中山大學 SUN YAT-SEN UNIVERSITY　国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compilation Principle
编 译 原 理

第18讲：中间代码(3)

张献伟

xianweiz.github.io

DCS290, 5/20/2021

Review Questions

- What is offset, and how do we use it?

  Offset is the relative address. Increment it after processing a variable.

- What is (IR) code generation?

  For variable definitions, lay out memory.
  For statements, translate into three-address code.

- Attributes *code* and *addr*?

  $E \rightarrow E_1 + E_2$; { E.addr = newtemp();
  E.code = E_1.code || E_2.code ||
  gen(E.addr '=' E_1.addr '+' E_2.addr); }

  Code: the TAC; addr: the address holding the value

- What is incremental translation (增量翻译)?

  Generate only the new TAC instructions, skipping over the copy.

- Type(a) = array(4, array(8, array(5, int))), addr(a[i][j][k]?

  addr(a[i][j][k]) = base + i*160 + j*20 + k*4

CodeGen: Boolean Expressions[布尔表达式]

- Boolean expression: a *op* b
  - where op can be <, <=, = !=, > or >=, &&, ||, …
- **Short-circuit** evaluation[短路计算]: to skip evaluation of the rest of a boolean expression once a boolean value is known
  - Given following C code: *if (flag || foo()) { bar(); };*
    - If *flag* is true, *foo()* never executes
    - Equivalent to: *if (flag) { bar(); } else if (foo()) { bar(); };*
  - Given following C code: *if (flag && foo()) { bar(); };*
    - If *flag* is false, *foo()* never executes
    - Equivalent to: *if (!flag) { } else if (foo()) { bar(); };*
  - Used to alter control flow, or compute logical values
    - Examples: *if (x < 5) x = 1*; *x = true*; *x = a < b*
    - For control flow, boolean operators translate to ***jump*** statements

Boolean Exprs (w/o Short-Circuiting)

- Computed just like any other arithmetic expression

E -> (a < b) or (c < d and e < f)

t_1 = a < b
t_2 = c < d
t_3 = e < f
t_4 = t_2 && t_3
t_5 = t_1 || t_4

- Then, used in control-flow statements
  - *S.next*: label for code generated after *S*

S -> if E S_1

if (!t_5) goto *S.next*
S_1.code
S.next: ...

Boolean Exprs (w/ Short-Circuiting)

- Implemented via a series of jumps [利用跳转]
  - Each relational op converted to two gotos (*true* and *false*)
  - Remaining evaluation skipped when result known in middle
- Example
  - *E.true*: label for code to execute when *E* is *'true'*
  - *E.false*: label for code to execute when *E* is *'false'*
  - E.g. if above is condition for a *while* loop
    - *E.true* would be label at beginning of loop body
    - *E.false* would be label for code after the loop

E -> (a < b) or (c < d and e < f)

```
    if (a < b) goto E.true
    goto L1
L1: if (c < d) goto L2
    goto E.false
L2: if (e < f) goto E.true
    goto E.false
```

SDT Translation of Booleans

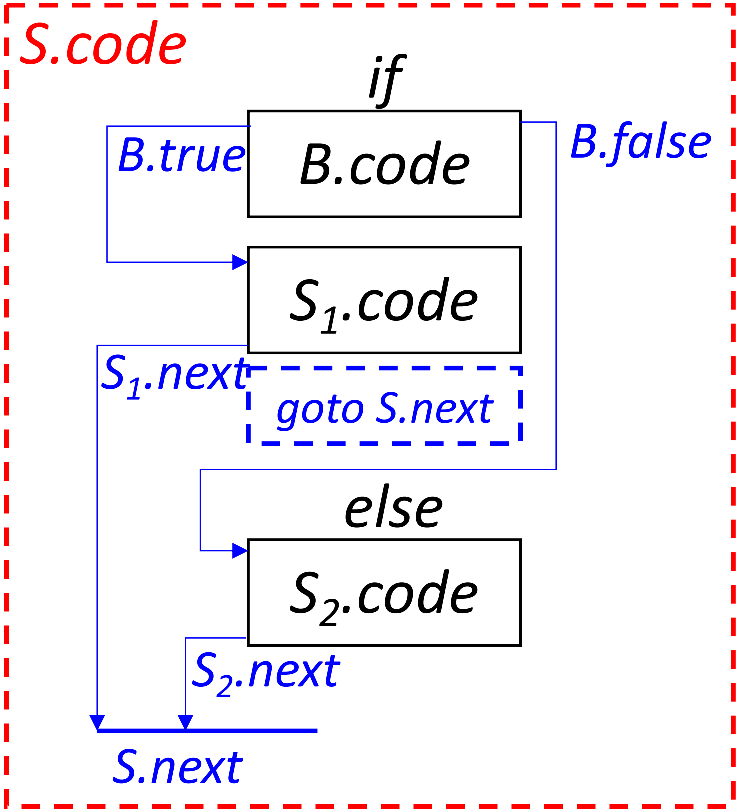
- *B* -> B_1 || B_2
  - B_1*.true* is same as *B.true*, B_2 must be evaluated if B_1 is false
  - The true and false exits of B_2 are the same as *B*
- *B* -> E_1 *relop* E_2
  - Translated directly into a comparison TAC inst with jumps

