Unit - IV

Chapter 9

Code Generation

Partha Sarathi Chakraborty
Assistant Professor
Department of Computer Science and Engineering
SRM University, Delhi – NCR Campus

Outline

- Issues in the Design of Code Generator
- The Target Machine
- Runtime Storage Management
- Basic Blocks and Flow Graphs
- Next-use Information
- DAG representation of Basic Blocks
- Peephole Optimization
- Cross Compiler – T diagrams
- A simple Code Generator

Code Generation

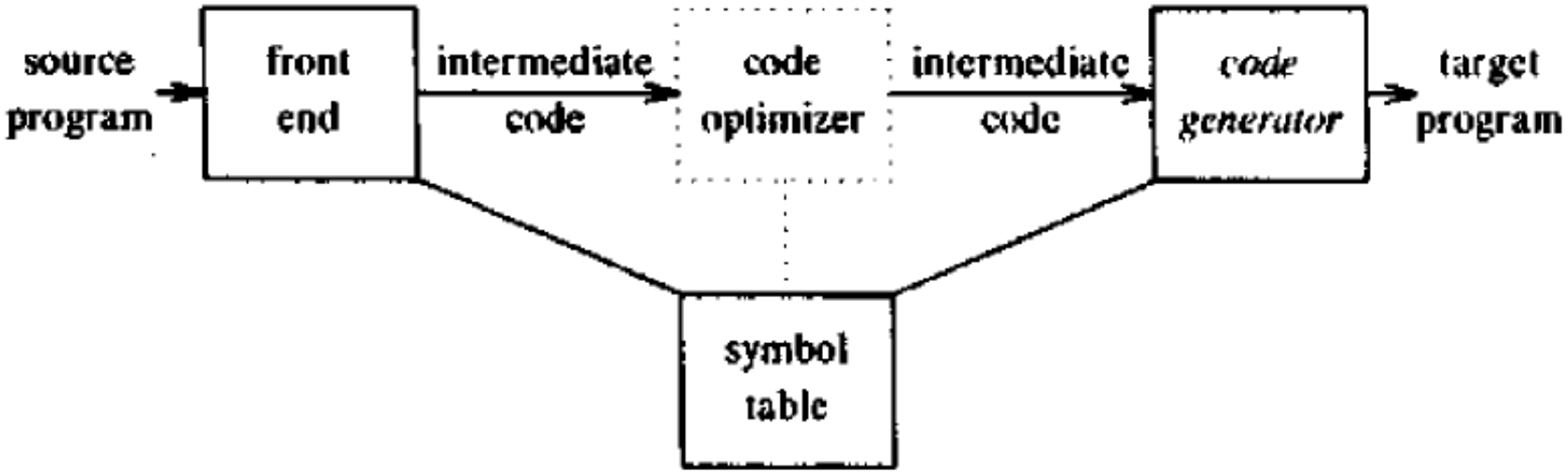


Fig. 9.1. Position of code generator.

The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine.

Issues in Code Generation

- Input to the Code Generator
- Target Programs
- Memory Management
- Instruction Selection
- Register Allocation
- Choice of Evaluation Order
- Approaches to Code Generation

Target Program Code

- The back-end code generator of a compiler may generate different forms of code, depending on the requirements:
  - Absolute machine code (executable code)
  - Relocatable machine code (object files for linker)
  - Assembly language (facilitates debugging)
  - Byte code forms for interpreters (e.g. JVM)

The Target Machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set
- Our (hypothetical) machine:
  - Byte-addressable (word = 4 bytes)
  - Has *n* general purpose registers **`R0`**, **`R1`**, …, **`R`***n*-1
  - Two-address instructions of the form

op source, destination

The Target Machine: Op-codes and Address Modes

- Op-codes (*op*), for example
  - **`MOV`** (move content of *source* to *destination*)
  - **`ADD`** (add content of *source* to *destination*)
  - **`SUB`** (subtract content of *source* from *dest.*)
- Address modes

Mode	Form	Address	Added Cost
Absolute	**`M`**	**`M`**	1
Register	**`R`**	**`R`**	0
Indexed	*c*(**`R`**)	*c*+*contents*(**`R`**)	1
Indirect register	***`R`**	*contents*(**`R`**)	0
Indirect indexed	****c*(**`R`**)	*contents*(*c*+*contents*(**`R`**))	1
Literal	**#***c*	N/A	1

Instruction Costs

- Machine is a simple, non-super-scalar processor with fixed instruction costs
- Realistic machines have deep pipelines, I-cache, D-cache, etc.
- Define the cost of instruction
  = 1 + cost(*source*-mode) + cost(*destination*-mode)

Examples

Instruction	Operation	Cost
MOV R0,R1	Store *content*(**R0**) into register **R1**	1
MOV R0,M	Store *content*(**R0**) into memory location **M**	2
MOV M,R0	Store *content*(**M**) into register **R0**	2
MOV 4(R0),M	Store *contents*(4+*contents*(**R0**)) into **M**	3
MOV *4(R0),M	Store *contents*(*contents*(4+*contents*(**R0**))) into **M**	3
MOV #1,R0	Store 1 into **R0**	2
ADD 4(R0),*12(R1)	Add *contents*(4+*contents*(**R0**)) to *contents*(12+*contents*(**R1**))	3

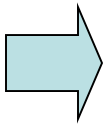
Instruction Selection

- Instruction selection is important to obtain efficient code
- Suppose we translate three-address code

  $x := y + z$

  to:
  ```
  MOV y,R0
  ADD z,R0
  MOV R0,x
  ```

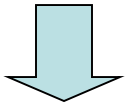
```
a:=a+1
```



```
MOV a,R0
ADD #1,R0
MOV R0,a
```

