Intermediate Code Generation

Ronghui Gu
Spring 2019
Columbia University

* Course website: https://www.cs.columbia.edu/~rgu/courses/4115/spring2019
** These slides are borrowed from Prof. Edwards.
Intermediate Code Generation

int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

0101110101...
Intermediate Code Generation

Intermediate Representation (IR):

- An abstract machine language
- Not specific to any particular machine
- Independent of source language

IR code generation is not necessary:

- Semantic analysis phase can generate assembly code directly.
- Hinders portability and modularity.
Suppose we wish to build compilers for \( n \) source languages and \( m \) target machines.

**Case 1: no IR.** Need \( n \times m \) compilers.
Suppose we wish to build compilers for \( n \) source languages and \( m \) target machines.

**Case 2: IR present.** Need just \( n \) front-ends and \( m \) back ends.
IR properties

- Must be convenient for semantic analysis phase to produce.
- Must be convenient to translate into real assembly code for all desired target machines.
Intermediate Representations/Formats
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}

Method int gcd(int, int)
  0 goto 19
  3 iload_1   // Push a
  4 iload_2   // Push b
  5 if_icmple 15 // if a <= b goto 15
     8 iload_1   // Push a
     9 iload_2   // Push b
   10 isub      // a - b
   11 istore_1  // Store new a
  12 goto 19
   15 iload_2  // Push b
  16 iload_1   // Push a
  17 isub      // b - a
  18 istore_2  // Store new b
  19 iload_1   // Push a
  20 iload_2   // Push b
  21 if_icmpne 3 // if a != b goto 3
  24 iload_1   // Push a
  25 ireturn   // Return a
Stack-Based IR: Bitcoin Script
Stack-Based IR: Bitcoin Script

Stack

PUSH

DUP HASH160 EQUALVERIFY CHECKSIG
Stack-Based IR: Bitcoin Script

![Diagram of Bitcoin script components including DUP, HASH160, EQUALVERIFY, and CHECKSIG operations on a stack.](image-url)
Stack-Based IR: Bitcoin Script

`POP` -> Stack

Stack:

- Key
- Key

 operations:

- HASH160
- EQUALVERIFY
- CHECKSIG
Stack-Based IR: Bitcoin Script
Stack-Based IR: Bitcoin Script
Stack-Based IRs

**Advantages:**

- Trivial translation of expressions
- Trivial interpreters
- No problems with exhausting registers
- Often compact

**Disadvantages:**

- Semantic gap between stack operations and modern register machines
- Hard to see what communicates with what
- Difficult representation for optimization
```c
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
    return a;
}
```

```c
sne $vr1.s32 <- gcd.a,gcd.b
seq $vr0.s32 <- $vr1.s32,0
btrue $vr0.s32,gcd._gcdTmp1 // if !(a != b) goto Tmp1

sl $vr3.s32 <- gcd.b,gcd.a
seq $vr2.s32 <- $vr3.s32,0
btrue $vr2.s32,gcd._gcdTmp4 // if !(a < b) goto Tmp4

mrk 2, 4 // Line number 4
sub $vr4.s32 <- gcd.a,gcd.b
mov gcd._gcdTmp2 <- $vr4.s32
mov gcd.a <- gcd._gcdTmp2 // a = a - b
jmp gcd._gcdTmp5

gcd._gcdTmp4:

mrk 2, 6
sub $vr5.s32 <- gcd.b,gcd.a
mov gcd._gcdTmp3 <- $vr5.s32
mov gcd.b <- gcd._gcdTmp3 // b = b - a

gcd._gcdTmp5:

jmp gcd._gcdTmp0

gcd._gcdTmp1:

mrk 2, 8
ret gcd.a // Return a
```
Register-Based IRs

Most common type of IR

Advantages:

• Better representation for register machines
• Dataflow is usually clear

Disadvantages:

• Slightly harder to synthesize from code
• Less compact
• More complicated to interpret
Most register-based IRs use **three-address code**: Arithmetic instructions have (up to) three operands: two sources and one destination.

