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

01011101010...
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.

```
Language-specific Frontends
C
C++
Java
Go
Objective C

Processor-specific Backends
IR
x86
ARM
MIPS
PPC
AVR
```
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

![Diagram of Bitcoin Script](image-url)
Stack-Based IR: Bitcoin Script

Stack

PUSH

DUP  HASH160  EQUALVERIFY  CHECKSIG
Stack-Based IR: Bitcoin Script

Stack

DUP HASH160 EQUALVERIFY CHECKSIG
Stack-Based IR: Bitcoin Script

Stack

POP

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

Stack

POP

CHECKSIG
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
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}

gcd:
gcd._gcdTmp0:
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

<table>
<thead>
<tr>
<th>C code:</th>
<th>Three-Address:</th>
<th>SSA:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int gcd(int a, int b) {</td>
<td><strong>WHILE</strong>:</td>
<td><strong>WHILE</strong>:</td>
</tr>
<tr>
<td>while (a != b)</td>
<td>$t = \text{sne } a, b$</td>
<td>$t_1 = \text{sne } a_1, b_1$</td>
</tr>
<tr>
<td>if (a &lt; b)</td>
<td>$b = \text{sub } b, a$</td>
<td>$b_2 = \text{sub } b_1, a_1$</td>
</tr>
<tr>
<td>b -= a;</td>
<td>$\text{jmp } \text{LOOP}$</td>
<td>$\text{jmp } \text{LOOP}$</td>
</tr>
<tr>
<td>else</td>
<td>ELSE:</td>
<td>ELSE:</td>
</tr>
<tr>
<td>a -= b;</td>
<td>$a = \text{sub } a, b$</td>
<td>$a_1 = \text{sub } a_1, b_1$</td>
</tr>
<tr>
<td>return a;</td>
<td>LOOP:</td>
<td>LOOP:</td>
</tr>
<tr>
<td>}</td>
<td>$\text{jmp } \text{WHILE}$</td>
<td>$\text{jmp } \text{WHILE}$</td>
</tr>
<tr>
<td></td>
<td>DONE:</td>
<td>DONE:</td>
</tr>
<tr>
<td></td>
<td>ret a</td>
<td>ret a1</td>
</tr>
</tbody>
</table>
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

- \( x = op\ y,\ z \): where \( op \) is a binary operation
- \( x = op\ y \): 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

\[ b + c \times d \]
Example

\( b + c \times d \)

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

\( t1 = \text{mul } c, d \)
Example

```
b + c * d
```

```
t2: +
    /   \
   b    t1: *
      /     \
     c       d
```

\[
t1 = \text{mul } c, d
\]

\[
t2 = \text{add } b, t1
\]
For each expression \( E \), we’ll synthesize two attributes:

- \( E.\text{addr} \): the name of the variable (often a temporary variable)
- \( E.\text{code} \): 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
  \[
  \text{Gen}(t, "=" , \text{op} , x,"","", y) \implies \text{"t = op x, y"}
  \]
CFG rule: \( E_0 \rightarrow \text{id} \)

Actions:
\[
\begin{align*}
E_0.\text{addr} & := \text{id} \\
E_0.\text{code} & := "" \quad \text{empty string}
\end{align*}
\]

*We do not consider scopes here.*

Example: \( E_0 = \text{ID("a")} \)
\[
\begin{align*}
E_0.\text{addr} & := "a" \\
E_0.\text{code} & := "" \quad \text{empty string}
\end{align*}
\]
CFG rule: $E_0 \rightarrow E_1 + E_2$

Actions:

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

Example: $a + b$

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

- $E_1.addr := "a"$  $E_1.code := ""$
- $E_2.addr := "b"$  $E_2.code := ""$
- $E_0.addr := "t1"$
- $E_0.code := "t1 = add a, b"$
Syntax-Directed Translation (SDT)

Example: $b + c \times d$

\begin{align*}
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} &:= E_1.\text{code} \ || \ E_2.\text{code} \ || \\
&\quad \text{Gen}(E_0.\text{addr}, "=", "add", E_1.\text{addr}, ",", E_2.\text{addr})
\end{align*}
Syntax-Directed Translation (SDT)