Cost = 6

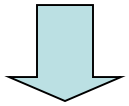
Better



```
ADD #1,a
```

Cost = 3

Better



```
INC a
```

Cost = 2

Instruction Selection: Utilizing Addressing Modes

- Suppose we translate **`a:=b+c`** into

```
MOV b,R0
ADD c,R0
MOV R0,a
```

- Assuming addresses of **`a`**, **`b`**, and **`c`** are stored in **`R0`**, **`R1`**, and **`R2`**

```
MOV *R1,*R0
ADD *R2,*R0
```

- Assuming **`R1`** and **`R2`** contain values of **`b`** and **`c`**

```
ADD R2,R1
MOV R1,a
```

Need for Global Machine-Specific Code Optimizations

- Suppose we translate three-address code

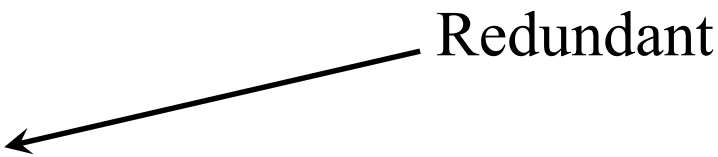
  x **:=** *y* **+** *z*

  to:
  ```
  MOV y,R0
  ADD z,R0
  MOV R0,x
  ```

- Then, we translate

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

  to:
  ```
  MOV a,R0
  ADD b,R0
  MOV R0,a
  MOV a,R0
  ADD e,R0
  MOV R0,d
  ```



Register Allocation and Assignment

- Efficient utilization of the limited set of registers is important to generate good code
- Registers are assigned by
  - *Register allocation* to select the set of variables that will reside in registers at a point in the code
  - *Register assignment* to pick the specific register that a variable will reside in
- Finding an optimal register assignment in general is NP-complete