① *B* -> { B_1*.true = B.true;* B_1*.false = newlabel();* } B_1
|| { *label(*B_1*.false);* B_2*.true = B.true;* B_2*.false = B.false;* } B_2
② *B* -> { B_1*.true = newlabel();* B_1*.false = B.false;* } B_1
&& { *label(*B_1*.true);* B_2*.true = B.true;* B_2*.false = B.false;* } B_2
③ *B* -> E_1 *relop* E_2 { *gen('if'* E_1*.addr relop* E_2*.addr 'goto' B.true);*
gen('goto' B.false; }
④ *B* -> ! { B_1*.true = B.false;* B_1*.false = B.true;* } B_1
⑤ *B* -> true { *gen('goto' B.true;* }
⑥ *B* -> false { *gen('goto' B.false;* }

CodeGen: Control Statement[控制语句]

① S -> if (B) S_1
② S -> if (B) S_1 else S_2
③ S -> while (B) S_1

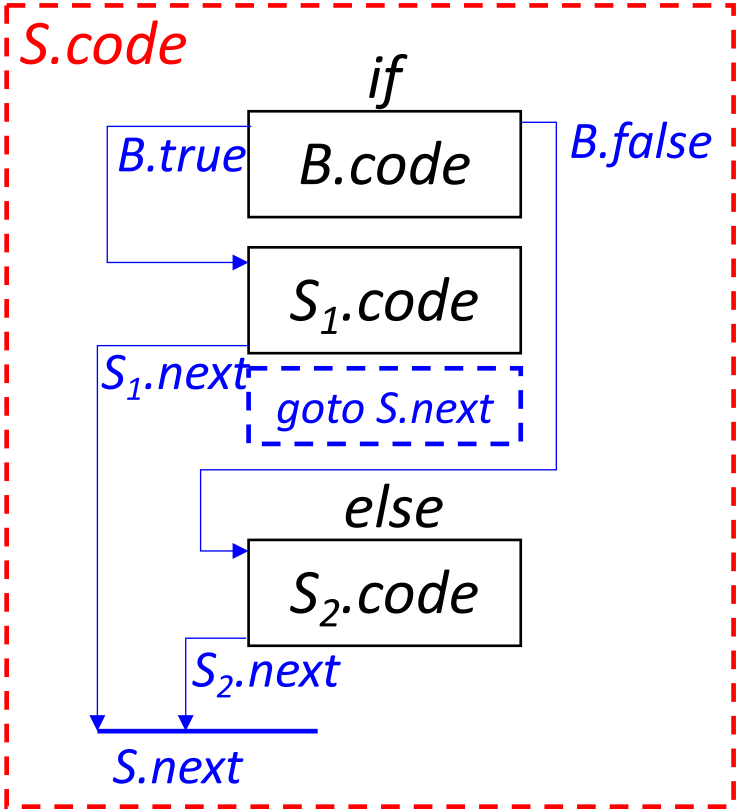
- Inherited attributes [继承属性]
  - *B.true*: the label to which control flows if *B* is true
  - *B.false*: the label to which control flows if *B* is false
  - *S.next*: a label for the instruction immediately after the code of *S*



