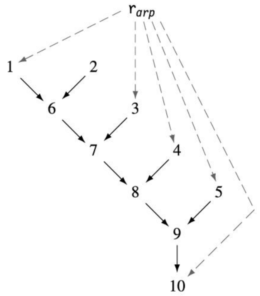
Although a basic block is a maximal-length sequence of branch-free code, we may build a blocks that are shorter, e.g. a single-statement block.

Dependence graph

Data-dependence graph - partial ordering on execution

1	loadAI	r_{arp}, @a	$\Rightarrow$	r_a
2	loadI	2	$\Rightarrow$	r_2
3	loadAI	r_{arp}, @b	$\Rightarrow$	r_b
4	loadAI	r_{arp}, @c	$\Rightarrow$	r_c
5	loadAI	r_{arp}, @d	$\Rightarrow$	r_d
6	mult	r_a, r_2	$\Rightarrow$	r_a
7	mult	r_a, r_b	$\Rightarrow$	r_a
8	mult	r_a, r_c	$\Rightarrow$	r_a
9	mult	r_a, r_d	$\Rightarrow$	r_a
10	storeAI	r_a	$\Rightarrow$	r_{arp}, @a

(a) Example Code from Chapter 1



(b) Dependence Graph for the Example

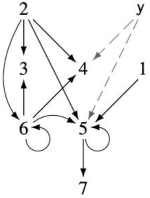
■ **FIGURE 4.3** An ILOC Basic Block and Its Dependence Graph.

Chula
Chulalongkorn University

Control-flow graph and dependence graph

```
x ← 0
i ← 1
while (i < 100)
  if (y[i] > 0)
    then x ← x + y[i]
  i ← i + 1
print x
```

(a) The Code



(b) Its Dependence Graph

■ **FIGURE 4.4** Interaction Control Between Flow and the Dependence Graph.

What are benefits from this analysis?

Call graph

A graph that represents each distinct procedural call.

- To perform inter-procedure and intra-procedure analysis

Possible complications

- External function calls
- Procedure-valued parameters e.g. callback functions
- Inheritance, i.e. different method implementations

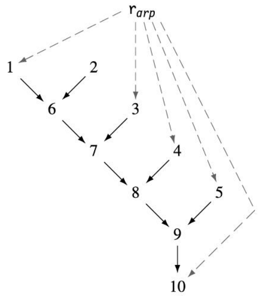
Outline

Linear IR

An ordered series of operations

1	loadAI	r_{arp}, @a	$\Rightarrow$	r_a
2	loadI	2	$\Rightarrow$	r_2
3	loadAI	r_{arp}, @b	$\Rightarrow$	r_b
4	loadAI	r_{arp}, @c	$\Rightarrow$	r_c
5	loadAI	r_{arp}, @d	$\Rightarrow$	r_d
6	mult	r_a, r_2	$\Rightarrow$	r_a
7	mult	r_a, r_b	$\Rightarrow$	r_a
8	mult	r_a, r_c	$\Rightarrow$	r_a
9	mult	r_a, r_d	$\Rightarrow$	r_a
10	storeAI	r_a	$\Rightarrow$	r_{arp}, @a

(a) Example Code from Chapter 1



(b) Dependence Graph for the Example

■ FIGURE 4.3 An ILOC Basic Block and Its Dependence Graph.

ILOC code has an implicit total order (those line numbers); the dependence graph imposes a partial order (graph direction) that allows multiple execution orders

Stack-machine code

One-address codes: top-of-stack address

- Take operands from the stack
- Push the result back onto the stack

E.g. `a - 2 x b`

```
push    2
push    b
multiply
push    a
subtract
```

- Compact (memory), with few names
- Java's bytecode is similar to stack-machine code.

Three-address code

`i <- j op k`

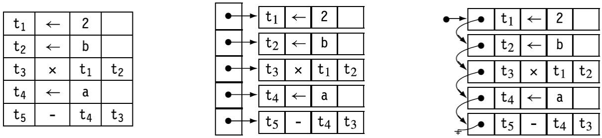
- Two addresses for operands
- One address for result

E.g. `a - 2 x b`

```
t1 <- 2
t2 <- b
t3 <- t1 x t2
t4 <- a
t5 <- t4 - t3
```

- Compact
- Most operations consists of an opcode and three names

Linear IR implementation



(a) Simple Array (b) Array of Pointers (c) Linked List

■ **FIGURE 4.5** Implementations of Three-Address Code for a - 2 × b.

