南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 11

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Translating expressions with array references
- The backpatching technique
- Project phase 3 introduction

Dealing with Arrays (Lab)

- An expression involve array accesses: $c + a[i][j]$
- An array reference $A[i][j]$ will expand into a sequence of three-address instructions that calculate an address for the reference

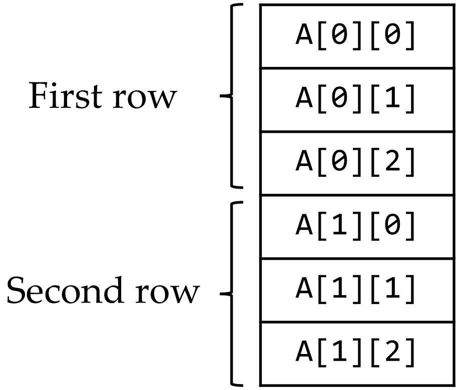
```
c + a[i][j]
```

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
```

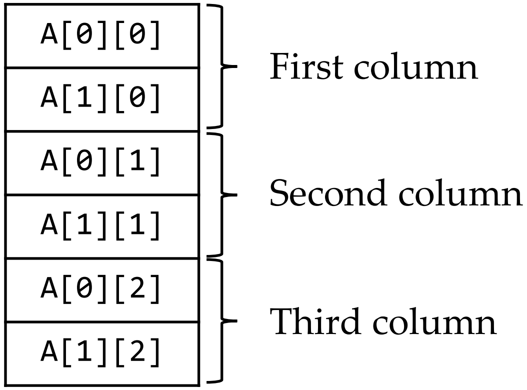
calculate address

Addressing Array Elements

- Array elements can be accessed quickly if they are stored consecutively
- For an array A with n elements, the relative address of $A[i]$ is:
  - $\boldsymbol{base + i * w}$ ($base$ is the relative address of $A[0]$, w is the width of an element)
- For a 2D array A (row-major layout), the relative address of $A[i_1][i_2]$ is:
  - $\boldsymbol{base + i_1 * w_1 + i_2 * w_2}$ (w_1 is the width of a row, w_2 is the width of an element)



Row-major (C)



Column-major (Fortran)

Addressing Array Elements

- Array elements can be accessed quickly if they are stored consecutively
- For an array A with n elements, the relative address of $A[i]$ is:
  - $\boldsymbol{base + i * w}$ ($base$ is the relative address of $A[0]$, w is the width of an element)
- For a 2D array A (row-major layout), the relative address of $A[i_1][i_2]$ is:
  - $\boldsymbol{base\ +\ i_1 * w_1 + i_2 * w_2}$ (w_1 is the width of a row, w_2 is the width of an element)
- Further generalize to k-dimensional array A (row-major layout), the relative address of $A[i_1][i_2] \ldots [i_k]$ is:
  - $\boldsymbol{base\ +\ i_1 * w_1 + i_2 * w_2 + \cdots + i_k * w_k}$ (w's can be generalized as above)

Translation of Array References

- The main problem in generating code for array references is to relate the address-calculation formula to the grammar

  - The relative address of $A[i_1][i_2] \ldots [i_k]$ is $base + i_1 * w_1 + i_2 * w_2 + \cdots + i_k * w_k$

  - Productions for generating array references: $L \rightarrow L\ [\ E\ ]\ |\ \mathbf{id}\ [\ E\ ]$

SDT for Array References (1)

$$
\begin{array}{lll}
L & \rightarrow \; \textbf{id} \; [\; E \;] & \{\; L.array = top.get(\textbf{id}.lexeme); \\
& & \quad L.type = L.array.type.elem; \\
& & \quad L.addr = \textbf{new}\; Temp\,(); \\
& & \quad gen(L.addr \;'='\; E.addr \;'*'\; L.type.width); \;\} \\
& \mid \; L_1 \; [\; E \;] & \{\; L.array = L_1.array; \\
& & \quad L.type = L_1.type.elem; \\
& & \quad t = \textbf{new}\; Temp\,(); \\
& & \quad L.addr = \textbf{new}\; Temp\,(); \\
& & \quad gen(t \;'='\; E.addr \;'*'\; L.type.width); \\
& & \quad gen(L.addr \;'='\; L_1.addr \;'+'\; t); \;\}
\end{array}
$$

Compute offset

A is a 2*3 array of integers
Translate A[i][j]

L.type is the type of A's element:
array(3, int)

A[i]
↓ Reduce using prod. #1
L

L[j]
↓ Reduce using prod. #2
L

L.type is the type of A[i]'s element:
int

$L.array$: a pointer to the symbol-table entry for the array name

$L.array.base$: the base address of the array

