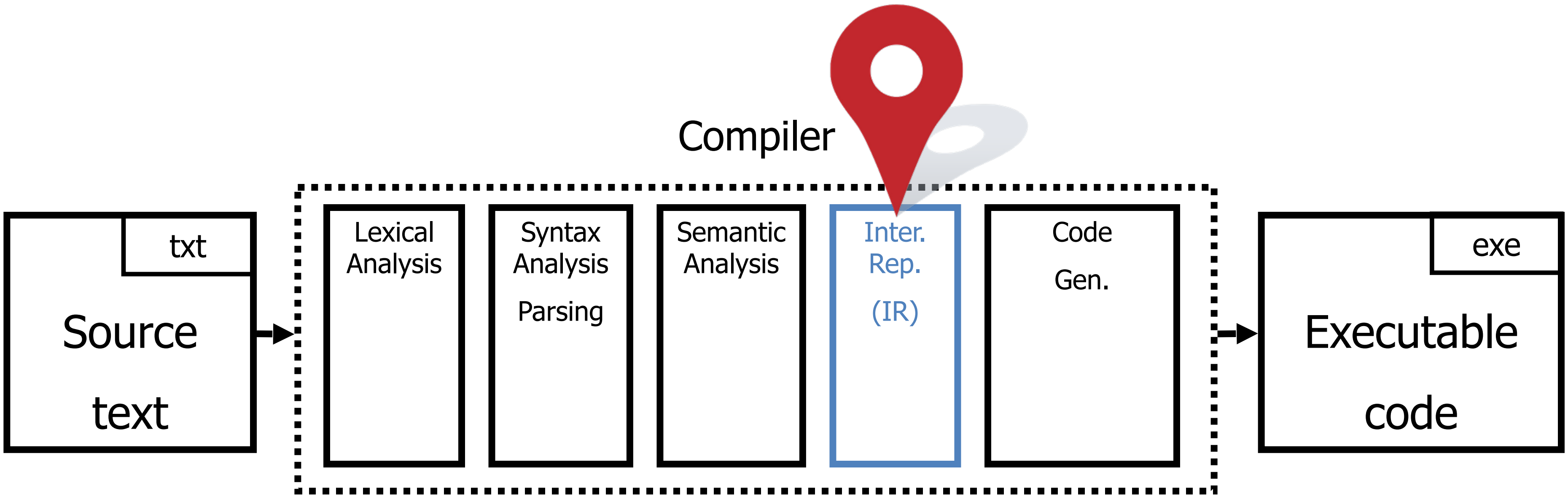
THEORY OF COMPILATION

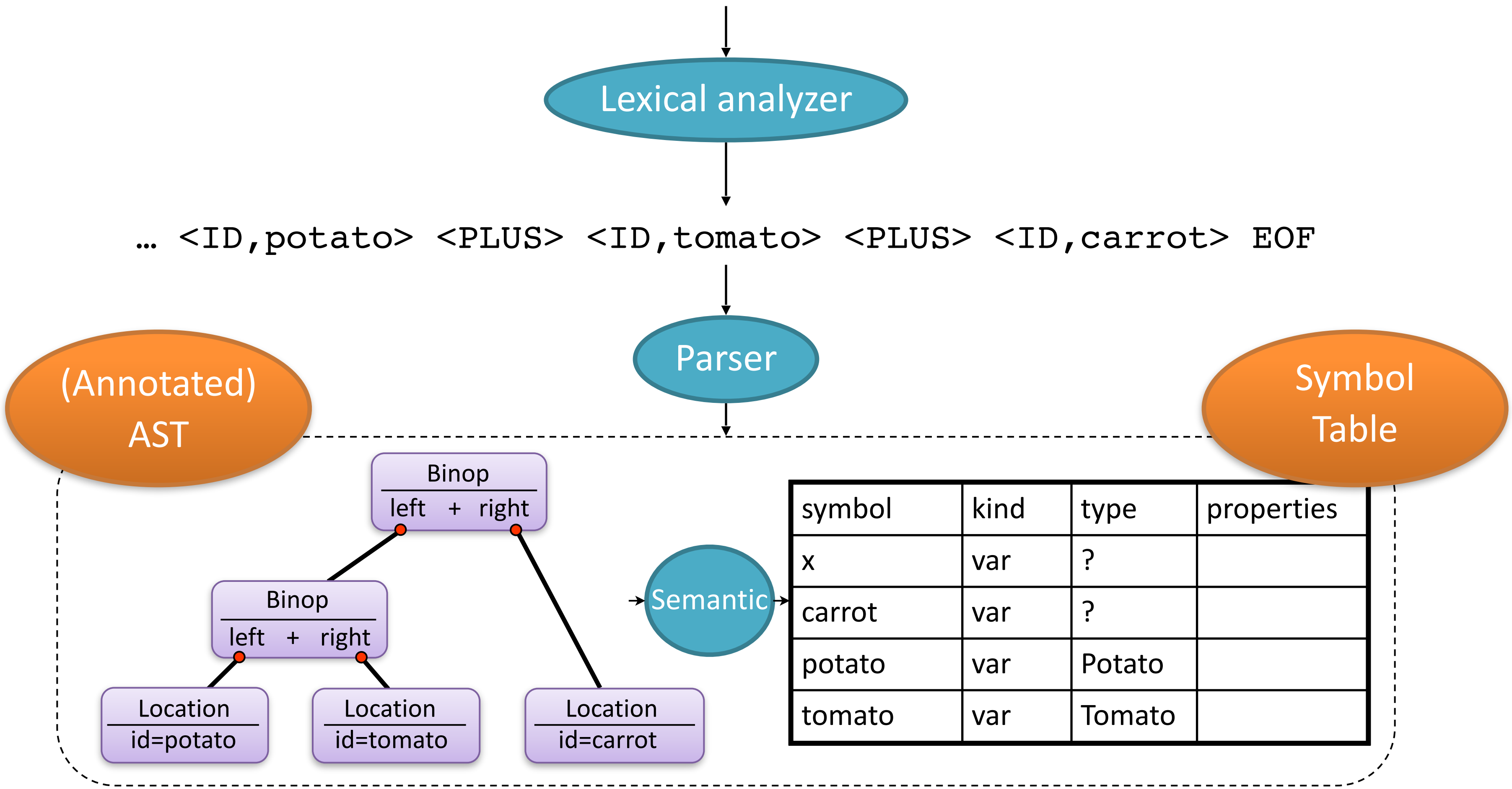
LECTURE 05

You are here


Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
Inter. Rep.
(IR)
Code Gen.
exe
Executable code

Reminder — Previous Steps

```
Potato potato;
Tomato tomato;
x = potato + tomato + carrot
```



symbol	kind	type	properties
x	var	?	
carrot	var	?	
potato	var	Potato	
tomato	var	Tomato	

‘carrot’ is undefined

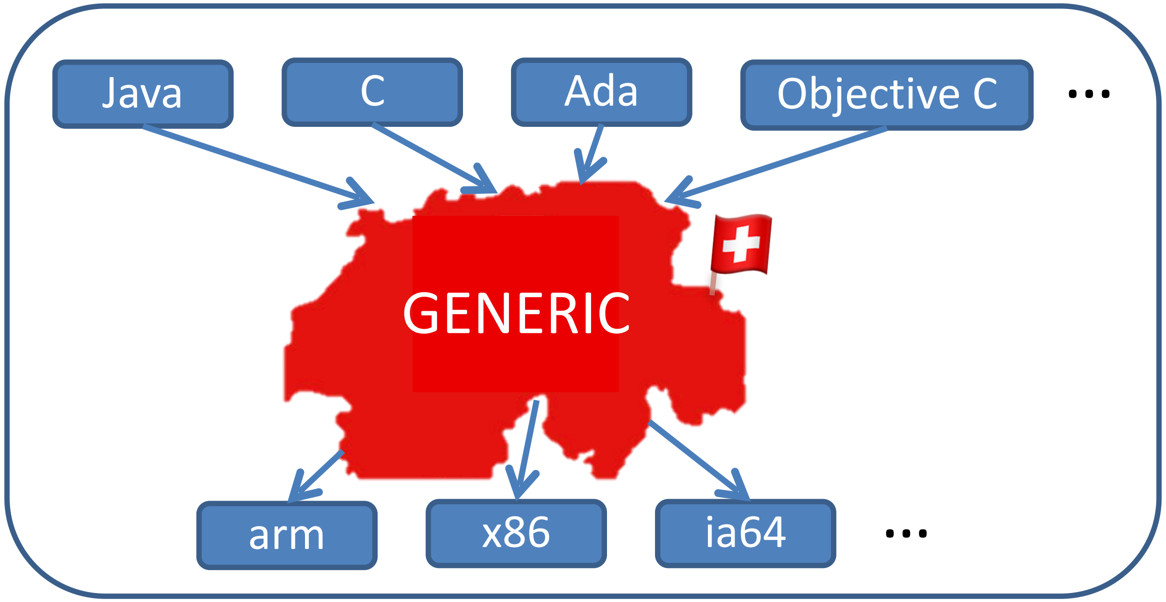
‘potato’ used before initialized

Cannot add ‘Potato’ and ‘Tomato’

Intermediate Representation

- "Neutral" representation between the front-end and the back-end
  - Abstracts away details of the source language
  - Abstract away details of the target language

- In practice, the IR may be biased toward a certain language (*e.g.*, gcc's GENERIC was created for C)



Intermediate Representation(s)

- Annotated abstract syntax tree
- Data dependence graph
- Three address code (3AC)
- …