**SSA Form**: each variable in an IR is assigned exactly once

**C code**: ```
int gcd(int a, int b) {
    while (a != b)
        if (a < b)
            b -= a;
        else
            a -= b;
    return a;
}
```  

**Three-Address**: ```
WHILE: 
    t = sne a, b
    bz DONE, t
    t = slt a, b
    bz ELSE, t
    b = sub b, a
    jmp LOOP
ELSE: 
a = sub a, b
LOOP: 
    jmp WHILE
DONE: 
    ret a
```  

**SSA**: ```
WHILE: 
    t1 = sne a1, b1
    bz DONE, t1
    t2 = slt a1, b1
    bz ELSE, t2
    b1 = sub b1, a1
    jmp LOOP
ELSE: 
a1 = sub a1, b1
LOOP: 
    jmp WHILE
DONE: 
    ret a1
```
Three-Address Code
What is an “Address” in Three-Address Code?

- **Name**: (from the source program) e.g., x, y, z
- **Constant**: (with explicit primitive type) e.g., 1, 2, ’a’
- **Compiler-generated temporary**: (”register”) e.g., t1, t2, t3
Instructions of Three-Address Code

• $x = \text{op} \ y, \ z$: where op is a binary operation
• $x = \text{op} \ y$: where op is a unary operation
• $x = y$: copy operation
• jmp L: unconditional jump to label L
• bz L, x: jump to L if x is zero
• bnz L, x: jump to L if x is not zero
• param x, call L, y, return z: function calls
**Goal**: take statements (AST) and produce a sequence of TAC.

**Example**:

\[ a := b + c \times d; \]

**TAC**:

\[ t1 = \text{mul} \ c, \ d \]
\[ t2 = \text{add} \ b, \ t1 \]
\[ a = t1 \]

Translate *expressions* and *statements*
Translating Expressions
Example

\[ b + c * d \]

Diagram:

```
  +
 /|
/  |
/   |
b   *
  /
 /|
/c  |d
```
Example

\[ b + c \times d \]

\[
\begin{array}{c}
+ \\
| \\
| \\
b \\
| \\
t1:* \\
| \\
c \quad d
\end{array}
\]

t1 = mul c, d
Example