Example

```
t:=a+b
t:=t*c
t:=t/d
```

{ R1=t }

```
MOV a,R1
ADD b,R1
MUL c,R1
DIV d,R1
MOV R1,t
```

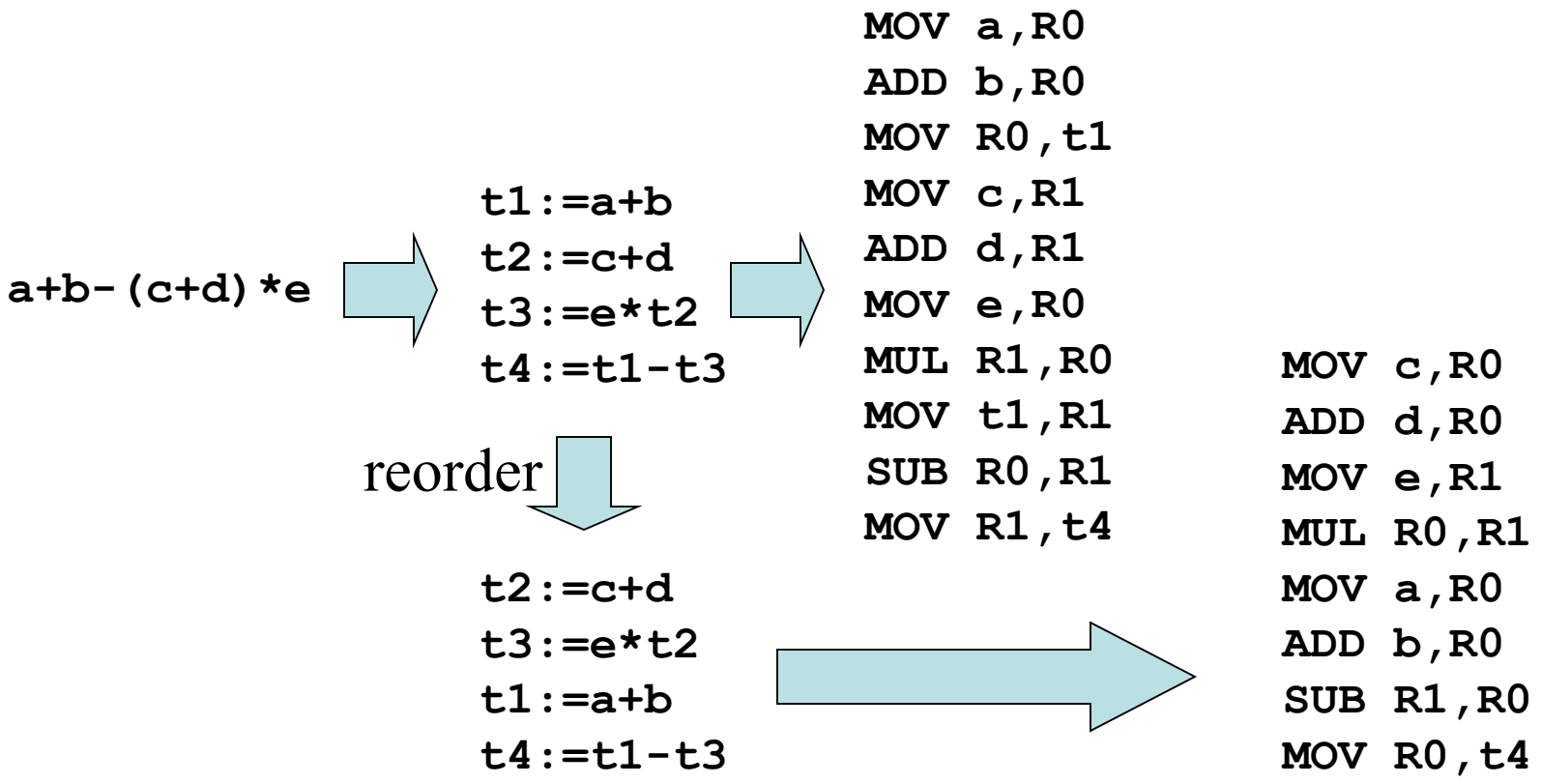
```
t:=a*b
t:=t+a
t:=t/d
```

{ R0=a, R1=t }

```
MOV a,R0
MOV R0,R1
MUL b,R1
ADD R0,R1
DIV d,R1
MOV R1,t
```

Choice of Evaluation Order

- When instructions are independent, their evaluation order can be changed



Generating Code for Stack Allocation of Activation Records

```
t1 := a + b
param t1
param c
t2 := call foo,2
…


func foo
…
return t1
```

```
100: ADD #16,SP        Push frame
108: MOV a,R0
116: ADD b,R0
124: MOV R0,4(SP)      Store a+b
132: MOV c,8(SP)       Store c
140: MOV #156,*SP      Store return address
148: GOTO 500          Jump to foo
156: MOV 12(SP),R0     Get return value
164: SUB #16,SP        Remove frame
172: …

500: …
564: MOV R0,12(SP)     Store return value
572: GOTO *SP          Return to caller
```

Note: Language and machine dependent
Here we assume C-like implementation with SP and no FP

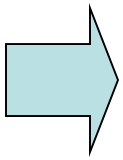
Basic Block

- A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.
- The following sequence of three address code forms a basic block:

```
t1 = a * a
t2 = a * b
t3 = t1 + t2
```

Basic Blocks

```
     MOV 1,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```



```
     MOV 1,R0
     MOV n,R1
     JMP L2
```

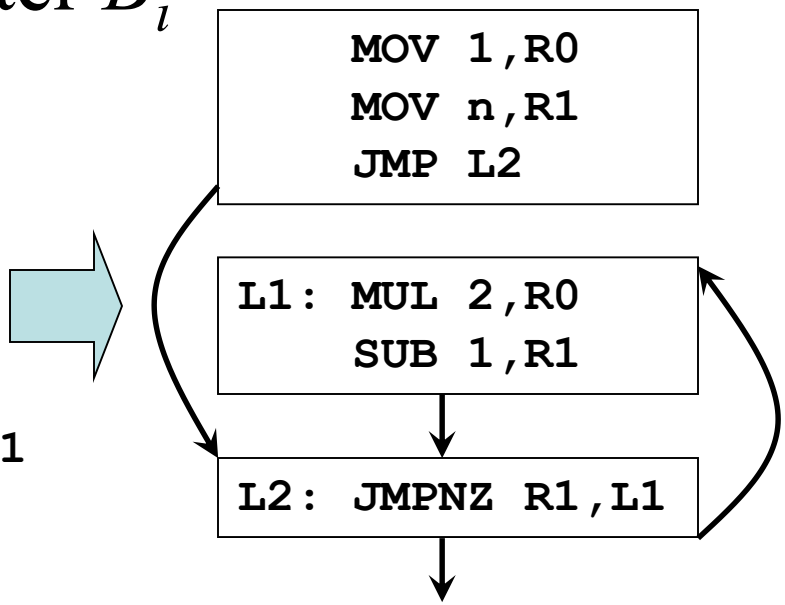
```
L1:  MUL 2,R0
     SUB 1,R1
```

```
L2:  JMPNZ R1,L1
```

Basic Blocks and Control Flow Graphs

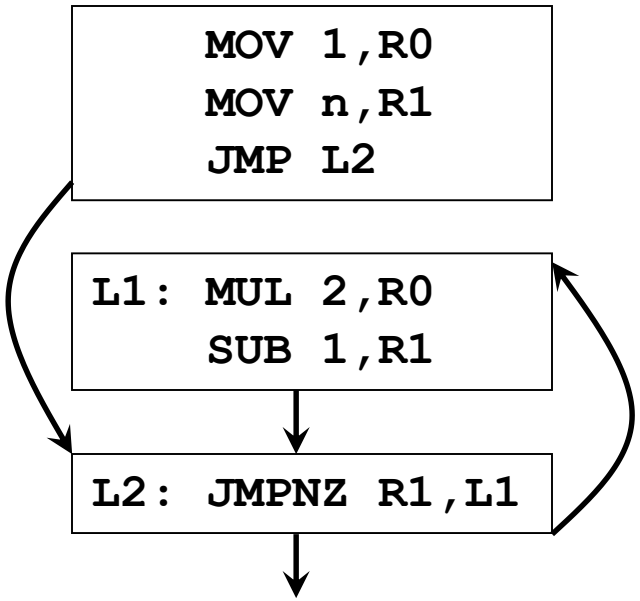
- A *control flow graph* (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ iff B_j can be executed immediately after B_i

```
      MOV 1,R0
      MOV n,R1
      JMP L2
L1:   MUL 2,R0
      SUB 1,R1
L2:   JMPNZ R1,L1
```



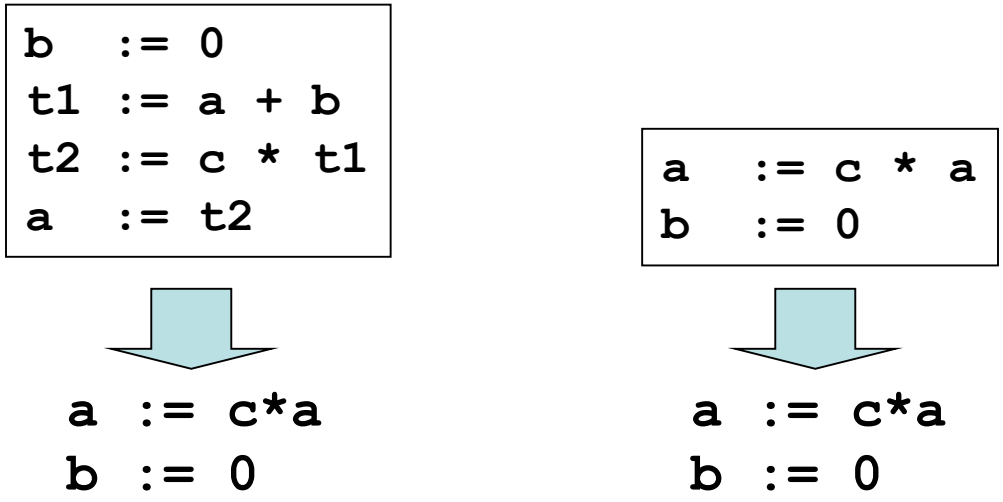
Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block B_1 is a *predecessor* of B_2
  - Basic block B_2 is a *successor* of B_1



Equivalence of Basic Blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions



Blocks are equivalent, assuming **`t1`** and **`t2`** are *dead*: no longer used (no longer *live*)

Partition Algorithm for Basic Blocks

Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements if basic blocks
   a) The first statement is the leader
   b) Any statement that is the target of a goto is a leader
   c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Dot Product of Two Vectors and Three-Address Code

