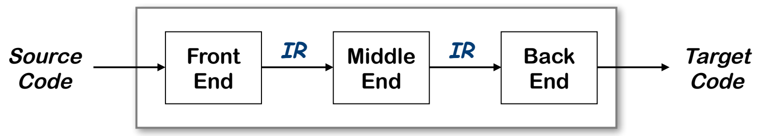
Most of the material in this lecture comes from Chapter 5 of EaC2

Intermediate Representations

Note by Baris Aktemur:
Our slides are adapted from Cooper and Torczon's slides that they prepared for COMP 412 at Rice.

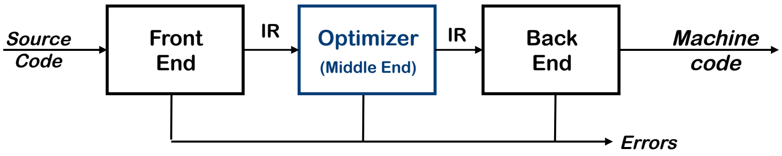
Intermediate Representations

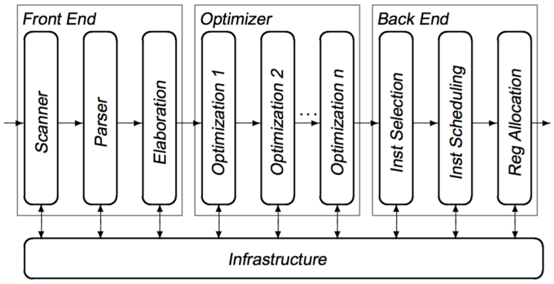


- Front end - produces an intermediate representation (*IR*)
- Middle end - transforms the *IR* into an equivalent *IR* that runs more efficiently
- Back end - transforms the *IR* into native code

- *IR* encodes the compiler's knowledge of the program
- Middle end usually consists of several passes

Traditional Three-part Compiler





■ FIGURE 1.1 Structure of a Typical Compiler.

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d) {
  int a, b, c, d;
     …
}
fee() {
  int f[3],g[0], h, i, j, k;
  char *p;
  fie(h,i,"ab",j, k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.\n",p,q);
  p = 10;
}
```

What is wrong with this program?
(let me count the ways …)

- number of args to fie()
- declared g[0], used g[17]
- "ab" is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are
"deeper than syntax"

To generate code, we need to understand its meaning !

Beyond Syntax

To generate code, the compiler needs to answer many questions

- Is “x” a scalar, an array, or a function? Is “x” declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of “x” does a given use reference?
- Is the expression “x * y + z” type-consistent?
- In “*a*[i,j,k]”, does *a* have three dimensions?
- Where can “z” be stored? *(register, local, global, heap, static)*
- In “f $\leftarrow$ 15”, how should 15 be represented?
- How many arguments does “fie()” take? What about “printf ()” ?
- Does “*p” reference the result of a “malloc()” ?
- Do “p” & “q” refer to the same memory location?
- Is “x” defined before it is used?

These are beyond the expressive power of a CFG

4

Semantic Analysis

- Before we transform the program (e.g. AST) into IR, we need to make sure it is semantically sane.
- Type checking…
- Will skip this part. See CS 321 content.

Where In The Course Are We?

- We are on the cusp of the art, science, & engineering of compilation
- Scanning & parsing are applications of automata theory
- The mid-section of the course will focus on issues where the compiler writer needs to choose among alternatives
  - The choices matter; they affect the quality of compiled code
  - There may be no "best answer" or "best practice"

The fun begins at this point

6

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

7

Types of Intermediate Representations

Three major categories

- Structural
  - Graphically oriented
  - Heavily used in source-to-source translators
  - 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

  Example:
  Control-flow graph

8

Three Address Code

Several different representations of three address code

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

  $x \leftarrow y \ \underline{op} \ z$

  With 1 operator ($\underline{op}$) and, at most, 3 names (x, y, & z)

Example:

$z \leftarrow x - \boxed{2 * y}$ becomes $\boxed{t} \leftarrow 2 * y$
$z \leftarrow x - \boxed{t}$

- Resembles many real machines
- Introduces a new set of names*

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

The original FORTRAN compiler used "quads"

```
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	3

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

Implicit names occupy no space

Remember, for a long time, 640Kb was a lot of RAM

11

Two Address Code

- Allows statements of the form

  $x \leftarrow x \; \underline{op} \; y$

  Has 1 operator ($\underline{op}$) and, at most, 2 names (x and y)

Example:

$z \leftarrow x - 2 * y$ becomes

```
t1 ← 2
t2 ← load y
t2 ← t2 * t1
z  ← load x
z  ← z - t2
```

- Can be very compact

Problems

- Machines no longer rely on destructive operations
- Difficult name space
  - Destructive operations make reuse hard
  - Good model for machines with destructive ops (PDP-11)

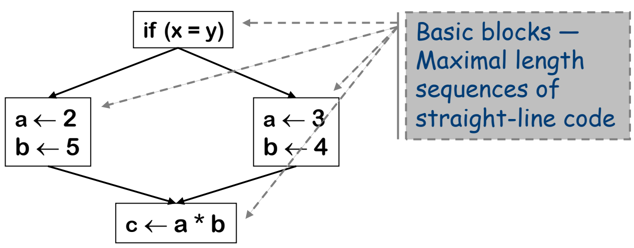
12

Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example



13

Exercise: Draw the CFG

```
stmtlist₀
switch (V) {
   case 1:  stmtlist₁
   case 2:  stmtlist₂
   ...
   case n:  stmtlistₙ
   default:  stmtlistₙ
}
stmtlistₙ₊₁
```

Exercise: Draw the CFG

"while" loop in C:

```
stmtlist0
while (x < k)
   stmtlist1
stmtlist2
```

"do-while" loop in C:

```
stmtlist0
do
   stmtlist1
while (x < k);
stmtlist2
```

"try-catch-finally" in Java:

```
stmtlist0
try {
   S0;                     // may throw
   S1;                     // may throw
} catch (etype1 e1) {
   S2;                     // simple statement
} catch (etype2 e2) {
   S3;                     // simple statement
} finally {
   S4;                     // simple statement
}
stmtlist1
```

Static Single Assignment Form

- The main idea: each name defined exactly once
- Introduce ϕ-functions to make it work

Original

```
x ← …
y ← …
while (x < k)
   x ← x + 1
   y ← y + x
```

SSA-form

```
        x0 ← …
        y0 ← …
        if (x0 >= k) goto next
loop:      x1 ← φ(x0,x2)
           y1 ← φ(y0,y2)
           x2 ← x1 + 1
           y2 ← y1 + x2
           if (x2 < k) goto loop
next:       …
```

Strengths of SSA-form

- Sharper analysis
- ϕ-functions give hints about placement
- (sometimes) faster algorithms

16