```
b + c * d
```

```
t2: +
  /
 b  t1: *
     /
    c  d
```
Algorithm: Syntax-Directed Translation (SDT)

For each expression E, we’ll synthesize two attributes:

- **E.addr**: the name of the variable (often a temporary variable)
- **E.code**: the IR instructions generated from E

**SDT**: each semantic rule corresponds to **actions computing two attributes** with the following auxiliary functions:

- Call **NewTemp** to create a new temporary variable
- Call **Gen**: to print a new three-address instruction
  
  Gen(t, "=" , op, x, ",", y)  \Rightarrow  "t = op x, y"
CFG rule: $E_0 \rightarrow \text{id}$

Actions:

- $E_0.\text{addr} := \text{id}$
- $E_0.\text{code} := ""$  empty string

*We do not consider scopes here.*

Example: $E_0 = \text{ID("a")}$

- $E_0.\text{addr} := \text{"a"}$
- $E_0.\text{code} := ""$  empty string
Syntax-Directed Translation (SDT)

CFG rule: \[ E_0 \rightarrow E_1 + E_2 \]

Actions:
\[ E_0.\text{addr} := \text{NewTemp()} \]
\[ E_0.\text{code} := E_1.\text{code} \text{ || } E_2.\text{code} \text{ ||} \]
\[ \text{Gen}(E_0.\text{addr}, ",\text{"}, \text{"add"}, E_1.\text{addr}, ",\text{"}, E_2.\text{addr}) \]

Example: \(a + b\)
\[ E_0 = \text{PLUS}(E_1, E_2) \quad E_1 = \text{ID(\"a\")} \quad E_2 = \text{ID(\"b\")} \]
\[ E_1.\text{addr} := \text{\"a\"} \quad E_1.\text{code} := \text{\"\"} \]
\[ E_2.\text{addr} := \text{\"b\"} \quad E_2.\text{code} := \text{\"\"} \]
\[ E_0.\text{addr} := \text{\"t1\"} \]
\[ E_0.\text{code} := \text{\"t1 = add a, b\"} \]
Example: \( b + c \times d \)

\[
E_0 = \text{PLUS} (E_1, E_2) \quad E_1 = \text{ID}("b") \\
E_2 = \text{MUL} (\text{ID}("c"), \text{ID}("d")) \\
E_0.\text{code} := E_1.\text{code} \| E_2.\text{code} \| \\
\text{Gen}(E_0.\text{addr}, "="", "add", E_1.\text{addr}, ",", E_2.\text{addr})
\]
Syntax-Directed Translation (SDT)

Example: \( b + c \times d \)

\[
E_0 = \text{PLUS} (E_1, E_2) \quad E_1 = \text{ID}("b") \\
E_2 = \text{MUL} (\text{ID}("c"), \text{ID}("d"))
\]

\[
E_0.\text{code} := E_1.\text{code} \parallel E_2.\text{code} \parallel \\
\text{Gen}(E_0.\text{addr}, ",", \text{ADD}, E_1.\text{addr}, ",", E_2.\text{addr})
\]
Syntax-Directed Translation (SDT)

Example: $b + c \times d$

\[ E_0 = \text{PLUS} \ (E_1, E_2) \quad E_1 = \text{ID}(“b”) \]
\[ E_2 = \text{MUL} \ (\text{ID}(“c”), \text{ID}(“d”)) \]
\[ E_0.\text{code} := “” \ || \ E_2.\text{code} \ || \]
\[ \text{Gen}(E_0.\text{addr}, “=”, “add”, E_1.\text{addr}, “,”, E_2.\text{addr}) \]
\[ E_1.\text{addr} = “b” \]
Syntax-Directed Translation (SDT)

Example: \( b + c \times d \)

\[
E_0 = \text{PLUS} (E_1, E_2) \quad E_1 = \text{ID}("b") \\
E_2 = \text{MUL} (\text{ID}("c"), \text{ID}("d"))
\]

\[
E_0.\text{code} := "" || "t1 = \text{mul} c, d" || \\
\text{Gen}(E_0.\text{addr}, "\text{=}", "\text{add}", E_1.\text{addr}, ",", E_2.\text{addr})
\]

\[
E_1.\text{addr} = "b" \quad E_2.\text{addr} = "t1"
\]
Example: $b + c \ast d$

\[
E_0 = \text{PLUS} \left( E_1, E_2 \right) \quad E_1 = \text{ID}("b")
\]

\[
E_2 = \text{MUL} \left( \text{ID}("c"), \text{ID}("d") \right)
\]

\[
E_0.\text{code} := "" \|| "t1 = \text{mul} c, d"\||
\]

\[
\text{Gen}(\text{NewTemp()}, "=", "\text{add}", E_1.\text{addr}, ",", E_2.\text{addr})
\]

\[
E_1.\text{addr} = "b" \quad E_2.\text{addr} = "t1"
\]
Syntax-Directed Translation (SDT)

Example: \( b + c \times d \)

\[
E_0 = \text{PLUS} (E_1, E_2) \quad E_1 = \text{ID}("b") \\
E_2 = \text{MUL} (\text{ID}("c"), \text{ID}("d"))
\]

\[
E_0.\text{code} := "\" || "t1 = \text{mul} c, d" || \\
\text{Gen("t2", ";", "add", } E_1.\text{addr}, ";", E_2.\text{addr})
\]

\[
E_1.\text{addr} = "b" \quad E_2.\text{addr} = "t1"
\]
**Example:** \( b + c \times d \)

\[
E_0 = PLUS (E_1, E_2) \quad E_1 = \text{ID}("b") \\
E_2 = \text{MUL} (\text{ID}("c"), \text{ID}("d"))
\]

\[
E_0\text{.code} := "" || "t1 = mul c, d" || \\
\text{Gen("t2", ",", "add", "b", ",", "t1")}
\]
Example: $b + c \times d$

$E_0 = \text{PLUS}(E_1, E_2) \quad E_1 = \text{ID}(\text{"b"})$

$E_2 = \text{MUL}(\text{ID}(\text{"c"}), \text{ID}(\text{"d"}))$

$E_0\.code := \text{"” || "t1 = mul c, d” || "t2 = add b, t1”}$
Translating Statements
CFG rule: \( S \rightarrow \text{id} := E \)

Actions:
\[
S\.code := E\.code \ || \ Gen(\text{id}, "=", E\.addr)
\]

Example: \( a := b + c \)
\[
S = \text{ASG(\text{ID("a")}, E)} \quad E = \text{PLUS(\text{ID("b")}, \text{ID("c")})}
\]
\[
E\.code := "t1 = add b, c" \quad E\.addr := "t1"
\]
\[
S\.code := "t1 = add b, c" \ || \ "a = t1"
\]
IF Statement

**AST:** IF($E$, $S$)

**Generated IR:**

```plaintext
E.code
bz Label_End, E.addr
S.code
Label_End:
```

**Example:** if (a > b) { a -= b }