$L.addr$: a temporary for computing the <u>offset</u> for the array reference

$L.type$: the type of the subarray generated by L

$t.elem$: for any array type t, $t.elem$ gives the element type

SDT for Array References (2)

- The semantic actions of L-productions compute offsets
- The address of an array element is $base + offset$

$$
\begin{array}{lll}
E & \rightarrow E_1 + E_2 & \{ E.addr = \textbf{new } Temp(); \\
 & & \quad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \} \\
 & \mid \; \textbf{id} & \{ E.addr = top.get(\textbf{id}.lexeme); \} \\
 & \mid \; L & \{ E.addr = \textbf{new } Temp(); \\
 & & \quad gen(\underline{E.addr \; '=' \; L.array.base \; '[' \; L.addr \; ']'}); \}
\end{array}
$$

Instruction of the form $x = a[i]$

Array references can be part of an expression

SDT for Array References (3)

$$
\begin{aligned}
S \;\rightarrow\; & \mathbf{id} = E \;; && \{\; gen(\; top.get(\mathbf{id}.lexeme) \;'='\; E.addr); \;\} \\
\mid\;\; & L = E \;; && \{\; gen(L.addr.base \;'['\; L.addr \;']'\; '='\; E.addr); \;\}
\end{aligned}
$$

Instruction of form $a[i] = x$

Array references can appear at the LHS of an assignment statement

$E \rightarrow E_1 + E_2$ { $E.addr$ = **new** $Temp$(); $gen(E.addr$ '=' $E_1.addr$ '+' $E_2.addr)$; } 6

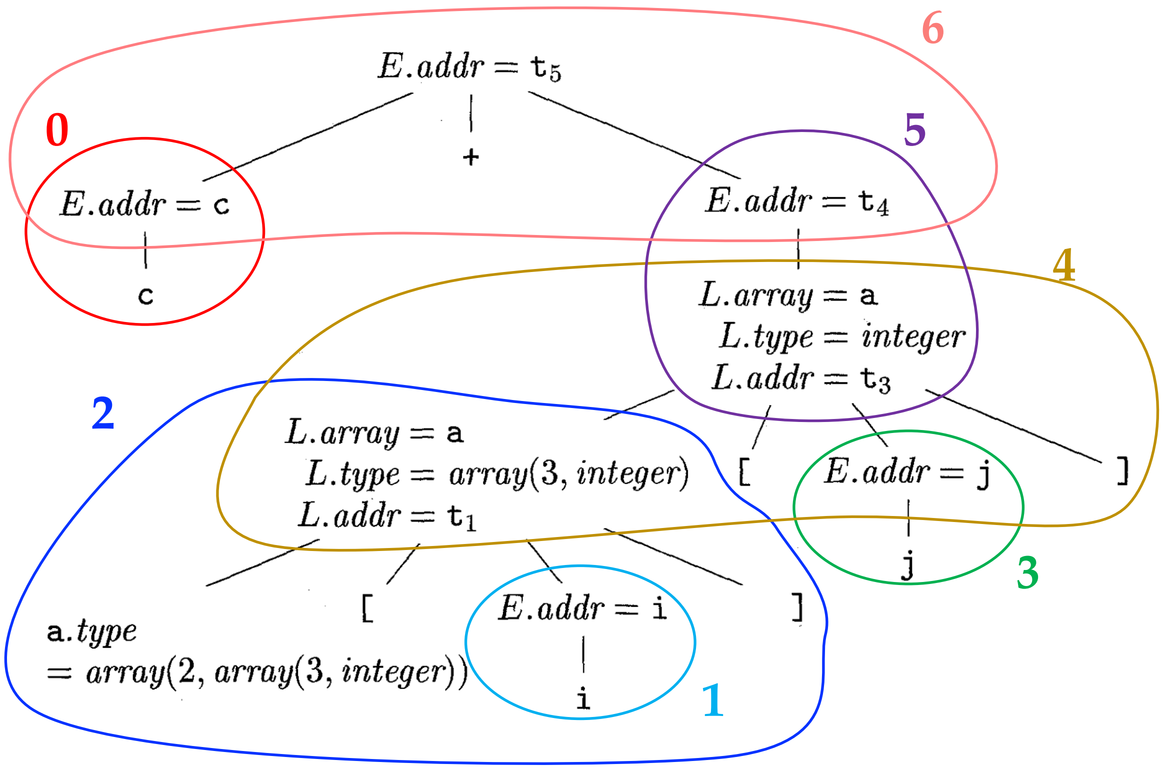
| **id** { $E.addr$ = $top.get($**id**$.lexeme)$; } 0 1 3

| L { $E.addr$ = **new** $Temp$(); $gen(E.addr$ '=' $L.array.base$ '[' $L.addr$ ']'$)$; } 5

$L \rightarrow$ **id** [E] { $L.array$ = $top.get($**id**$.lexeme)$; $L.type$ = $L.array.type.elem$; $L.addr$ = **new** $Temp$(); $gen(L.addr$ '=' $E.addr$ '*' $L.type.width)$; } 2

| L_1 [E] { $L.array$ = $L_1.array$; $L.type$ = $L_1.type.elem$; t = **new** $Temp$(); $L.addr$ = **new** $Temp$(); $gen(t$ '=' $E.addr$ '*' $L.type.width)$; $gen(L.addr$ '=' $L_1.addr$ '+' $t)$; } 4

Translating `c + a[i][j]`



Generated code:

Code	
`t1 = i * 12`	2
`t2 = j * 4`	4
`t3 = t1 + t2`	4
`t4 = a[t3]`	5
`t5 = c + t4`	6

Outline

- Translating expressions with array references
- The backpatching technique
- Project phase 3 introduction

Backpatching (回填)

- A **key problem** when generating code for boolean expressions and flow-of-control statements is to match a jump instruction with the jump target

- **Example: if (*B*) *S***
  - According to the short-circuit translation, B's code contains a jump to the instruction following the code for S (executed when B is false)
  - However, B must be translated before S. The jump target is unknown when translating B
  - Earlier, we address the problem by passing labels as inherited attributes ($S.next$), but this requires another separate pass (traversing the parse tree) after parsing

How to address the problem in one pass?

One-Pass Code Generation Using Backpatching

- **Basic idea of backpatching (基本思想):**
  - When a jump is generated, its target is temporarily left unspecified.
  - Incomplete jumps are grouped into lists. All jumps on a list have the same target.
  - Fill in the labels for incomplete jumps when the targets become known.

- **The technique (技术细节):**
  - For a nonterminal B that represents a boolean expression, we define two synthesized attributes: $truelist$ and $falselist$