```
begin
     prod := 0;
     i := 1;
     do begin
         prod := prod + a[i] * b[i];
         i := i + 1
     end
     while i <= 20
end
```

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]          /* compute a[i] */
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]          /* compute b[i] */
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

Basic Block

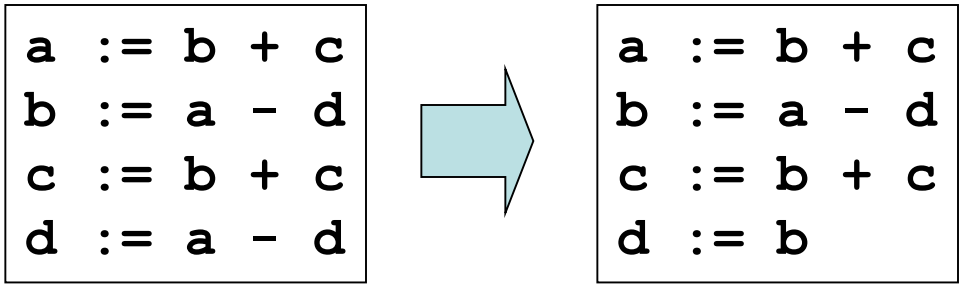
- Transformations on Basic Blocks
- Structure-Preserving Transformations
- Algebraic Transformations
- Representations of Basic Blocks

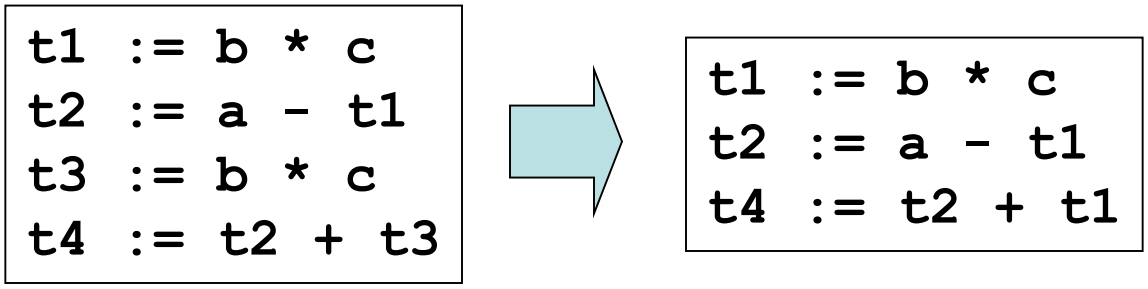
Transformations on Basic Blocks

- A *code-improving transformation* is a code optimization to improve speed or reduce code size
- *Global transformations* are performed across basic blocks
- *Local transformations* are only performed on single basic blocks
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

Common-Subexpression Elimination

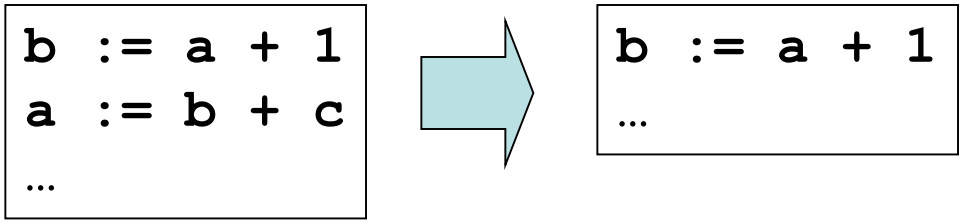
- Remove redundant computations



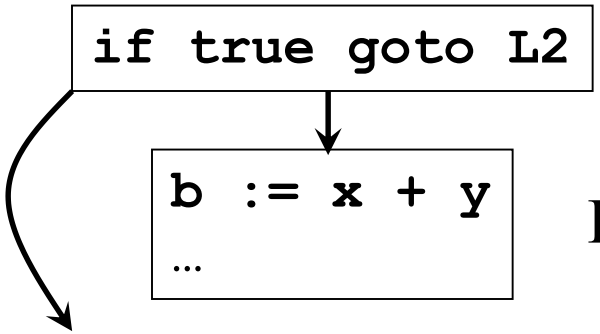


Dead Code Elimination

- Remove unused statements



Assuming **a** is *dead* (not used)

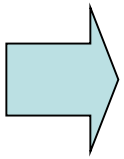


Remove unreachable code

Renaming Temporary Variables

- Temporary variables that are dead at the end of a block can be safely renamed

```
t1 := b + c
t2 := a - t1
t1 := t1 * d
d := t2 + t1
```



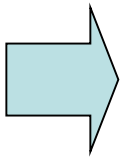
```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

Normal-form block

Interchange of Statements

- Independent statements can be reordered

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```



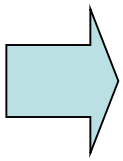
```
t1 := b + c
t3 := t1 * d
t2 := a - t1
d := t2 + t3
```

Note that normal-form blocks permit all statement interchanges that are possible

Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms

```
t1 := a - a
t2 := b + t1
t3 := 2 * t2
```



```
t1 := 0
t2 := b
t3 := t2 << 1
```

Flow Graph

- The representation of flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a *flow graph*.
- The nodes of the flow graph are the basic blocks.
- One node is distinguished as initial; it is the block whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence; that is, if
  - there is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2, or
  - B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.
- B_1 is a predecessor of B_2, and B_2 is a successor of B_1.