```plaintext
t1 = slt a, b
bz Label_End, t1
a = sub a, b
Label_End:
```
**IF-ELSE Statement**

**AST**: `IFELSE(E, S_1, S_2)`

**Generated IR**: 

```plaintext
E.code
bz Label_Else, E.addr
S_1.code
jmp Label_End

Label_Else:
S_2.code

Label_End:
```
Loop

**AST:** \( \text{WHILE}(E, S) \)

**Generated IR:**

\[
\text{Label\_While:} \\
E.\text{code} \\
bz \text{Label\_End}, E.\text{addr} \\
S.\text{code} \\
\text{jmp Label\_While} \\
\text{Label\_End:}
\]
A **Basic Block** is a sequence of IR instructions with two properties:

1. The first instruction is the only entry point (no other branches in; can only start at the beginning)
2. Only the last instruction may affect control (no other branches out)

∴ If any instruction in a basic block runs, they all do

Typically “arithmetic and memory instructions, then branch”

**ENTER:**
```
t2 = add t1, 1
```
```
t3 = slt t2, 10
```
```
bz NEXT, t3
```
Basic Blocks and Control-Flow Graphs

WHILE:

- \( t1 = \text{sne} \ a1, \ b1 \)
- \( \text{bz} \ \text{DONE}, \ t1 \)
- \( t2 = \text{slt} \ a1, \ b1 \)
- \( \text{bz} \ \text{ELSE}, \ t2 \)
- \( b1 = \text{sub} \ b1, \ a1 \)
- \( \text{jmp} \ \text{LOOP} \)

ELSE:
- \( a1 = \text{sub} \ a1, \ b1 \)

LOOP:
- \( \text{jmp} \ \text{WHILE} \)

DONE:
- \( \text{ret} \ a1 \)

- Leaders: branch targets & after conditional branch
Basic Blocks and Control-Flow Graphs

WHILE:  
  \( t_1 = \text{sne} \ a_1, \ b_1 \)
  \( \text{bz} \ \text{DONE,} \ t_1 \)
  \( t_2 = \text{slt} \ a_1, \ b_1 \)
  \( \text{bz} \ \text{ELSE,} \ t_2 \)
  \( b_1 = \text{sub} \ b_1, \ a_1 \)
  \( \text{jmp} \ \text{LOOP} \)

ELSE:  
  \( a_1 = \text{sub} \ a_1, \ b_1 \)

LOOP:  
  \( \text{jmp} \ \text{WHILE} \)

DONE:  
  \( \text{ret} \ a_1 \)

- Leaders: branch targets & after conditional branch
- Basic blocks: start at a leader; end before next
Basic Blocks and Control-Flow Graphs

- Leaders: branch targets & after conditional branch
- Basic blocks: start at a leader; end before next
- Basic Blocks are nodes of the Control-Flow Graph
Three-address code instructions; Static single-assignment; Explicit control-flow graph; Local names start with %; Types throughout; User-defined functions

```c
int add(int x, int y)
{
    return x + y;
}
```

```llvm
define i32 @add(i32 %x, i32 %y) {
  entry:
  %x1 = alloca i32
  store i32 %x, i32* %x1
  %y2 = alloca i32
  store i32 %y, i32* %y2
  %x3 = load i32* %x1
  %y4 = load i32* %y2
  %tmp = add i32 %x3, %y4
  ret i32 %tmp
}
```
i32: 32-bit signed integer type
alloca: Allocate space on the stack; return a pointer
store: Write a value to an address
load: Read a value from an address
add: Add two values to produce a third
ret: Return a value to the caller
Basic Blocks

```c
int cond(bool b) {
    int x;
    if (b) x = 42;
    else x = 17;
    return x;
}
```

define i32 @cond(i1 %b) {
    entry:
    %b1 = alloca i1
    store i1 %b, i1* %b1
    %x = alloca i32
    %b2 = load i1* %b1
    br i1 %b2, label %then, label %else

    merge: ; preds = %else, %then
    %x3 = load i32* %x
    ret i32 %x3

    then: ; preds = %entry
    store i32 42, i32* %x
    br label %merge

    else: ; preds = %entry
    store i32 17, i32* %x
    br label %merge

    ret i32 %x3
```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}
int gcd(int a, int b) {
    while (a != b)
        if (a > b) a = a - b;
        else b = b - a;
    return a;
}