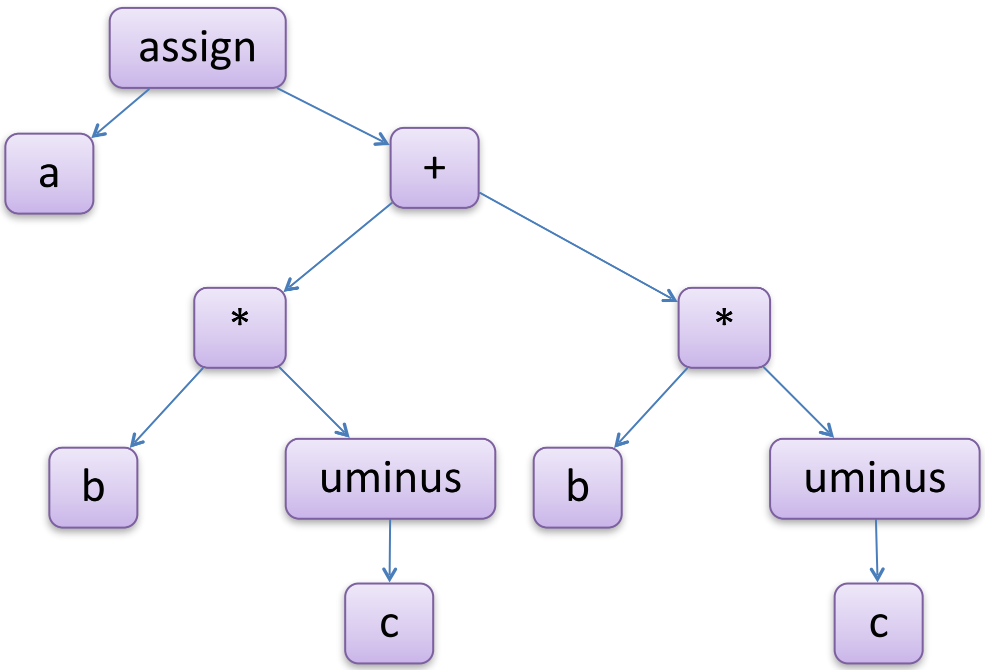
- A compiler may have ***multiple*** intermediate representations and move between them

Three Address Code (3AC)

- Every instruction operates on (at most) three addresses
  - **result** := **operand1** operator **operand2**
- Close to low-level operations in the machine language
  - Operator is a basic operation
- Statements in the source language may be mapped to multiple instructions in 3AC

Three Address Code — Example

AST



a = b * −c + b * −c

3AC

$t_1 := -\ c$

$t_2 := b\ *\ t_1$

$t_3 := -\ c$

$t_4 := b\ *\ t_3$

$t_5 := t_2\ +\ t_4$

$a := t_5$

Three Address Code

example instructions

instruction	meaning
x := y ◇ z	assignment with binary operator
x := ○ y	assignment with unary operator
x := y	assignment (copy)
x := &y	assign address of y
x := *y	assign value in address y (deref)
*x := y	assign into address x

(data)

instruction	meaning
goto L	unconditional jump
if x △ y **goto** L	conditional jump

(control)

Array Operations

- Are these 3AC operations?

```
x := y[i]
```

```
t1 := &y          ; t1 = address-of y
t2 := t1 + i      ; t2 = address of y[i]
x  := *t2         ; load value from y[i]
```

```
x[i] := y
```

```
t1  := &x         ; t1 = address-of x
t2  := t1 + i     ; t2 = address of x[i]
*t2 := y          ; store value at x[i]
```

Three Address Code — Example

```
int main(void) {
  int i;
  int b[10];
  for (i = 0; i < 10; ++i)
    b[i] = i*i;
}
```

```
     i := 0                 ; assignment
L1:  if i >= 10 goto L2     ; conditional jump
     t0 := i * i
     t1 := &b               ; t1 = address of b
     t2 := t1 + i           ; t2 = address of b[i]
     *t2 := t0              ; store i*i at b[i]
     i := i + 1
     goto L1
L2:
```

(example source: wikipedia)

Creating 3AC

- Assume bottom up parser
  - Why?

- Creating 3AC via **syntax directed translation**
  - Semantic attributes:

code	3-address code fragment generated for a nonterminal
var	name of variable that stores the result of code

