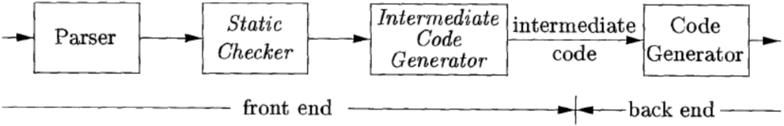
Code generation
& Optimization



Code generation

Input to the Code Generator

(a) **Intermediate representation (IR)** of the source program produced by the front end,

(b) **Symbol table** that is used to determine the **run-time** addresses of the data objects

The Target Program

Instruction-set architecture of the target machine

- Has a significant impact on the **difficulty of constructing a good code generator** that produces high-quality machine code.
- The most common target-machine architectures are **RISC** (reduced instruction set computer), **CISC** (complex instruction set computer)
- A RISC machine typically has **many registers**, three-address instructions, **simple addressing modes**, and a relatively **simple** instruction-set architecture.

Instruction Selection

The **code generator** must **map** the **IR program** into a **code sequence** that can be executed by the target machine.
The **complexity** of performing this **mapping** is determined by a factors such as

- the level of the IR
- the nature of the **instruction-set architecture**
- the **desired quality** of the **generated code**.

- The **code generator** may translate **each IR statement** into a sequence of machine instructions **using code templates**.
- Such statement by statement code generation, however, often produces **poor code**

Translation scheme: If we do not care about the efficiency of the target program, instruction selection is straightforward.
For **each type of three-address statement**, we can design a **code skeleton** that defines the **target code** to be generated for that construct

Instruction Selection – Example

example, every three-address statement of the form x = y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD   R0, y          // R0 = y           (load y into register R0)
ADD  R0, R0, z      // R0 = R0 + z      (add z to R0)
ST   x, R0          // x = R0           (store R0 into x)
```

```
a = b + c
d = a + e
```

would be translated into

```
LD   R0, b          // R0 = b
ADD  R0, R0, c      // R0 = R0 + c
ST   a, R0          // a = R0       ⟸
LD   R0, a          // R0 = a       ⟸    Redundant
ADD  R0, R0, e      // R0 = R0 + e
ST   d, R0          // d = R0
```

Instruction Selection – Example

- On most machines, a **given IR program** can be implemented by **many different code sequences**,
    - Significant **cost differences** between the **different implementations**.

- A **naive translation** of the intermediate code may therefore **lead to correct but unacceptably inefficient target code**.

- For example, if the target machine has an **"increment" instruction (INC)**
- The three-address statement **a = a + 1** may be implemented more efficiently by the **single instruction INC a**,
    - Rather than by a more obvious sequence that loads a into a register, adds one to the register, and then stores the result back into a

```
LD  R0, a        // R0 = a
ADD R0, R0, #1   // R0 = R0 + 1
ST  a, R0        // a = R0
```

Register Allocation

- A **key problem** in code generation is **deciding what values to hold in what registers**.

- Registers are the **fastest** computational unit on the target machine,
    - but we usually do not have **enough** of them to hold **all values**.
    - Values not held in registers need to reside **in memory**.

- **Instructions involving register** operands are invariably **shorter and faster** than those involving **operands in memory**,
    - **Efficient utilization of registers** is particularly important.

The **use of registers** is often subdivided into **two subproblems:**

1. **Register allocation**, during which we **select the set of variables** that will **reside in registers** at each point in the program.
2. **Register assignment**, during which we **pick the specific register** that a variable will reside in.

Basic Blocks & Flow graphs

- Introduce a **graph representation of intermediate code** that is helpful for discussing code generation
    - Even if the graph is not constructed explicitly by a code-generation algorithm.

- Code generation **benefits from context**.

- We can do a **better job of register allocation** if we know how variables are **defined and used**.

Basic Blocks & Flow graphs

The representation is constructed as follows:

1. Partition the intermediate code into *basic blocks*, which are maximal sequences of consecutive three-address instructions with the properties that
   (a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
   (b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.
2. The basic blocks become the nodes of a *flow graph*, whose edges indicate which blocks can follow which other blocks.

Basic Blocks

- We begin a **new basic block** with the **first instruction**

- Keep adding instructions
  - until we meet either a **jump, a conditional jump**,
  - or **a label** on the following instruction.

- In the **absence of jumps and labels**, control proceeds **sequentially** from one instruction to the next.

- **Task:** ***Identify leaders,*** that is, the **first instructions** in some **basic block**.

Basic Blocks - Leaders

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

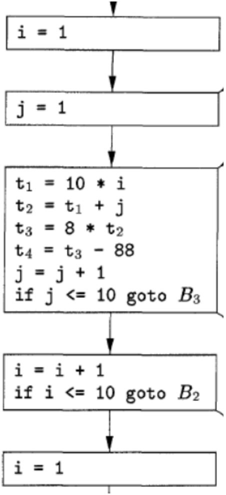
Basic Blocks

for i from 1 to 10 **do**
    for j from 1 to 10 **do**
        $a[i, j] = 0.0;$
for i from 1 to 10 **do**
    $a[i, i] = 1.0;$