Representation of Basic Blocks in Flow Graph

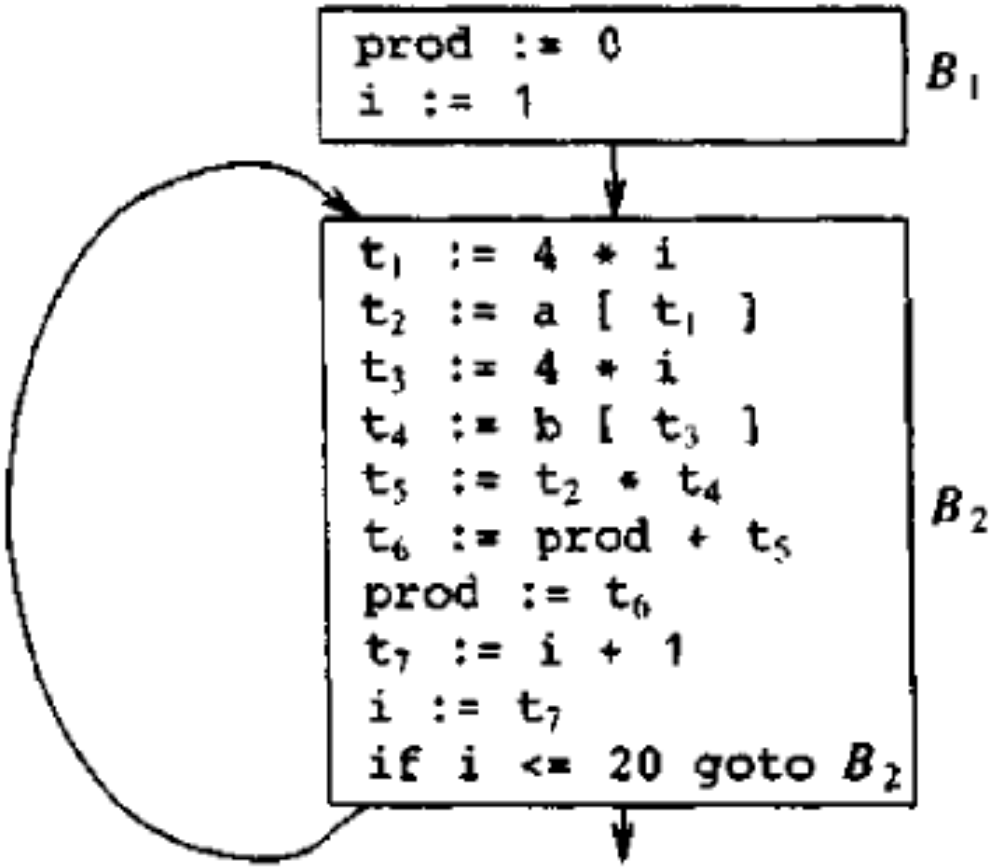


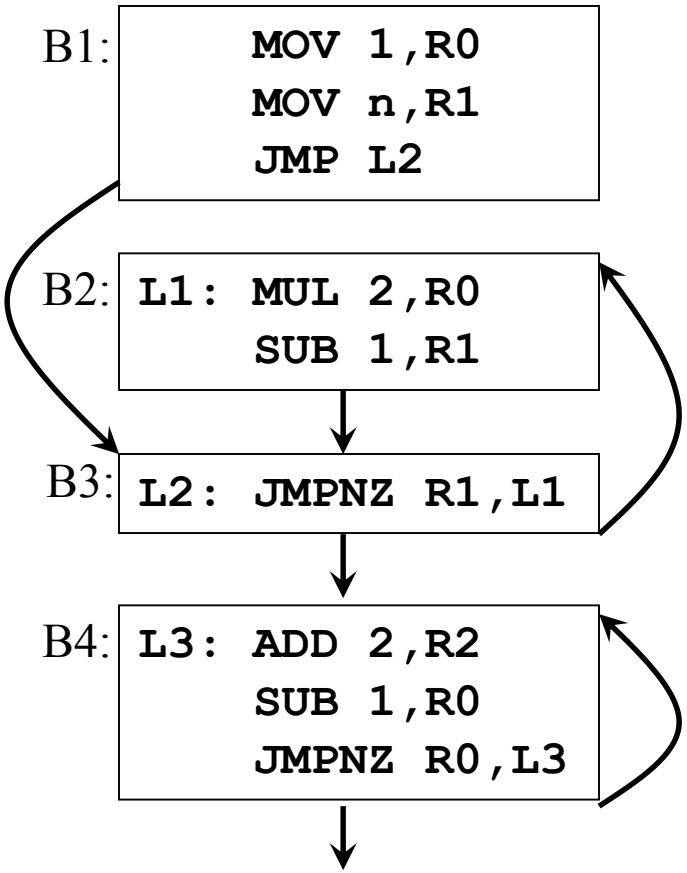
Fig. 9.9. Flow graph for program.

Loop

- A *Loop* is a collection of nodes in a flow graph such that
  - All nodes in the collection are ***strongly connected***; that is, from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and
  - The collection of nodes has a unique ***entry***, that is, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

- A loop that contains no other loops is called an ***inner*** loop.