- Load/store operations
- Code moving during optimization
- Storage requirement

Building a control-flow graph

Chulalongkorn University

```
FindLeaders()
    next ← 1
    Leader[next++] ← 1
    create a CFG node for l_1

    for i ← 2 to n do
        if op_i has a label l_i then
            Leader[next++] ← i
            create a CFG node for l_i

    // MaxStmt is a global variable
    MaxStmt ← next - 1
```

(a) Finding Leaders

```
BuildGraph()
    for i ← 1 to MaxStmt do
        j ← Leader[i] + 1
        while (j ≤ n and op_j ∉ Leader) do
            j ← j + 1

        j ← j - 1
        Last[i] ← j

        if op_j is "cbr r_k → l_1, l_2" then
            add edge from j to node for l_1
            add edge from j to node for l_2

        else if op_j is "jumpI → l_1" then
            add edge from j to node for l_1

        else if op_j is "jump → r_1" then
            add edges from j to all labeled
                statements
        end
```

(b) Finding Last and Adding Edges

FIGURE 4.6 Building a Control-Flow Graph.

Complications in CFG construction

- Ambiguous jumps
- Fall-through branch
- PC (program counter) related branch
- Branch delay slots
- Inter-procedural jumps

Exercise: Building a control-flow graph

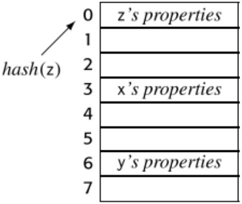
```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Outline

Symbol tables

A symbol table is a collection of information about names and values

- E.g., data type, size, storage location
- Map from textual name to an index in a repository
- A repository where index leads to the names' property



A compiler may use multiple tables to represent different kinds of information about different kinds of values.

Name resolution

If we found a reference to a name `n` at point `p` in a program

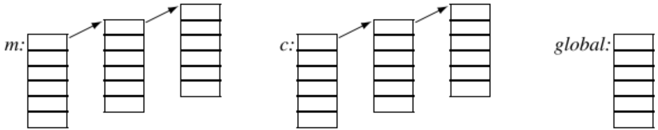
- We have to map `n` back to its declaration in the naming environment that holds `p`

Scope: the region of a program where a given name can be accessed

- Lexical scopes: nested regions of codes
- Instance hierarchies: superclass and subclass

Hierarchical tables

Look for name m



Lexical Tables for m Inheritance Tables for c Global Table

- These tables are also useful for performance monitoring and debugging.
- Some language constructs e.g. records, structs and objects act as independent scope.

Table implementation

Data structure

- Linear list
- (Balanced) Tree
- Hash map
- Static map e.g. DFA similar to a a scanner.

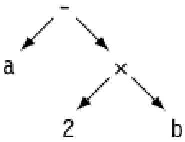
Some common properties

- Record (`struct`) storage should be either contiguous or block-contiguous
- Each repository (table) should contain enough information to rebuild the lookup structure
- The repository should support changes to the search path e.g. the parser moves in and out of different scopes

Outline

Name spaces in the IR

Implicit vs explicit names



AST for a - 2 × b

load	@b		$\Rightarrow r_0$
multI	2,	r_0	$\Rightarrow r_1$
load	@a		$\Rightarrow r_2$
subI	r_2,	r_1	$\Rightarrow r_3$

ILOC for a - 2 × b
With Unique Names

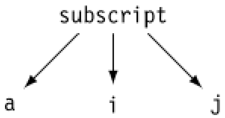
load	@b		$\Rightarrow r_0$
multI	2,	r_0	$\Rightarrow r_1$
load	@a		$\Rightarrow r_0$
subI	r_0,	r_1	$\Rightarrow r_1$

ILOC for a - 2 × b
With Name Reuse

Name spaces in the IR

Variables versus values

- $a \leftarrow 2 * b + cos(c/3)$
  - a, b, c can be used in subsequent statements
  - $2 * b$, $c/3$, $cos(c/3)$ cannot
- If evaluated expressions have their own unique name, we may be able to reuse them



Source-Level Tree

subI	r_i, 1	$\Rightarrow$	r_1
multI	r_1, 10	$\Rightarrow$	r_2
subI	r_j, 1	$\Rightarrow$	r_3
add	r_2, r_3	$\Rightarrow$	r_4
multI	r_4, 4	$\Rightarrow$	r_5
loadI	@a	$\Rightarrow$	r_6
add	r_5, r_6	$\Rightarrow$	r_7
load	r_7	$\Rightarrow$	r_{aij}

ILOC Code

ILOC code provides explicit names for each subexpression in the calculation

Naming temporary values

$$
\begin{array}{lcl}
a & \leftarrow & b + c \\
b & \leftarrow & a - d \\
c & \leftarrow & b + c \\
d & \leftarrow & a - d
\end{array}
$$

(a) Source Code

$$
\begin{array}{lcl}
t_1 & \leftarrow & b \\
t_2 & \leftarrow & c \\
t_3 & \leftarrow & t_1 + t_2 \\
a & \leftarrow & t_3 \\
t_4 & \leftarrow & d \\
t_1 & \leftarrow & t_3 - t_4 \\
b & \leftarrow & t_1 \\
t_2 & \leftarrow & t_1 + t_2 \\
c & \leftarrow & t_2 \\
t_4 & \leftarrow & t_3 - t_4 \\
d & \leftarrow & t_4
\end{array}
$$

(b) Source Names

$$
\begin{array}{lcl}
t_1 & \leftarrow & b \\
t_2 & \leftarrow & c \\
t_3 & \leftarrow & t_1 + t_2 \\
a & \leftarrow & t_3 \\
t_4 & \leftarrow & d \\
t_5 & \leftarrow & t_3 - t_4 \\
b & \leftarrow & t_5 \\
t_6 & \leftarrow & t_5 + t_2 \\
c & \leftarrow & t_6 \\
t_5 & \leftarrow & t_3 - t_4 \\
d & \leftarrow & t_5
\end{array}
$$

(c) Value Names

Static single-assignment form (SSA)

SSA is an IR that has a value-based name system, created by renaming and use of pseudo-operations called ϕ-functions. SSA encodes both control and value flow.

- Each definition has a distinct name.
- Each use refer to a single definition.
- A ϕ-function is inserted at a point where different control-flow paths merges, and defines a new name.
  - All ϕ functions run in parallel at the beginning of the block

A ϕ function takes an arbitrary number of operands