Translation of Controls

① S -> if (B) S_1
② S -> if (B) S_1 else S_2
③ S -> while (B) S_1

S -> *if* { *B.true = newlabel();*
B.false = newlabel(); }
(*B*) { *label(B.true);* S_1*.next = S.next;* }
S_1 { *gen('goto' S.next);* }
else { *label(B.false);* S_2*.next = S.next;* } S_2



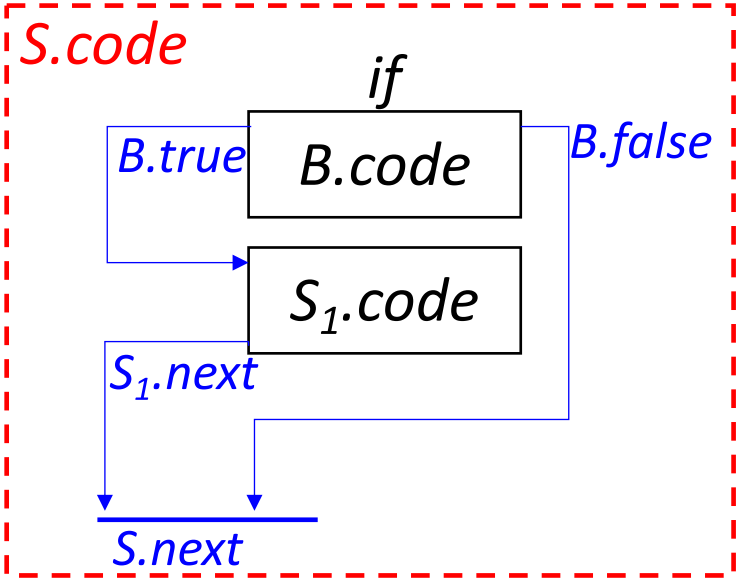
- Helper functions
  - *newlabel()*: creates a new label
  - *label(L)*: attaches label *L* to the next three-address inst to be generated

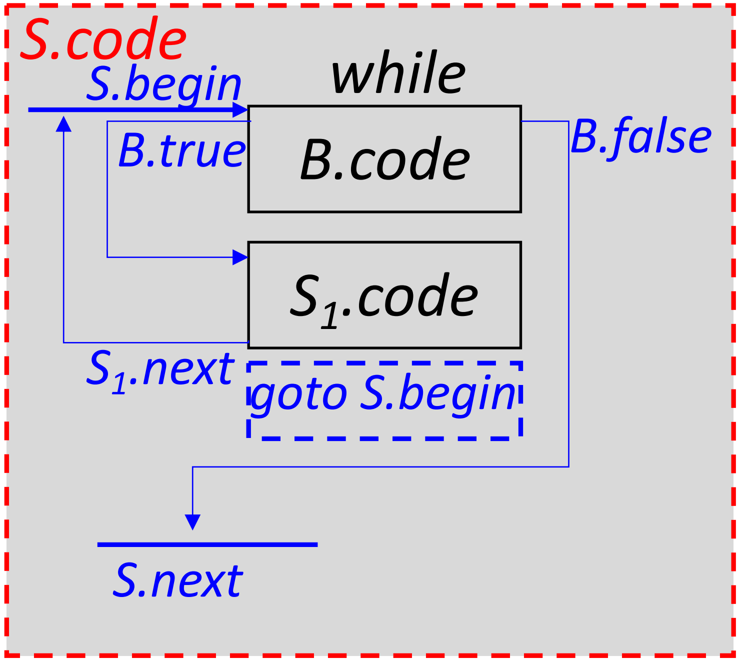
Translation of Controls (cont.)