  - $truelist$: a list of jump instructions whose target is the jump target when B is true
  - $falselist$: a list of jump instructions whose target is the jump target when B is false

One-Pass Code Generation Using Backpatching

- **The technique (技术细节) Cont.:**
  - $makelist(i)$: create a new list containing only i, the index of a jump instruction, and return the pointer to the list
  - $merge(p_1, p_2)$: concatenate the lists pointed by p_1 and p_2, and return a pointer to the concatenated list
  - $backpatch(p, i)$: insert i as the target for each of the jump instructions on the list pointed by p

Backpatching for Boolean Expressions (布尔表达式的回填)

- An SDT suitable for generating code for boolean expressions during bottom-up parsing

- Grammar:
  - $B \to B_1 \parallel M B_2 \mid B_1 \,\&\&\, M B_2 \mid\, !\, B_1 \mid (B_1) \mid E_1\ \textbf{rel}\ E_2 \mid \textbf{true} \mid \textbf{false}$
  - $M \to \epsilon$

Keep this question in mind: Why do we introduce M before B_2?

1)	$B \to B_1 \;\|\|\; M \; B_2$	$\{$ *backpatch*$(B_1.\mathit{falselist}, M.\mathit{instr});$ $B.\mathit{truelist} = \mathit{merge}(B_1.\mathit{truelist}, B_2.\mathit{truelist});$ $B.\mathit{falselist} = B_2.\mathit{falselist};\ \}$
2 2)	$B \to B_1 \;\&\&\; M \; B_2$	$\{$ *backpatch*$(B_1.\mathit{truelist}, M.\mathit{instr});$ $B.\mathit{truelist} = B_2.\mathit{truelist};$ $B.\mathit{falselist} = \mathit{merge}(B_1.\mathit{falselist}, B_2.\mathit{falselist});\ \}$
3)	$B \to \;!\; B_1$	$\{\ B.\mathit{truelist} = B_1.\mathit{falselist};$ $B.\mathit{falselist} = B_1.\mathit{truelist};\ \}$
4)	$B \to (\ B_1\ )$	$\{\ B.\mathit{truelist} = B_1.\mathit{truelist};$ $B.\mathit{falselist} = B_1.\mathit{falselist};\ \}$
1 5)	$B \to E_1 \ \mathbf{rel}\ E_2$	$\{\ B.\mathit{truelist} = \mathit{makelist}(\mathit{nextinstr});$ $B.\mathit{falselist} = \mathit{makelist}(\mathit{nextinstr}+1);$ *gen*('`if`' $E_1.\mathit{addr}$ **rel**.*op* $E_2.\mathit{addr}$ '`goto _`'); *gen*('`goto _`'); }
6)	$B \to \mathbf{true}$	$\{\ B.\mathit{truelist} = \mathit{makelist}(\mathit{nextinstr});$ *gen*('`goto _`'); }
7)	$B \to \mathbf{false}$	$\{\ B.\mathit{falselist} = \mathit{makelist}(\mathit{nextinstr});$ *gen*('`goto _`'); }
8)	$M \to \epsilon$	$\{\ M.\mathit{instr} = \mathit{nextinstr};\ \}$

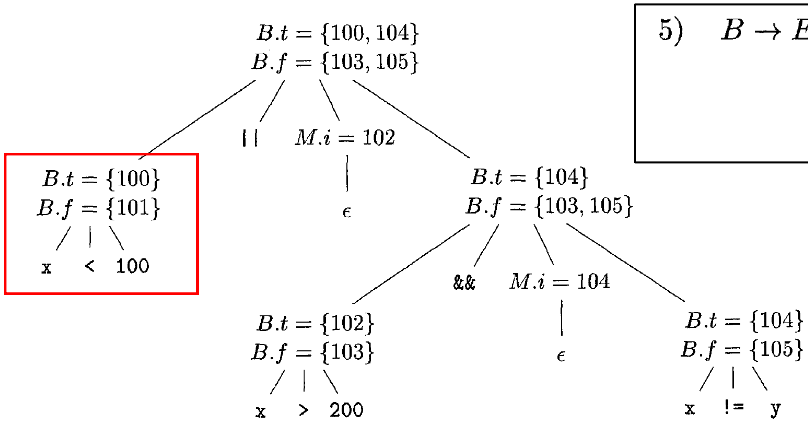
When finishing processing B1 && B2, we know the jump target for B1.truelist

When finishing processing E1 rel E2, we do not know the jump targets, so generate incomplete instructions first

Tip: understand 1 and 2 at a high level first and then revisit this slide after you understand the later examples.

Example – Boolean Expressions

- The earlier SDT is a postfix SDT. The semantic actions can be performed during a bottom-up parse.
- Boolean expression: $x\ <\ 100 \parallel x > 200\ \&\&\ x\ != y$
- **Step 1:** reduce $x\ <\ 100$ to B by production (5)



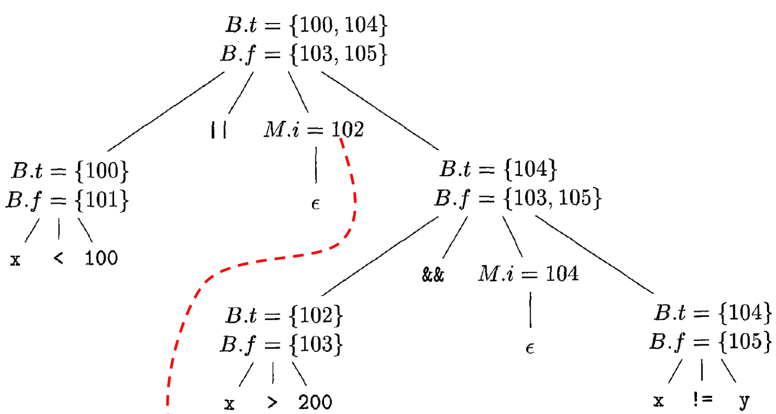
5)	$B \rightarrow E_1$ **rel** E_2	$\{$ $B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ *gen*('`if`' $E_1.addr$ **rel**$.op$ $E_2.addr$ '`goto _`'); *gen*('`goto _`'); $\}$

Generated code:

```
100:  if x < 100 goto _
101:  goto _
```

Example – Boolean Expressions

- The earlier SDT is a postfix SDT. The semantic actions can be performed during a bottom-up parse.
- Boolean expression: $x < 100 \parallel x > 200 \;\&\&\; x \mathrel{!}= y$
- **Step 2:** reduce ϵ to M by production (8)

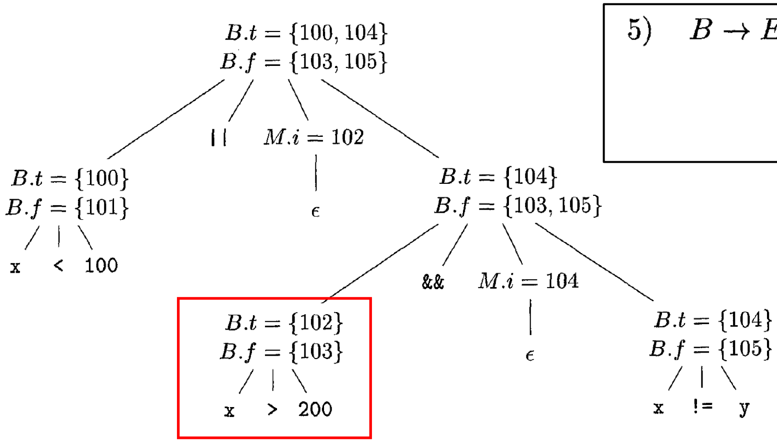


$$\boxed{8)\quad M \to \epsilon \qquad\qquad \{\ M.instr = nextinstr;\ \}}$$

The marker nonterminal records the value of *nextinstr*, 102

Example – Boolean Expressions

- Boolean expression: $x < 100 \parallel x > 200 \;\&\&\; x \mathrel{!}= y$
- **Step 3:** reduce $x > 200$ to B by production (5)



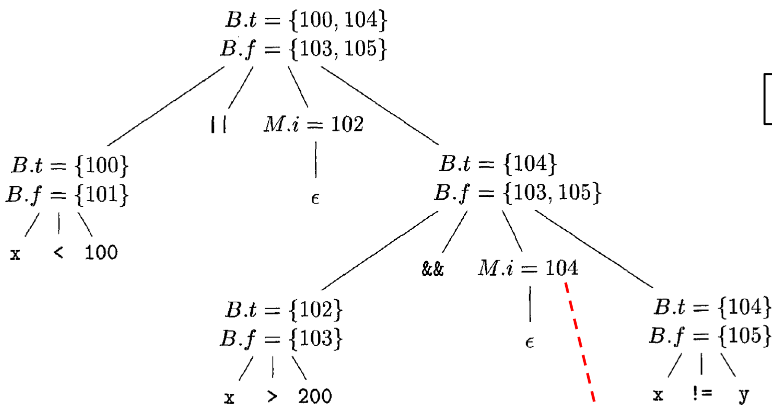
5) $B \rightarrow E_1$ **rel** E_2 $\{$ $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
gen('`if`' $E_1.addr$ **rel**.op $E_2.addr$ '`goto _`');
gen('`goto _`'); $\}$

Generated code:

```
102:  if x > 200 goto _
103:  goto _
```

Example – Boolean Expressions

- Boolean expression: $x < 100 \parallel x > 200 \; \&\& \; x \,!= y$
- **Step 4:** reduce ϵ to M by production (8)

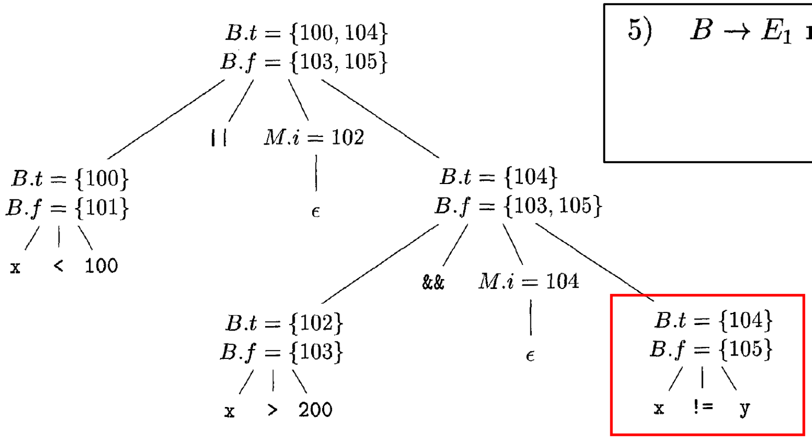


8)	$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$

The marker nonterminal records the value of *nextinstr*, 104

Example – Boolean Expressions

- Boolean expression: $x < 100 \parallel x > 200 \;\&\&\; x\,!= y$
- **Step 5:** reduce $x! = y$ to B by production (5)



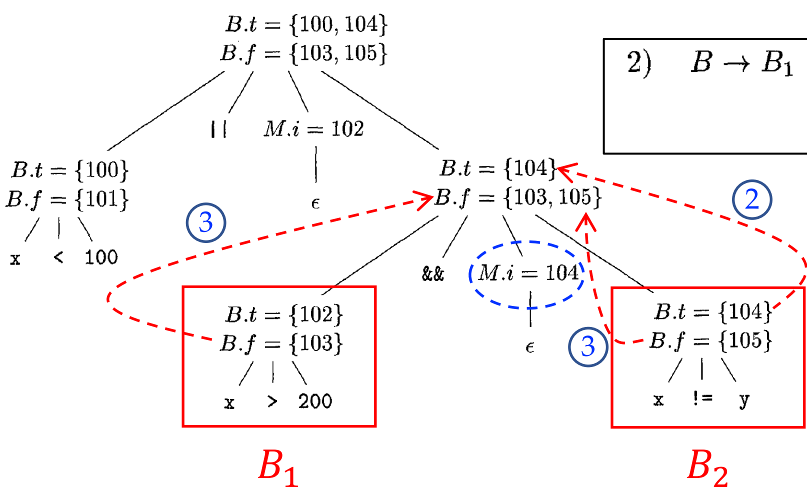
5)	$B \rightarrow E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ *gen*('`if`' $E_1.addr$ **rel**.op $E_2.addr$ '`goto _`'); *gen*('`goto _`'); }

Generated code:

```
104:   if x != y goto _
105:   goto _
```

Example – Boolean Expressions

- Boolean expression: $x < 100 \,\|\, x > 200 \,\&\&\, x\,! = y$
- **Step 6:** reduce B_1 && MB_2 to B by production (2)



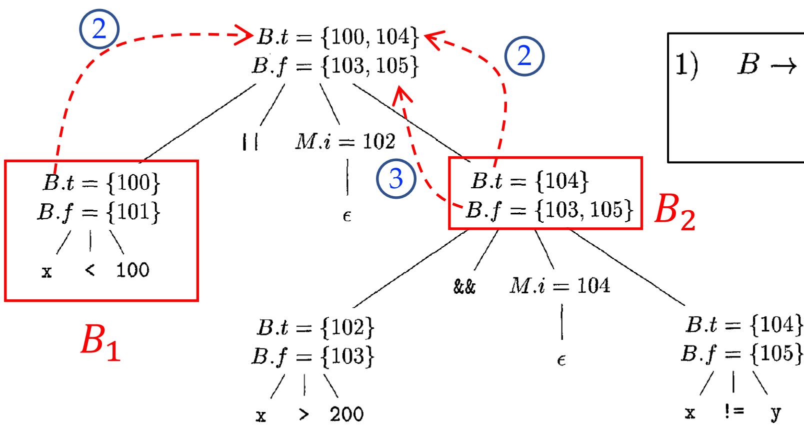
2) $B \rightarrow B_1$ && $M\ B_2$ { $backpatch(B_1.truelist, M.instr)$; 1
$B.truelist = B_2.truelist$; 2
$B.falselist = merge(B_1.falselist, B_2.falselist)$; } 3

Backpatch instruction 102: 1