Loops (Example)



Strongly connected components:

SCC={ {B2,B3},
{B4} }

Entries:
B3, B4

Next-Use

- Next-use information is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

  i: $x := y$ op z

  - Add liveness/next-use info on x, y, and z to statement i
  - Set x to "not live" and "no next use"
  - Set y and z to "live" and the next uses of y and z to i

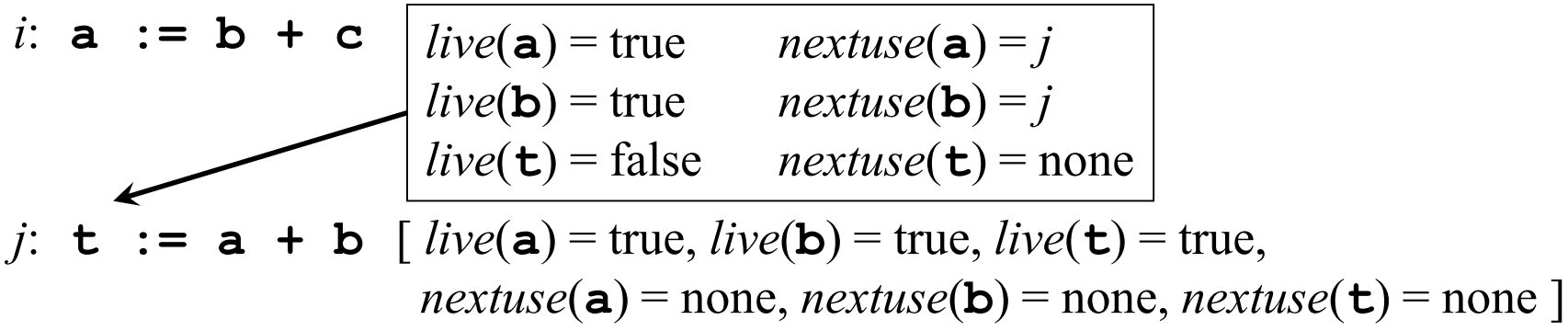
Next-Use (Step 1)

i: **`a := b + c`**

j: **`t := a + b`** [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Attach current live/next-use information
Because info is empty, assume variables are live
(Data flow analysis Ch.10 can provide accurate information)

Next-Use (Step 2)



Compute live/next-use information at *j*

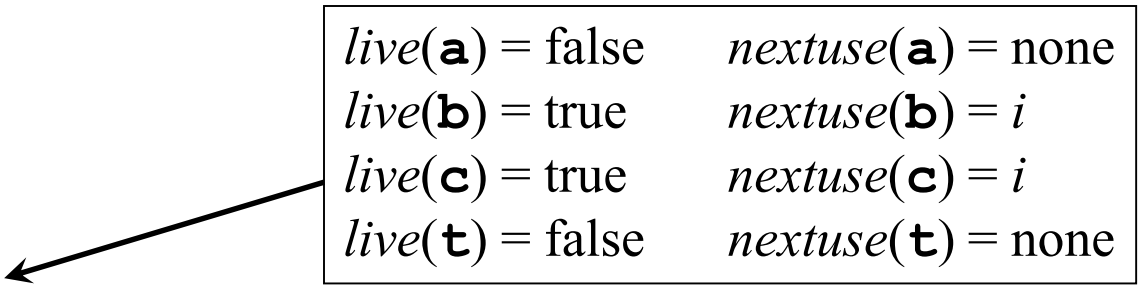
Next-Use (Step 3)

i: **`a := b + c`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`c`**) = false,
nextuse(**`a`**) = *j*, *nextuse*(**`b`**) = *j*, *nextuse*(**`c`**) = none]

j: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Attach current live/next-use information to *i*

Next-Use (Step 4)



i: **a := b + c** [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false,
nextuse(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

j: **t := a + b** [*live*(**a**) = false, *live*(**b**) = false, *live*(**t**) = false,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Compute live/next-use information *i*

A Code Generator

- Generates target code for a sequence of three-address statements using next-use information
- Uses new function *getreg* to assign registers to variables
- Computed results are kept in registers as long as possible, which means:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Checks if operands to three-address code are available in registers

The Code Generation Algorithm

- For each statement $x := y$ op z
  1. Set location $L = getreg(y, z)$
  2. If $y \notin L$ then generate
     MOV *y'*,*L*
     where *y'* denotes one of the locations where the value of *y* is available (choose register if possible)
  3. Generate
     OP *z'*,*L*
     where *z'* is one of the locations of *z*;
     Update register/address descriptor of *x* to include *L*
  4. If *y* and/or *z* has no next use and is stored in register, update register descriptors to remove *y* and/or *z*

Register and Address Descriptors

- A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

  `MOV a,R0` "`R0` contains `a`"

- An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

  ```
  MOV a,R0
  MOV R0,R1
  ```
  "`a` in `R0` and `R1`"

The *getreg* Algorithm

- To compute *getreg*(*y*,*z*)
  1. If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* has no next use, then return *R*;
     Update address descriptor: value *y* no longer in *R*
  2. Else, return a new empty register if available
  3. Else, find an occupied register *R*;
     Store contents (register spill) by generating
     MOV *R*,*M*
     for every *M* in address descriptor of *y*;
     Return register *R*
  4. Return a memory location

Code Generation Example

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*
		Registers empty	
`t := a - b`	`MOV a,R0` `SUB b,R0`	`R0` contains `t`	`t` in `R0`
`u := a - c`	`MOV a,R1` `SUB c,R1`	`R0` contains `t` `R1` contains `u`	`t` in `R0` `u` in `R1`
`v := t + u`	`ADD R1,R0`	`R0` contains `v` `R1` contains `u`	`u` in `R1` `v` in `R0`
`d := v + u`	`ADD R1,R0` `MOV R0,d`	`R0` contains `d`	`d` in `R0` `d` in `R0` and memory