Example: b + c * d

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

\[ E_0.\text{code} := \text{E}_1.\text{code} \ || \ E_2.\text{code} \ || \]
\[ \text{Gen}(E_0.\text{addr}, ",", \text{"add"}, E_1.\text{addr}, ",", E_2.\text{addr}) \]
Example: \( b + c \times d \)

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

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

\[
E_0.\text{code} := "" \mid \mid E_2.\text{code} \mid \mid
\]

\[
\text{Gen}(E_0.\text{addr}, ",", \text{"add"}, E_1.\text{addr}, ",", E_2.\text{addr})
\]

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

Example: $b + c \times d$

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

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

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

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

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

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

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

\[
\begin{align*}
E_0 &= \text{PLUS} \ (E_1, E_2) \\
E_1 &= \text{ID}("b") \\
E_2 &= \text{MUL} \ (\text{ID}("c"), \text{ID}("d")) \\
E_0.\text{code} &= \text{""} \ || \ \text{"t1 = mul c, d"} \ || \\
&\quad \text{Gen("t2", ",", \text{"add"}, E_1.\text{addr}, ",", E_2.\text{addr})} \\
E_1.\text{addr} &= "b" \\
E_2.\text{addr} &= "t1"
\end{align*}
\]
Syntax-Directed Translation (SDT)

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

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

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

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

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

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

"t2 = \text{add} \ b, \ t1"
Translating Statements
Assignment

CFG rule: \( S \rightarrow \text{id} := E \)

Actions:

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

Example: \( a := b + c \)

\[ S = \text{ASG (ID("a"), E)} \quad E = \text{PLUS(ID("b"), ID("c"))} \]

\[ E\text{.code := "t1 = add b, c"} \quad E\text{.addr := "t1"} \]

\[ S\text{.code := "t1 = add b, c" || "a = t1"} \]
IF Statement

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

**Generated IR:**

\[
E.\text{code} \\
bz \text{Label}_\text{End}, E.\text{addr} \\
S.\text{code} \\
\text{Label}_\text{End:}
\]

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

\[
t1 = \text{slt} a, b \\
bz \text{Label}_\text{End}, t1 \\
a = \text{sub} a, b \\
\text{Label}_\text{End:}
\]
AST: IFELSE($E$, $S_1$, $S_2$)

Generated IR:

```
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:**

```
Label_While:
E.code
bz Label_End, E.addr
S.code
jmp Label_While

Label_End:
```
Function Calls

\[
f (E_1, \cdots, E_n)
\]

Generated IR:

\[
E_n\text{.code} \\
E_{n-1}\text{.code} \\
\cdots \\
E_1\text{.code} \\
\text{param } E_n\text{.addr} \\
\cdots \\
\text{param } E_1\text{.addr} \\
\text{call } f, n
\]
Function Calls

\[ f (E_1, \cdots, E_n) \]

Generated IR:

\[ E_n.\text{code} \]
\[ E_{n-1}.\text{code} \]
\[ \cdots \]
\[ E_1.\text{code} \]
\[ \text{param } E_n.\text{addr} \quad \text{how to pass parameters?} \]
\[ \cdots \]
\[ \text{param } E_1.\text{addr} \]
\[ \text{call } f, n \]
int x;  
where is this x stored? what is x.addr?

int main () {
    x = 4 ;
    int y;  
where is this y stored? what is y.addr?
...
}

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:  
\[ t1 = \text{sne } a1, b1 \]  
\[ \text{bz } DONE, t1 \]  
\[ t2 = \text{slt } a1, b1 \]  
\[ \text{bz } ELSE, t2 \]  
\[ b1 = \text{sub } b1, a1 \]  
\[ \text{jmp LOOP} \]

ELSE:  
\[ a1 = \text{sub } a1, b1 \]

LOOP:  
\[ \text{jmp WHILE} \]

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;
}
```

```c
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
}
```
The LLVM IR

\texttt{i32}: 32-bit signed integer type
\texttt{alloca}: Allocate space on the stack; return a pointer
\texttt{store}: Write a value to an address
\texttt{load}: Read a value from an address
\texttt{add}: Add two values to produce a third
\texttt{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
}
```
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;
}