```
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)
```

leaders are instructions 1, 2, 3, 10, 12, and 13

Basic Blocks



Flow Graphs

- We represent the **flow of control** by a **flow graph**.
- The **nodes** of the flow graph are the **basic blocks**.
- There is an **edge from block B to block C** if and only if
    - it is possible for the **first instruction in block C** to immediately follow the **last instruction in block B**.

There are two ways that such an edge could be justified:

- There is a **conditional or unconditional jump** from the end of B to the beginning of C.
- **Block C immediately follows Block B** in the original order of the three-address instructions
    - B does not end in an unconditional jump
    - **Maybe due to labels**

We say that **B is a predecessor of C**, and **C is a successor of B**.

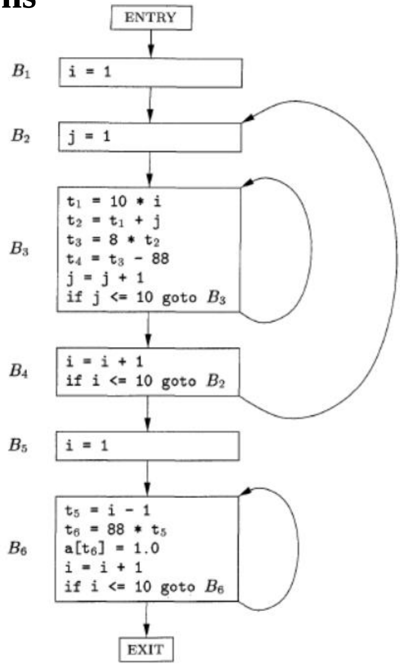
Flow Graphs

- Often we add two nodes, called the **entry and exit**,

- There is an **edge from the entry** to the **first executable node** of the flow graph,
    - that is, to the **basic block** that comes from the **first instruction** of the intermediate code.

- There is an edge **to the exit** from **any basic block** that contains an instruction that could be the **last executed instruction** of the program.

Flow Graphs


ENTRY
B1
i = 1
B2
j = 1
B3
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
j = j + 1
if j <= 10 goto B3
B4
i = i + 1
if i <= 10 goto B2
B5
i = 1
B6
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B6
EXIT

Code Generator

- Algorithm that generates code for a **single basic block**
- It considers **each three-address instruction** in turn, and keeps track of **what values are in what registers** so it can avoid generating **unnecessary loads and stores**.
- Deciding how to **use registers to best advantage**
- In most machine architectures, some or all of the **operands** of an operation must be in **registers** in order to perform the operation.
- These are competing needs, since the number of registers available is **limited**.

Code Generator

- We further assume that for each operator, there is exactly one machine instruction that takes the necessary operands in registers and performs that operation, leaving the result in a register. The machine instructions are of the form

$$\text{LD } reg,\ mem$$

$$\text{ST } mem,\ reg$$

$$OP\ reg,\ reg,\ reg$$

Register and Address Descriptors

- Our code-generation algorithm considers each three-address instruction in turn and **decides what loads** are necessary to get the needed operands into registers.

- After generating the loads, it generates the **operation itself.**

- Then, if there is a need **to store the result** into a memory location, it also generates that store.
- We require a **data structure** that tells us what program **variables** currently have **their value in a register, and in which register**

- We also need to know whether the **memory location for a given variable currently has the proper value for that variable**
  - Since a new value for the variable may have been computed in a register and not yet stored.

Register Descriptors

- a ***register descriptor*** keeps track of the variable names whose current value is in that register.
- **All register descriptors are empty**. As the code generation progresses, each register will hold the value of zero or more names.

Address Descriptors

For each program variable, an **address descriptor** keeps track of the **location or locations** where the current value of that variable can be found.
The **location** might be a **register, a memory address** etc.

The information can be stored in the **symbol-table entry** for that **variable name**.

The Code-Generation Algorithm

- An essential part of the algorithm is a **function getReg(I),**
    - which selects registers for each memory location associated with the three-address instruction I.

- Function **getReg** has access to the **register and address descriptors** for all the variables of the basic block

- While we do not know the total number of registers available for local data belonging to a basic block, we assume that there are **enough registers**

Machine Instructions for Operations

For a three-address instruction such as $x = y + z$, do the following:

1. Use $getReg(x = y + z)$ to select registers for x, y, and z. Call these R_x, R_y, and R_z.
2. If y is not in R_y (according to the register descriptor for R_y), then issue an instruction `LD` R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y).
3. Similarly, if z is not in R_z, issue and instruction `LD` R_z, z', where z' is a location for z.
4. Issue the instruction `ADD` R_x, R_y, R_z.

Ending the Basic Block

- If the **variable is live on exit** from the block,
    - Or if we **don't know** which variables are live on exit,
    - then we assume that the **value of the variable** is **needed later**.

- In that case, for **each variable x** whose **address descriptor does not say** that its value is located in the **memory location for x**

- We must generate the instruction **ST x, R**, where **R is a register in which x value** exists at the end of the block.

Managing Register and Address Descriptors

- As the code-generation algorithm **issues load, store, and other machine instructions**,
- It needs to **update the register and address descriptors**.
- The rules are as follows:

1. For the instruction LD R, x
   (a) Change the register descriptor for register R so it holds only x.
   (b) Change the address descriptor for x by adding register R as an additional location.

2. For the instruction ST x, R, change the address descriptor for x to include its own memory location.

3. For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction $x = y + z$
   (a) Change the register descriptor for R_x so that it holds only x.
   (b) Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.
   (c) Remove R_x from the address descriptor of any variable other than x.

Machine Instructions for Copy Statements

three-address copy statement of the form x = y.

- We assume that **getReg** will always choose the **same register** for **both x and y**

- If y is **not** already in that register **Ry**,
  - then generate the machine instruction **LD Ry, y**.
  - If y was already in Ry, we **do nothing**.

- It is only necessary that we adjust the **register description** for **Ry**
  - So that it **includes x** as one of the values found there.

Machine Instructions for Copy Statements

When we process a copy statement $x = y$, after generating the load for y into register R_y, if needed, and after managing descriptors as for all load statements (per rule 1):

(a) Add x to the register descriptor for R_y.

(b) Change the address descriptor for x so that its only location is R_y.

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

R1	R2	R3	a	b	c	d	t	u	v
			a	b	c	d			