DAG representation of Basic Blocks

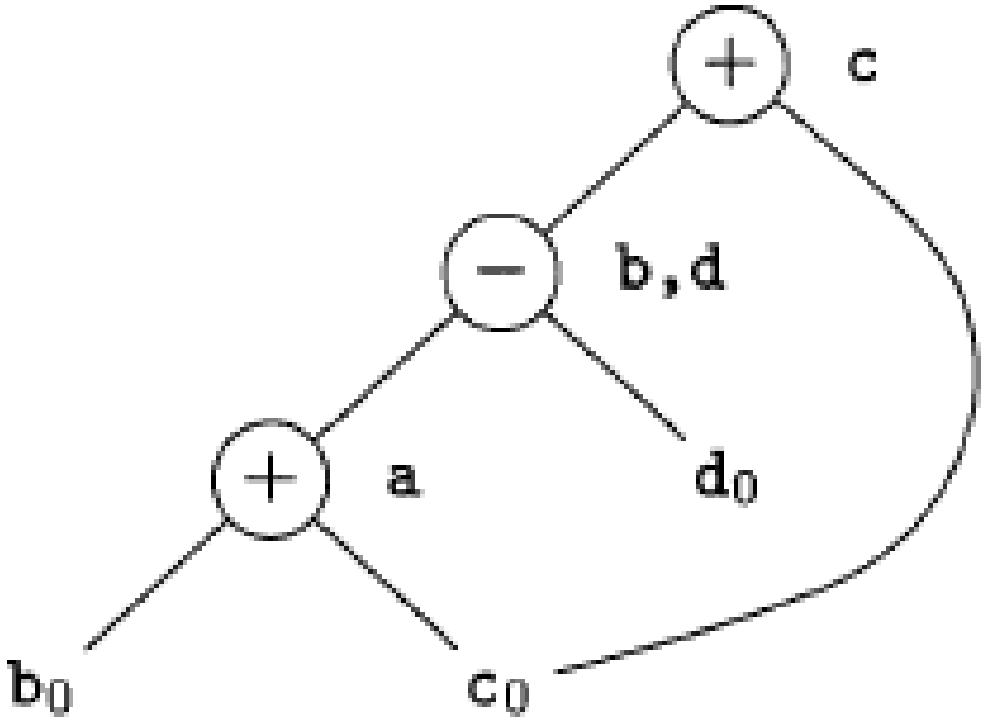
- useful data structures for implementing transformations on basic blocks
- gives a picture of how value computed by a statement is used in subsequent statements
- good way of determining common sub-expressions
- A dag for a basic block has following labels on the nodes
  - leaves are labeled by unique identifiers, either variable names or constants
  - interior nodes are labeled by an operator symbol
  - nodes are also optionally given a sequence of identifiers for labels

DAG for Basic Block : Example

Three Address Code

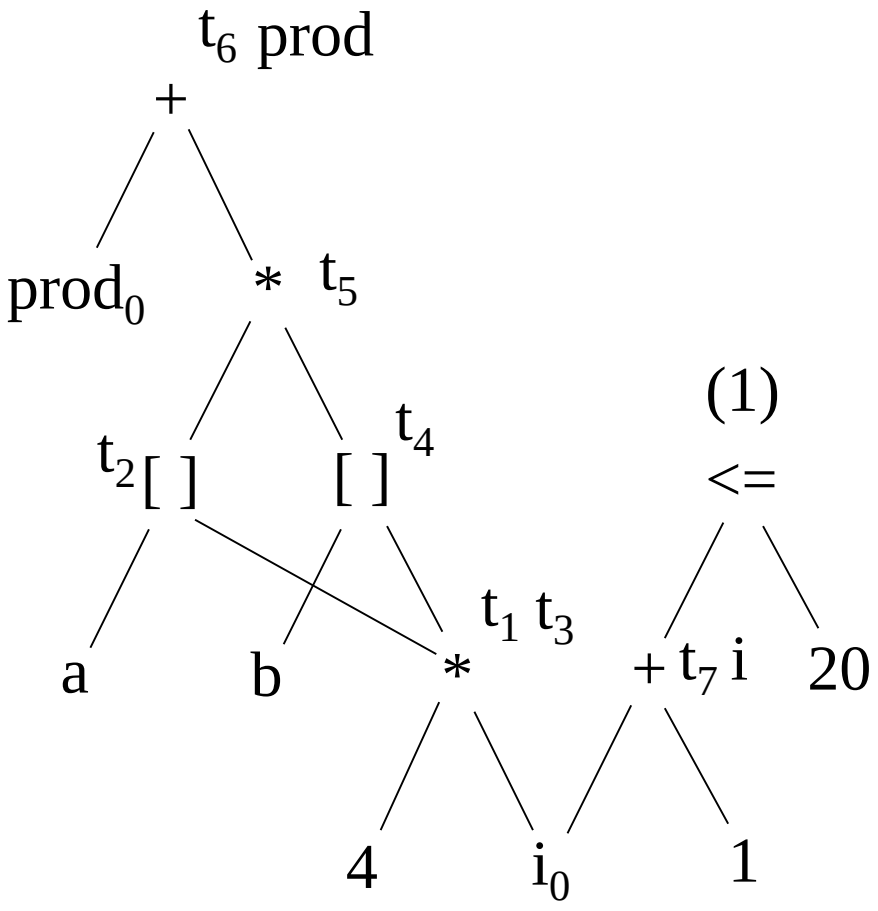
```
a = b + c
b = a - d
c = b + c
d = a - d
```

DAG



DAG Representation: Example

1. $t_1 := 4 * i$
2. $t_2 := a[t_1]$
3. $t_3 := 4 * i$
4. $t_4 := b[t_3]$
5. $t_5 := t_2 * t_4$
6. $t_6 := prod + t_5$
7. $prod := t_6$
8. $t_7 := i + 1$
9. $i := t_7$
10. if i <= 20 goto (1)



Peephole Optimization

- A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.
- Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.
- The peephole is a small, sliding window on a program. The code in the peephole need not be contiguous, although some implementations do require this.
- It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- In general, repeated passes over the target code are necessary to get the maximum benefit.