```
100:   if x < 100 goto _
101:   goto _
102:   if x > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

Example – Boolean Expressions

- Boolean expression: $x \ < \ 100 \parallel x > 200 \ \&\& \ x \, ! = y$
- **Step 7:** reduce $B_1 \ \parallel \ MB_2$ to B by production (1)



1) $B \rightarrow B_1 \ || \ M \ B_2$ $\{$ $backpatch(B_1.falselist, M.instr);$ 1
$B.truelist = merge(B_1.truelist, B_2.truelist);$ 2
$B.falselist = B_2.falselist;$ $\}$ 3

Backpatch instruction 101: 1

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

The remaining jump targets will be filled in later parsing steps

Backpatching vs. Non-Backpatching (1)

(1) Non-backpatching SDD with inherited attributes:

$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$

(2) Backpatching scheme:

$B \rightarrow E_1$ **rel** E_2	$\{\ B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto \_}');$ $gen('\texttt{goto \_}');\ \}$

Comparison:

- In (2), incomplete instructions (指令坯) are added to corresponding lists
- The instruction jumping to $B.true$ in (1) is added to $B.truelist$ in (2)
- The instruction jumping to $B.false$ in (1) is added to $B.falselist$ in (2)

Backpatching vs. Non-Backpatching (2)

(1) Non-backpatching SDD with inherited attributes:

$B \rightarrow B_1 \;\|\|\; B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$

(2) Backpatching scheme:

$$
\begin{array}{ll}
B \rightarrow B_1 \;||\; M\; B_2 & \{\; backpatch(B_1.falselist, M.instr); \\
& \;\; B.truelist = merge(B_1.truelist, B_2.truelist); \\
& \;\; B.falselist = B_2.falselist; \;\}
\end{array}
$$