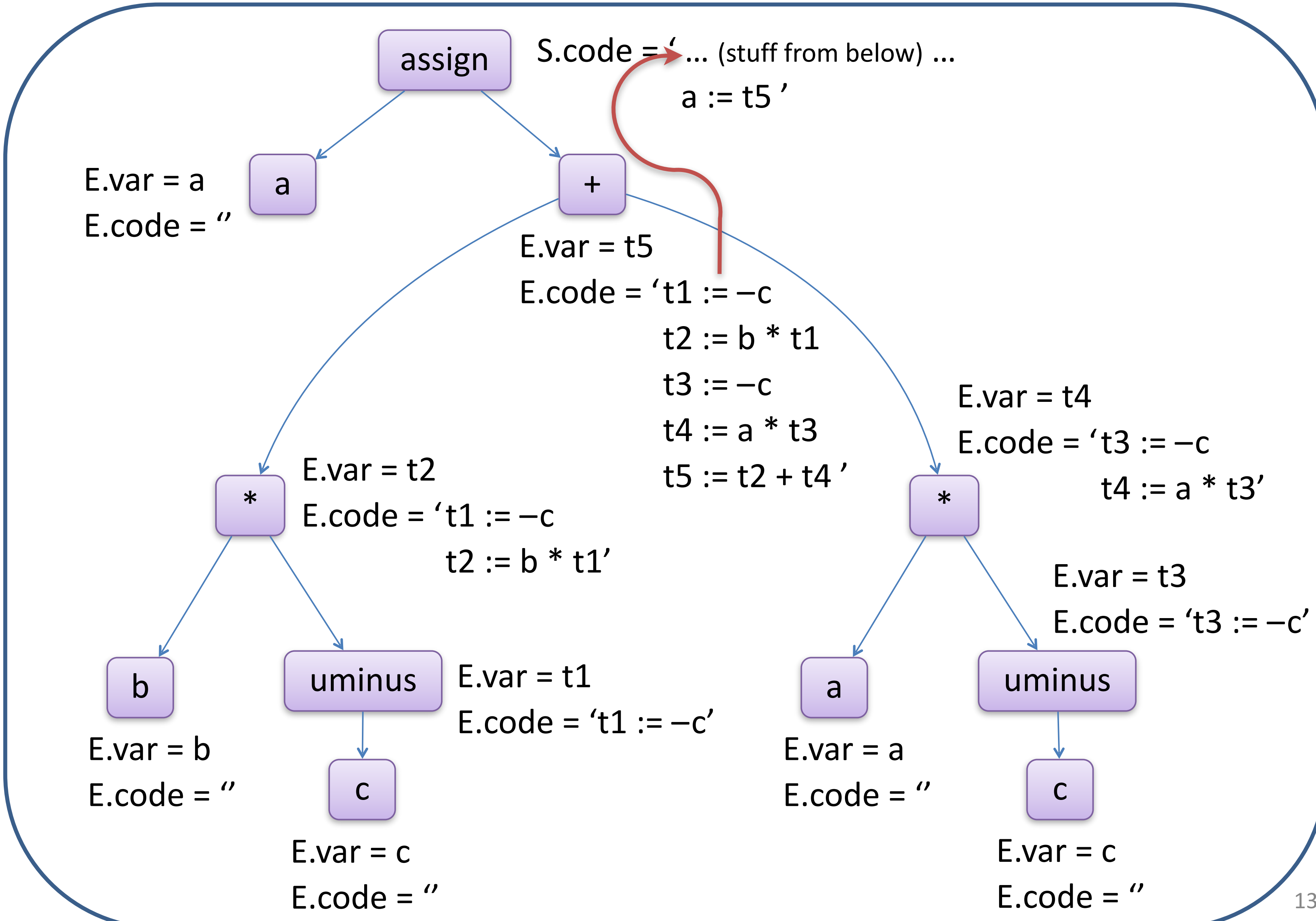
  - freshVar – helper function that returns (the name of) a fresh variable

Creating 3AC: Expressions

production	semantic rule
$S \longrightarrow \mathrm{id}$ **:=** E	S.code = E.code || (id.name ‘:=’ E.var)
$E \longrightarrow E_1 + E_2$	E.var = freshVar(); E.code = E_1.code || E_2.code || (E.var ‘:=’ E_1.var ‘+’ E_2.var)
$E \longrightarrow E_1 * E_2$	E.var = freshVar(); E.code = E_1.code || E_2.code || (E.var ‘:=’ E_1.var ‘*’ E_2.var)
$E \longrightarrow - E_1$	E.var = freshVar(); E.code = E1.code || (E.var ‘:=’ ‘–’ E_1.var)
$E \longrightarrow (E_1)$	E.var = E_1.var; E.code = E_1.code
$E \longrightarrow \mathrm{id}$	E.var = id.name; E.code = ‘’

(we use || to denote concatenation of intermediate code fragments)

Example



assign
S.code = ‘ … (stuff from below) …
a := t5 ’
a
E.var = a
E.code = ‘’
+
E.var = t5
E.code = ‘t1 := –c
t2 := b * t1
t3 := –c
t4 := a * t3
t5 := t2 + t4 ’
*
E.var = t2
E.code = ‘t1 := –c
t2 := b * t1’
*
E.var = t4
E.code = ‘t3 := –c
t4 := a * t3’
b
E.var = b
E.code = ‘’
uminus
E.var = t1
E.code = ‘t1 := –c’
c
E.var = c
E.code = ‘’
a
E.var = a
E.code = ‘’
uminus
E.var = t3
E.code = ‘t3 := –c’
c
E.var = c
E.code = ‘’

Creating 3AC

- Option 1
  accumulate code in AST attributes
  (which is what we just did)

- Option 2 (incremental translation)
  emit IR code to a file during compilation
  - Possible when for every production, the code of the left-hand side is constructed from a concatenation of the code of the right-hand side **in some fixed order**

Expressions and Assignments

This is me cheating 😬

production	semantic action
$S \longrightarrow id := E$	**emit**(id.name ':=' E.var)
$E \longrightarrow E_1 \diamond E_2$	E.var := freshVar(); **emit**(E.var ':=' E_1.var $\diamond$ E_2.var)
$E \longrightarrow - E_1$	E.var := freshVar(); **emit**(E.var ':=' '–' E_1.var)
$E \longrightarrow (E_1)$	E.var := E_1.var
$E \longrightarrow id$	E.var := id.name