① S -> if (B) S_1
② S -> if (B) S_1 else S_2
③ S -> while (B) S_1

S -> if { *B.true = newlabel();*
B.false = S.next; }
(*B*) { *label(B.true);* S_1*.next = S.next;* }
S_1

S -> while { *S.begin = newlabel();*
label(S.begin);
B.true = newlabel();
B.false = S.next; }
(*B*) { *label(B.true);* S_1*.next = S.begin;* }
S_1 { *gen('goto' S.begin);* }





Jumping Labels[跳转标签]

- Key of generating code for Boolean and flow-control: matching a jump inst with the target of jump
  - Forward jump: a jump to an instruction below you
  - Label for jump target has not yet been generated
  - The labels are not *L-attributed*

B -> { B_1.true = newlabel(); B_1.false = B.false; } B_1
&& { label(B_1.true); B_2.true = B.true; B_2.false = B.false; } B_2

S -> *if* { B.true = newlabel();
B.false = S.next; }
(B) { label(B.true); S_1.next = S.next; }
S_1

Handle Non-L-Attribute Labels

- Idea: generate code using dummy labels first then patch them with addresses later after labels are generated

- **Two-pass** approach: requires two scans of code
  - Pass 1:
    - Generate code creating dummy labels for forward jumps. (Insert label into a hashtable when created)
    - When label emitted, record address in hashtable.
  - Pass 2:
    - Replace dummy labels with target addresses (Use previously built hashtable for mapping)
- **One-pass** approach
  - Generate holes when forward jumping to a un-generated label
  - Maintain a list of holes for that label
  - Fill in holes with addresses when label generated later on

Two-Pass Code Generation

- **newlabel()**: generates a new dummy label
  - Label inserted into hashtable, initially with no address
- Pass 1: generate code with non-address-mapped labels
  - For *S -> if (B)* S_1:
    - Dummy labels: *B.true=newlabel(); B.false=S.next;*
    - Generate *B.code* using dummy labels *B.true*, *B.false*
    - Generate label *B.true*: in the process mapping it to an address
    - Generate S_1*.code* using dummy label S_1*.next*
- Pass 2: Replace labels with addresses using hashtable
  - Any forward jumps to dummy labels *B.true*, *B.false* are replaced with jump target addresses

```
S -> if { B.true = newlabel();
          B.false = S.next; }
     ( B ) { label(B.true); S1.next = S.next; }
     S1
```

One-Pass Code Generation

- If *L-attributed*, grammar can be processed in one pass
- However, forward jumps introduce *non-L-attributes*
  - E.g. $E_1.false = E_2.label$ in $E \rightarrow E_1 \,||\, E_2$
  - We need to know address of $E_2.label$ to insert jumps in E_1
  - Is there a general solution to this problem?
- Solution: **Backpatching** [回填]
  - Leave holes in IR in place of forward jump addresses
  - Record indices of jump instructions in a hole list
  - When target address of label for jump is eventually known, backpatch holes using the hole list for that particular label
- Can be used to handle any *non-L-attribute* in a grammar

Backpatching[回填]

- Synthesized attributes [综合属性]. *S -> if (B) S_1*
  - *B.truelist*: a list of jump or conditional jump insts into which we must insert the label to which control goes if *B* is true [B为真时控制流应该转向的指令的标号]
  - *B.falselist*: a list of insts that eventually get the label to which control goes when *B* is false [B为假时控制流应该转向的指令的标号]
  - *S.nextlist*: a list of jumps to the inst immediately following the code for *S* [紧跟在S代码之后的指令的标号]
- Functions to implement backpatching
  - *makelist(i)*: creates a new list out of statement index *i*
  - *merge(p_1, p_2)*: returns merged list of p_1 and p_2
  - *backpatch(p, i)*: fill holes in list *p* with statement index *i*

Backpatching (cont.)

- $B \text{ -> } B_1 \;||\; M\; B_2$
  - If B_1 is true, then B is also true
  - If B_1 is false, we must next test B_2, so the target for jump B_1*.falselist* must be the beginning of the code of B_2

① $B \text{ -> } E_1$ *relop* E_2 { *B.truelist = makelist(nextinst);*
B.falselist = makelist(nextinst+1);
gen('if' E_1*.addr relop* E_2*.addr 'goto _');*
gen('goto _'); }

② $B \text{ -> } B_1 \;||\; M\; B_2$ { *backpatch(*B_1*.falselist, M.inst);*
B.truelist = merge(B_1*.truelist,* B_2*.truelist);*
B.falselist = B_2*.falselist; }*