```
t = a - b
    LD R1, a
    LD R2, b
    SUB R2, R1, R2
```

R1	R2	R3	a	b	c	d	t	u	v
a	t		a, R1	b	c	d	R2		

```
u = a - c
    LD R3, c
    SUB R1, R1, R3
```

R1	R2	R3	a	b	c	d	t	u	v
u	t	c	a	b	c, R3	d	R2	R1	

```
v = t + u
    ADD R3, R2, R1
```

R1	R2	R3	a	b	c	d	t	u	v
u	t	v	a	b	c	d	R2	R1	R3

```
a = d
    LD R2, d
```

R1	R2	R3	a	b	c	d	t	u	v
u	a, d	v	R2	b	c	d, R2		R1	R3

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

	R1	R2	R3	a	b	c	d	t	u	v
	u	t	v	a	b	c	d	R2	R1	R3
a = d LD R2, d										
	u	a, d	v	R2	b	c	d, R2		R1	R3
d = v + u ADD R1, R3, R1										
	d	a	v	R2	b	c	R1			R3
exit ST a, R2 ST d, R1										
	d	a	v	a, R2	b	c	d, R1			R3

Design of the Function *getReg*

- There are many options,
- although there are also some **absolute prohibitions** against choices that lead to incorrect code due to the loss of the value of one or more live variables

- We use **x = y + z** as the generic example.
- First, we must **pick a register for y** and **a register for z**.
- The issues are the same, so we shall concentrate on picking register **Ry for y**.

1. If y is currently in a register, pick a register already containing y as R_y. Do not issue a machine instruction to load this register, as none is needed.
2. If y is not in a register, but there is a register that is currently empty, pick one such register as R_y.
3. The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse. Let R be a candidate

register, and suppose v is one of the variables that the register descriptor for R says is in R. We need to make sure that v's value either is not really needed, or that there is somewhere else we can go to get the value of R. The possibilities are:

(a) If the address descriptor for v says that v is somewhere besides R, then we are OK.

(b) If v is x, the value being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.

(c) Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.

(d) If we are not OK by one of the first two cases, then we need to generate the store instruction ST v, R to place a copy of v in its own memory location. This operation is called a *spill*.

Since R may hold several variables at the moment, we repeat the above steps for each such variable v. At the end, R's "score" is the number of store instructions we needed to generate. Pick one of the registers with the lowest score.

Selection of the register *Rx*

Now, consider the selection of the register R_x. The issues and options are almost as for y, so we shall only mention the differences.

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for R_x. This statement holds even if x is one of y and z, since our machine instructions allows two registers to be the same in one instruction.

2. If y is not used after instruction I, in the sense described for variable v in item (3c), and R_y holds only y after being loaded, if necessary, then R_y can also be used as R_x. A similar option holds regarding z and R_z.

The last matter to consider specially is the case when I is a copy instruction $x = y$. We pick the register R_y as above. Then, we always choose $R_x = R_y$.