$\diamond \in \{'+', '*'\}$

Creating 3AC: Control Statements

- 3AC only supports jumps (conditional and unconditional)
  - Add labels to code
  - Store labels in attributes:

instruction	meaning
goto L	unconditional jump
if x △ y **goto** L	conditional jump

begin	marks beginning of code fragment
next	follows end of code fragment

- freshLabel – helper function that allocates a new, unused label

Creating 3AC: Control statements

For all simple (non-control) statements —

S.begin: **S.code**

S.next:

```
x := x + 2 * y
```

l1: t_1 := 2 * y
x := x + t_1

l2:

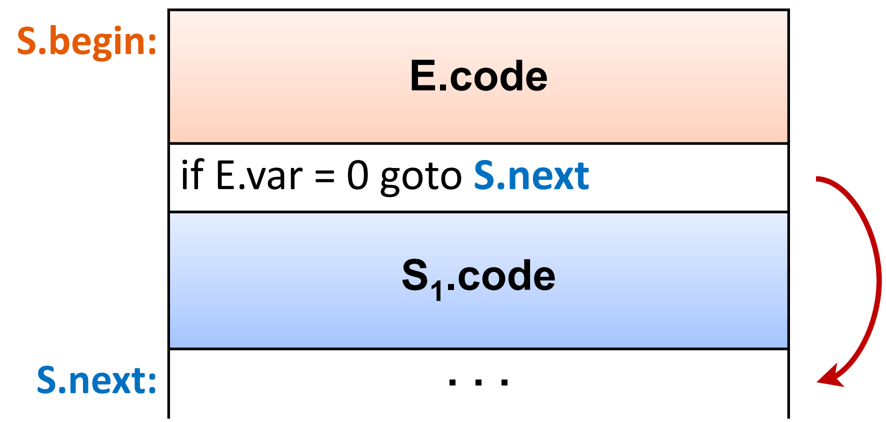
production	semantic action
S → id := E	S.begin = freshLabel(); S.next = freshLabel(); S.code = (S.begin ':') || E.code || id.name ':=' E.var) || (S.next ':')

Creating 3AC: Control statements

S → **if** E **then** S_1

```
if 1 then x := x + 1
```

```
l1:  z := 1
     if z = 0 goto l2
     x := x + 1
l2:
```



production	semantic action
S → **if** E **then** S_1	S.begin = freshLabel(); S.next = freshLabel(); S.code = (S.begin ':') || E.code || ('**if**' E.var '=' '0' '**goto**' S.next) || S_1.code || (S.next ':')

Creating 3AC: Control statements

$S \rightarrow$ `if` E `then` S_1
`else` S_2

```
if 1 then x:=x+1 else x:=x-1
```

l_5:	z := 1
	if z = 0 **goto** l_3
l_1:	x := x + 1
l_2:	goto l_4
l_3:	x := x – 1
l_4: l_6:	

this line is both S.next and S_2.next

S.begin: **E.code**

if E.var = 0 goto **S_2.begin**

S_1.begin: **S_1.code**

goto **S.next**

S_2.begin: **S_2.code**

S.next:

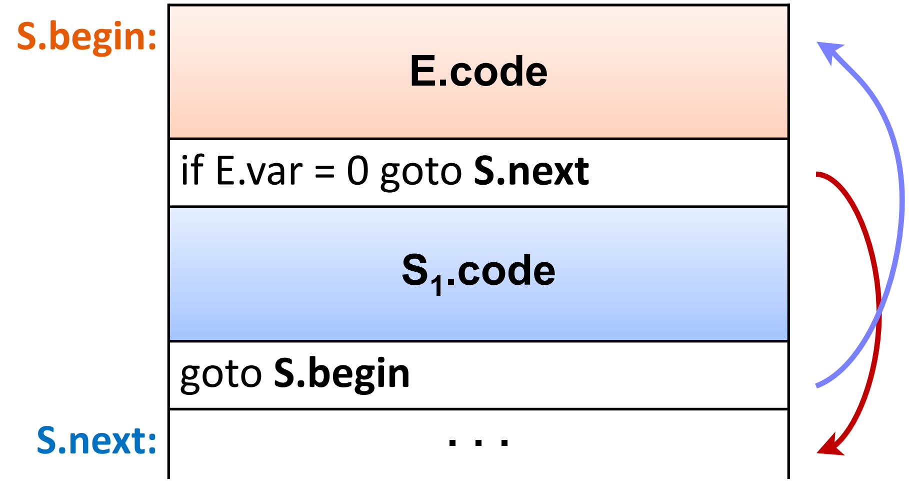
production	semantic action
$S \rightarrow$ `if` E `then` S_1 `else` S_2	S.begin = freshLabel(); S.next = freshLabel(); S.code = (S.begin ':') || E.code || ('**if**' E.var '=' '0' '**goto**' S_2.begin) || S_1.code || ('**goto**' S.next) || S_2.code || (S.next ':')

Creating 3AC: Control statements

$S \rightarrow$ `while` E `do` S_1

`while 1 do x := x + 1`

*l*1: z := 1
if z = 0 **goto** *l*2
x := x + 1
goto *l*1
*l*2:



production	semantic action
$S \rightarrow$ `while` E do S_1	S.begin = freshLabel(); S.next = freshLabel(); S.code = (S.begin ':') || E.code || ('**if**' E.var '=' '0' '**goto**' S.next) || S_1.code || ('**goto**' S.begin) || (S.next ':')