③ $B \text{ -> } B_1 \;\&\&\; M\; B_2$ { *backpatch(*B_1*.truelist, M.inst);*
B.truelist = B_2*.truelist;*
B.falselist = merge(B_1*.falselist,* B_2*.falselist); }*

④ $M \text{ -> } \varepsilon$ { *M.inst = nextinst; }*

Example

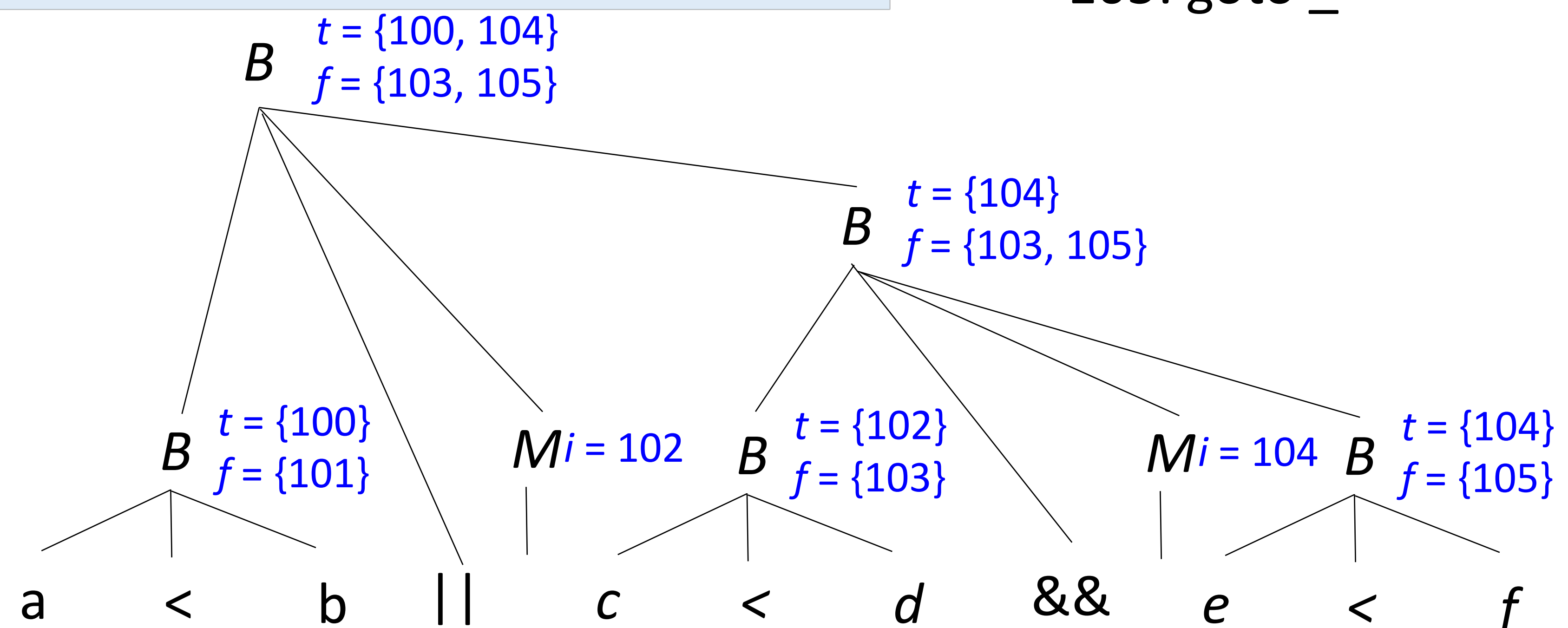
① $B \rightarrow E_1$ *relop* E_2 { *B.truelist = makelist(nextinst);*
B.falselist = makelist(nextinst+1);
gen('if' E_1*.addr relop* E_2*.addr 'goto _');*
gen('goto _'); }

② $B \rightarrow B_1$ || *M* B_2 { *backpatch(*B_1*.falselist, M.inst);*
B.truelist = merge(B_1*.truelist,* B_2*.truelist);*
B.falselist = B_2*.falselist; }*

③ $B \rightarrow B_1$ && *M* B_2 { *backpatch(*B_1*.truelist, M.inst);*
B.truelist = B_2*.truelist;*
B.falselist = merge(B_1*.falselist,* B_2*.falselist); }*