Peephole Optimization

- Redundant-instruction elimination
  - Redundant Loads and Store
  - Unreachable Code
- Flow-of-control optimizations
  - Elimination of unnecessary jumps

```
      goto  L1    ⇒     goto  L2
L1:   goto  L2        L1: goto  L2
```

- Algebraic simplifications

  x = x + 0 x = x * 1

- Reduction in Strength

  x = x ^ 2 ⇒ x = x * x

  x = 2 * x ⇒ x = x + x

- Use of machine idioms

  auto increment, auto-decrement, right-shift, left-shift

```
MOV R0, a      MOV a, R0
```

```
     if debug == 1 goto L1
     goto L2
L1:  print debugging information
L2:
```

```
     if debug != 1 goto L2
     print debugging information
L2:
```

Cross Compiler – T diagrams

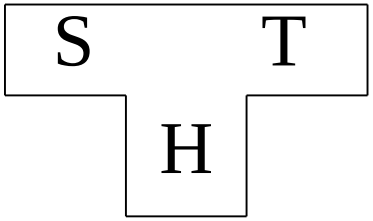
- Cross Compiler: A compiler may run on one machine and produce target code for another machine.

Third Language for Compiler Construction

- Machine language
  - compiler to execute immediately;
- Another language with existed compiler on the same target machine : (First Scenario)
  - Compile the new compiler with existing compiler
- Another language with existed compiler on different machine : (Second Scenario)
  - Compilation produce a cross compiler

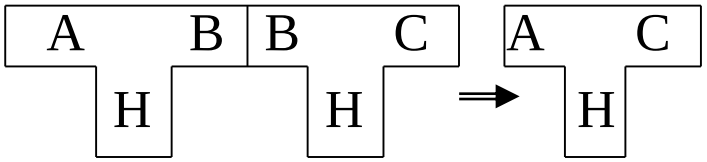
T-Diagram Describing Complex Situation

- A compiler written in language H that translates language S into language T.



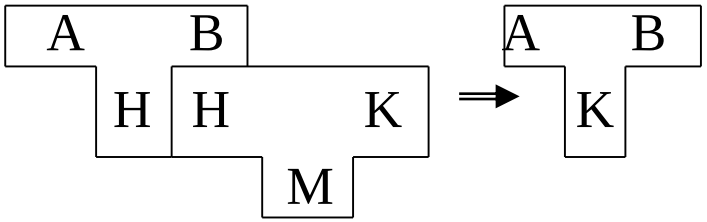
- T-Diagram can be combined in two basic ways.

The First T-diagram Combination



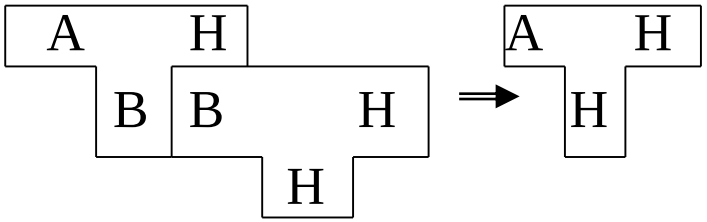
- Two compilers run on the same machine H
  - First from A to B
  - Second from B to C
  - Result from A to C on H

The Second T-diagram Combination



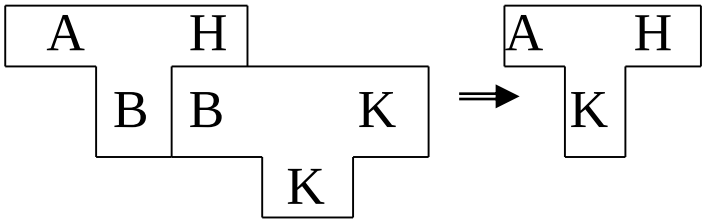
- Translate implementation language of a compiler from H to K
- Use another compiler from H to K

The First Scenario



- Translate a compiler from A to H written in B
  - Use an existing compiler for language B on machine H

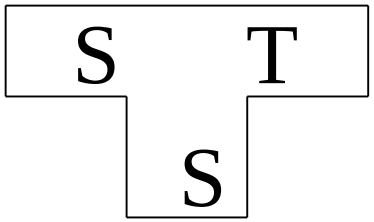
The Second Scenario



- Use an existing compiler for language B on different machine K
  - Result in a cross compiler

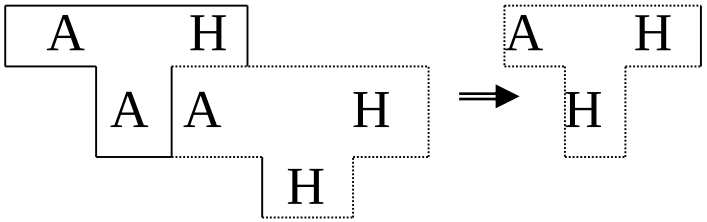
Process of Bootstrapping

- Write a compiler in the same language



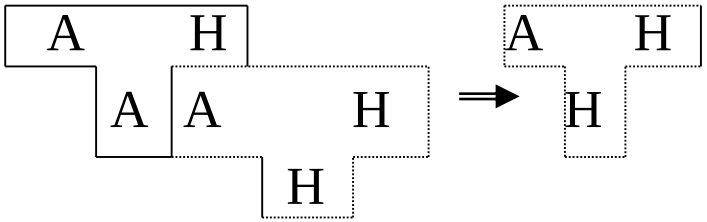
- No compiler for source language yet
- Porting to a new host machine

The First step in bootstrap



- “quick and dirty” compiler written in machine language H
- Compiler written in its own language A
- Result in running but inefficient compiler

The Second step in bootstrap



- Running but inefficient compiler
- Compiler written in its own language A
- Result in final version of the compiler