Boolean Expressions

This is me cheating again 😬

production	semantic action
$E \longrightarrow E_1 \circledast E_2$	E.var := freshVar(); E.code := (E.var ':=' E_1.var $\circledast$ E_2.var)
$E \longrightarrow$ `not` E1	E.var := freshVar(); E.code := (E.var ':=' 'not' E_1.var)
$E \longrightarrow (E_1)$	E.var := E_1.var
$E \longrightarrow$ `true`	E.var := freshVar(); E.code := (E.var ':=' '1')
$E \longrightarrow$ `false`	E.var := freshVar(); E.code := (E.var ':=' '0')

$\circledast \in \{$'`and`', '`or`'$\}$

Boolean expressions via jumps

production	semantic action
E ⟶ id_1 ⊜ id_2 😬	E.var := freshVar(); t := freshLabel(); after := freshLabel(); E.code := (‘**if**’ id_1.var ⊜ id_2.var ‘**goto**’ t) || (E.var ‘:=’ ‘0’) || (‘**goto**’ after) || (t ‘:’) || (E.var ‘:=’ ‘1’) || (after ‘:’)

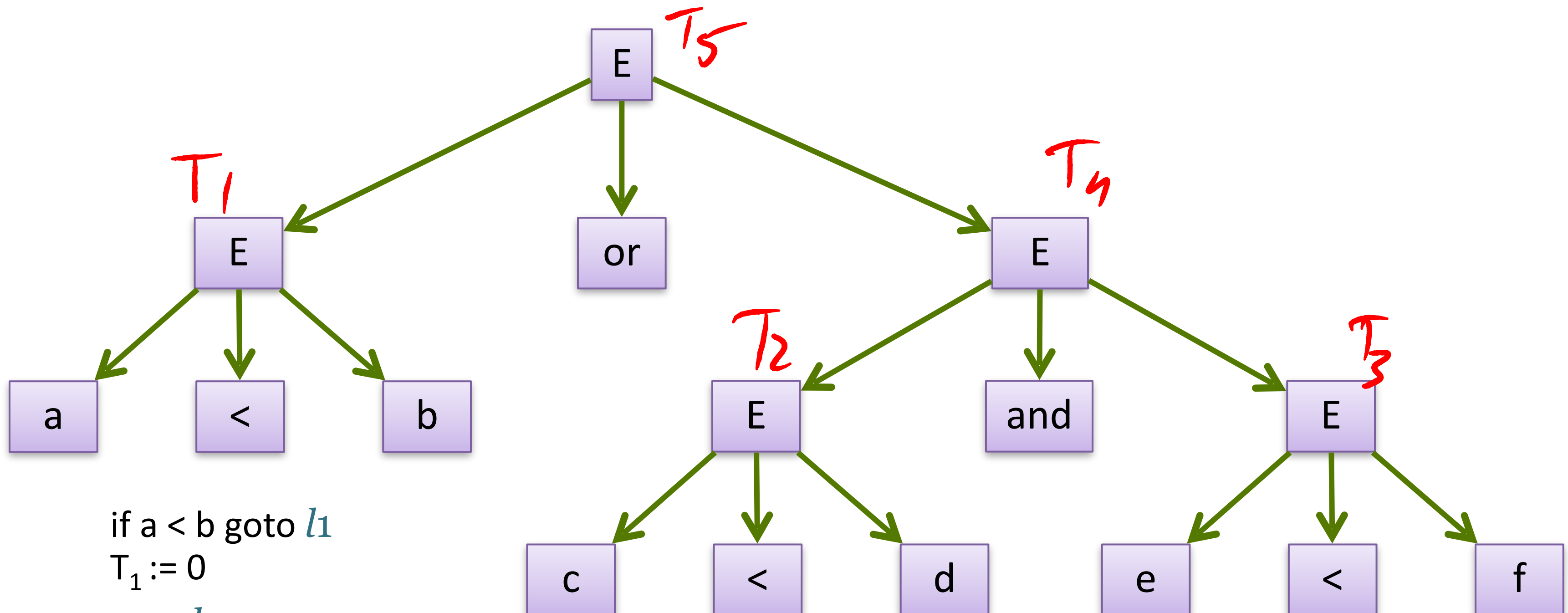
⊜ ∈ {‘<’, ‘<=’, ‘>’, ‘>=’}

`a < b`

	if a < b **goto** *l1*
	t_1 := 0
	goto *l2*
l1:	t_1 := 1
l2:	

Example

`a < b  or  (c < d  and  e < f)`



if a < b goto *l*1
T_1 := 0
goto *l*2
*l*1: T_1 := 1

*l*2: if c < d goto *l*3
T_2 := 0
goto *l*4
*l*3: T_2 := 1

*l*4: if e < f goto *l*5
T_3 := 0
goto *l*6
*l*5: T_3 := 1
*l*6: T_4 := T_2 and T_3
T_5 := T_1 or T_4

Short-Circuit Evaluation

- Second argument of a Boolean operator is only evaluated if the first argument does not already determine the outcome