④ $M \rightarrow \varepsilon$ { *M.inst = nextinst;* }

100: if a < b: goto _
101: goto 102
102: if c < d: goto 104
103: goto _
104: if e < f: goto _
105: goto _



Backpatching of Control-Flow

- *S.nextlist*: a list of all jumps to the inst following S

① $S \rightarrow$ *if* (B) M S_1 { *backpatch(B.truelist, M.inst)*
S.nextlist = merge(B.falselist, S_1*.nextlist); }*

② $S \rightarrow$ *if* (B) M_1 S_1 N *else* M_2 S_2 { *backpatch(B.truelist,* M_1*.inst);*
backpatch(B.falselist, M_2*.inst);*
temp = merge(S_1*.nextlist, N.nextlist);*
S.nextlist = merge(temp, S_2*.nextlist); }*

③ $S \rightarrow$ *while* M_1 (B) M_2 S_1 { *backpatch(*S_1*.nextlist,* M_1*.inst);*
backpatch(B.truelist, M_2*.inst);*
S.nextlist = B.falselist);
gen('goto' M_1*.inst); }*

④ $M \rightarrow \varepsilon$ { *M.inst = nextinst; }*

⑤ $N \rightarrow \varepsilon$ { *N.nextlist = makelist(nextinst);*
gen('goto _'); }

Summary

- Code generation: generate TAC instructions using syntax directed translation
  - Variable definitions [变量定义]
  - Expressions and statements
    - Assignment [赋值]
    - Array references [数组引用]
    - Boolean expressions [布尔表达式]
    - Control-flow [控制流]

- Translations not covered
  - Switch statements [switch语句]
  - Procedure calls [过程调用]

Compilation Principle
编 译 原 理

第18讲：运行时环境

张献伟

xianweiz.github.io

DCS290, 5/20/2021

Run-Time Environment[运行时环境]

- Programming languages contain high-level structures
  - Functions, objects, exceptions, loops, …
- The physical computer only operates in terms of several primitive operations
  - Arithmetic
  - Data movement
  - Control jumps
- We need to represent these high-level structures using the low-level structures of the machine
  - A set of data structures maintained at runtime to implement these high-level structures

Run-Time Environment (cont.)

- **Runtime Environment**: the 'environment' in which the program executes in at runtime [运行时环境]
  - Includes HW: CPU, main memory, ...
  - Includes OS: environment variables, ...
  - Includes Runtime Libraries: C Runtime Library (libc), ...
- When a program is invoked [程序被调用]
  - The OS allocates memory for the program
  - Program code and data is loaded into memory
  - Program initializes runtime environment
  - Program jumps to entry point 'main()'
- All program binaries include two parts
  - Code implementing semantics of program
  - Runtime code

Runtime Code[运行时代码]

- **Runtime code**: any code not implementing semantics
  - Code to manage runtime environment
    - Manage memory storage (e.g. heap/stack)
    - Manage CPU register storage
    - Manage multiple CPUs (for languages with threading)
  - Code to implement language execution model
    - Code to pass function arguments according to model
    - Code to do dynamic type checking (if applicable)
    - Code to ensure security (if applicable)
  - May even include compiler itself! (just-in-time compiler)
- Some runtime codes are pre-fabricated libraries
  - E.g. heap data management, threading library ...
- Some generated by compiler, interleaved in program code
  - E.g. stack data management, register management, argument passing, type checking, ...

Runtime Code for Memory Management

- Three types of data that need to be stored in memory
  - ①Data with **static** lifetimes (duration of program)
    - E.g. global variables, static local variables, program code
  - ②Data with **scoped** lifetimes (within given scope)
    - E.g. local variables, function parameters
  - ③Data with **arbitrary** lifetimes (on-demand alloc/free)
    - E.g. malloc()/free(), new/delete
- ① and ② are called **named memory**
  - Has either variable or function name associated with data
  - For code gen, compiler must know address for each name
    - Compiler must lay out named memory at compile time
    - Compiler must also generate memory management code
- ③ is called **unnamed memory**
  - Pointers may point to data, but data itself is anonymous
  - Can be managed by runtime library, not involving compiler