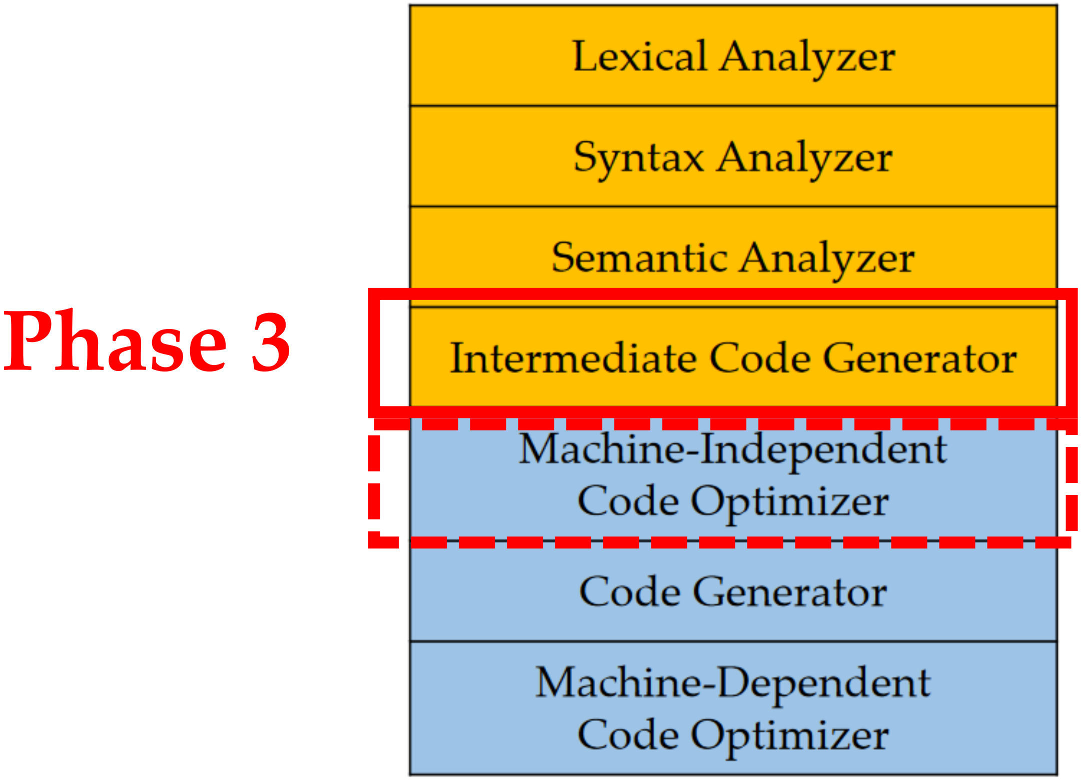
Comparison:

- The assignments to $true/false$ attributes in (1) correspond to the manipulations of $truelist/falselist$ in (2)

Outline

- Translating expressions with array references
- The backpatching technique
- Project phase 3 introduction

Phase 3 Overview



Goal

- Generate *intermediate representation* (IR) of a semantically valid SPL program, and do optimizations when possible

TAC Specification

Instruction	Description
`LABEL x :`	define a label x
`FUNCTION f :`	define a function f
`x := y`	assign value of y to x
`x := y + z`	arithmetic addition
`x := y - z`	arithmetic subtraction
`x := y * z`	arithmetic multiplication
`x := y / z`	arithmetic division
`x := &y`	assign address of y to x
`x := *y`	assign value stored in address y to x
`*x := y`	copy value y to address x
`GOTO x`	jump to label x without condition
`IF x [relop] y GOTO z`	if the condition (binary boolean) is true, jump to label z
`RETURN x`	exit the current function and return value x
`DEC x [size]`	allocate space pointed by x, size must be a multiple of 4
`PARAM x`	declare a function parameter
`ARG x`	pass argument x
`x := CALL f`	call a function, assign the return value to x
`READ x`	read x from somewhere
`WRITE x`	write x to somewhere

IR Simulator

To run `test_a.ir` with three integer inputs 1, 9, 42:

```
dist/irsim test_a.ir -i 1,9,42
```

```
SUSTech-CS323 IR-Simulator [test04.ir]

CODE

< step >  < exec >  < stop >

  t31 := v4 * #4
  t32 := &v3 + t31
  ARG &v2
  t33 := CALL add
  *t32 := t33
  t41 := v4 * #4
  t42 := &v3 + t41
  t35 := *t42
  WRITE t35
  v4 := v4 + #1
  v5 := #0
  GOTO label1
  LABEL label3 :
@ RETURN #0
```

```
SYMBOLS

t33 | 3
t35 | 3
t41 | 4
t42 | 12
 v2 | [1, 2]
 v3 | [1, 3]
 v4 | 2
 v5 | 0
```

```
[program output] 1
[program output] 3
[INFO] Total instructions = 81
```

Deadline & Grading

- 10:00 PM, December 24, 2023

- **Grading**

  Required test cases: 75 points

  Competitive score: 15 points ---------------

  <u>*Optional features: 10 points*</u>

Top 20%: 15 points
20~40%: 12 points
40~60%: 9 points
60~80%: 6 points
Below 80%: 3 points

Fewer instructions executed
during testing => ranked higher