`(`x `and` y`)` *is equivalent to* `if` x `then` y `else false`

`(`x `or` y`)` *is equivalent to* `if` x `then true else` y

Short-Circuit Evaluation

`a < b or (c < d and e < f)`

100: if a < b goto 103
101: T_1 := 0
102: goto 104
103: T_1 := 1
104: if c < d goto 107
105: T_2 := 0
106: goto 108
107: T_2 := 1
108: if e < f goto 111
109: T_3 := 0
110: goto 112
111: T_3 := 1
112: T_4 := T_2 and T_3
113: T_5 := T_1 or T_4

naïve evaluation

100: if a < b goto 106
101: if c < d goto 103
102: goto 104
103: if e < f goto 106
104: T := 0
105: goto 107
106: T := 1
107:

short-circuit evaluation

Boolean Expressions & Control Structures

$$
\begin{aligned}
S \rightarrow\ & \texttt{if } B \texttt{ then } S_1 \\
\mid\ & \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \\
\mid\ & \texttt{while } B \texttt{ do } S_1
\end{aligned}
$$

- For every Boolean expression B, we attach two attributes

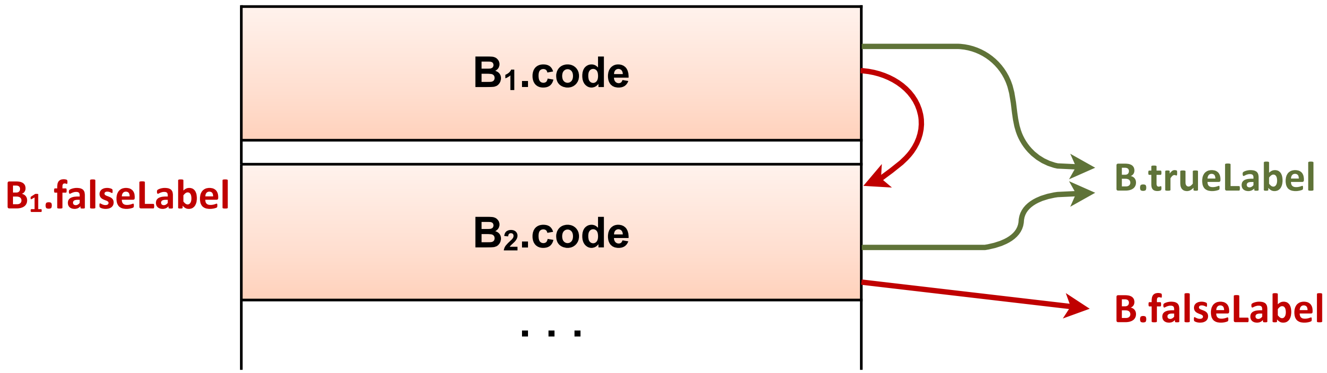
B.falseLabel	target label for a jump when condition B evaluates to **false**
B.trueLabel	target label for a jump when condition B evaluates to **true**

- For every statement S we attach an attribute

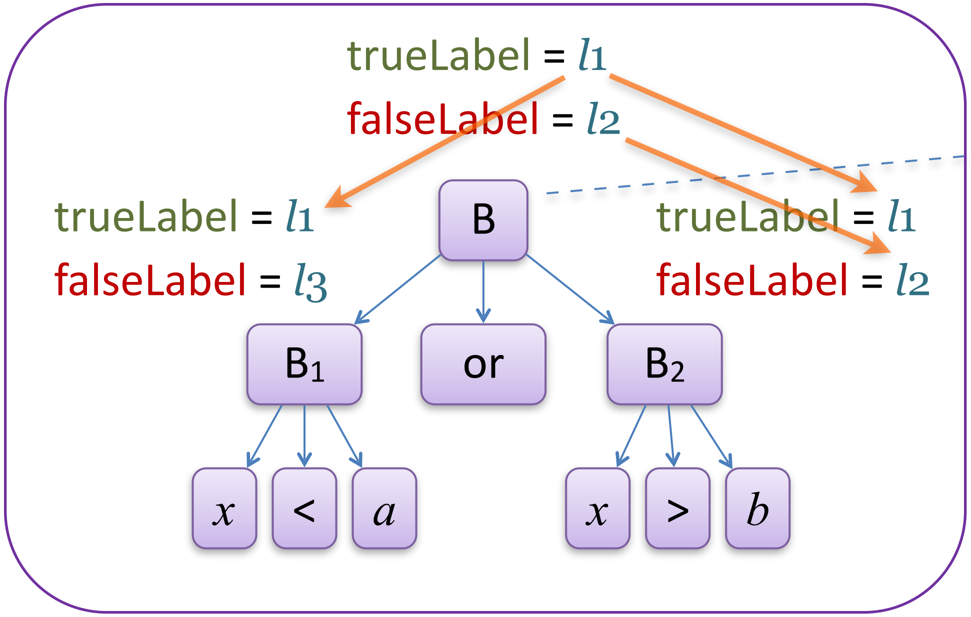
S.next	the label of the **next** code to execute after S