```
x ← ···
y ← ···
while (x < 100)
  x ← x + 1
  y ← y + x
```

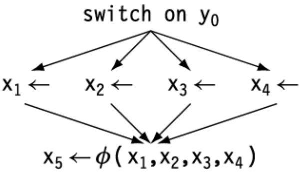
(a) Original Code

```
       x0 ← ···
       y0 ← ···
       if (x0 ≥ 100) goto next
loop:  x1 ← φ(x0,x2)
       y1 ← φ(y0,y2)
       x2 ← x1 + 1
       y2 ← y1 + x2
       if (x2 < 100) goto loop
next:  x3 ← φ(x0,x2)
       y3 ← φ(y0,y2)
```

(b) Code in SSA Form

FIGURE 4.7 A Small Loop in SSA Form.

SSA



ϕ-Function at the End of a Case Statement

Outline

Placement of values in memory

- Physical register
- Memory in a specific location `<base address, offest>`
- Virtual register
- Symbolic label

The location's lifetime must match the lifetime of the value.

Memory models

1. Register-to-register
   - Keep values in registers unless explicitly specified
   - Fast computation
   - May require a large numbers of registers
2. Memory-to-memory
   - Move values to registers just before they are used
   - Store values back to memory just after they are defined
   - Require few registers
   - May have several unnecessary load/store
3. Stack model

```
add @a, @b ⇒ @c

load  @a        ⇒ vr_i
load  @b        ⇒ vr_j
add   vr_i, vr_j ⇒ vr_k
store vr_k      ⇒ @c
```

(a) Memory-to-Memory Model

```
add vr_a, vr_b ⇒ vr_c
```

(b) Register-to-Register Model

```
push @b
push @a
add
pop  ⇒ @c
```

(c) Stack Model

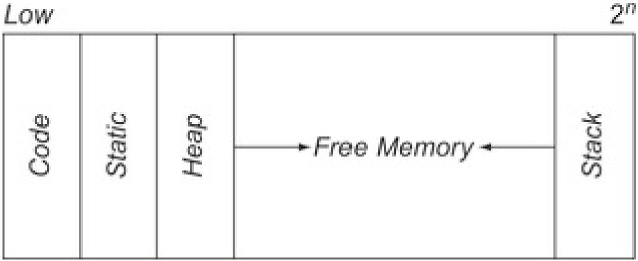
FIGURE 4.8 Three-Operand Add Under Different Memory Models.

Keeping values in registers

Register-to-register memory model

- Map (unlimited numbers of) virtual registers to physical registers
- Spill data in any virtual register that cannot be kept in a physical register

Memory layout



Assigning values to data areas

Storage during runtime depends on the following properties

- Lifetime
- Region of visibility
- Declaring scope

	Scope	Lifetime	Location
ALLs	Local	Automatic	Registers or local data area of declaring scope
	Local	Static	Procedure or file static data area
	File	Static	File static data area
	Global	Static	Global data area
OOLs	Method	Automatic	Registers or local data area of declaring scope
	Method	Static	Class record or method-specific static data area
	Class	Static	Class record
	Global	Static	Global data area
	any scope	Irregular	Explicitly allocated on the heap

■ **FIGURE 4.9** Variable Placement by Scope and Lifetime.