Boolean Expressions

production	semantic action
$B \longrightarrow \texttt{true}$	B.code = ‘**goto**’ B.trueLabel
$B \longrightarrow \texttt{false}$	B.code = ‘**goto**’ B.falseLabel
$B \longrightarrow id_1 \circleddash id_2$	B.code = (‘**if**’ id_1.var $\circleddash$ id_2.var ‘**goto**’ B.trueLabel) || (‘**goto**’ B.falseLabel);
$B \longrightarrow B_1 \texttt{ or } B_2$	B_1.trueLabel = B.trueLabel; B_1.falseLabel = freshLabel(); B_2.trueLabel = B.trueLabel; B_2.falseLabel = B.falseLabel; B.code = B_1.code || (B_1.falseLabel ‘:’) || B_2.code



Boolean Expressions

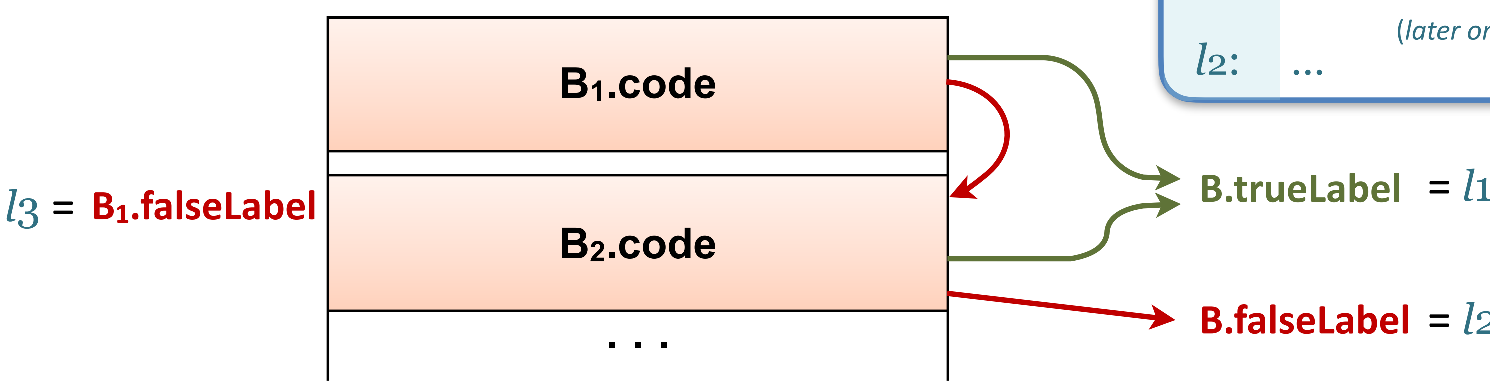


B_1.trueLabel = B.trueLabel;
B_1.falseLabel = freshLabel();
B_2.trueLabel = B.trueLabel;
B_2.falseLabel = B.falseLabel;
B.code = B_1.code || (B_1.falseLabel ‘:’) || B_2.code

`x < a or x > b`

```
     if x < a goto l1
     goto l3
l3:  if x > b goto l1      B.code
     goto l2

l1:  ...
                 (later on)
l2:  ...
```



Boolean Expressions

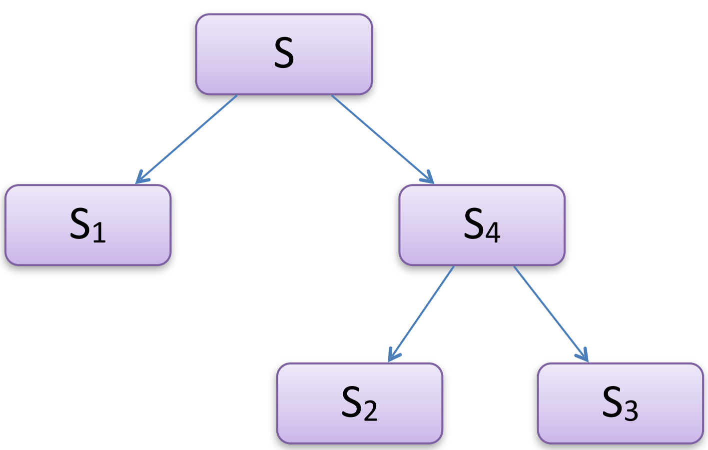
production	semantic action
$B \longrightarrow B_1$ `or` B_2	B_1.trueLabel = B.trueLabel; B_1.falseLabel = freshLabel(); B_2.trueLabel = B.trueLabel; B_2.falseLabel = B.falseLabel; B.code = B_1.code || (B_1.falseLabel ':') || B_2.code
$B \longrightarrow B_1$ `and` B_2	B_1.trueLabel = freshLabel(); B_1.falseLabel = B.falseLabel; B_2.trueLabel = B.trueLabel; B_2.falseLabel = B.falseLabel; B.code = B_1.code || (B_1.trueLabel ':') || B_2.code
$B \longrightarrow$ `not` B_1	B_1.trueLabel = B.falseLabel; B_1.falseLabel = B.trueLabel; B.code = B_1.code;
$B \longrightarrow (B_1)$	B_1.trueLabel = B.trueLabel; B_1.falseLabel = B.falseLabel; B.code = B_1.code;
$B \longrightarrow id_1 \circleddash id_2$	B.code = ('**if**' id_1.var $\circleddash$ id_2.var '**goto**' B.trueLabel) || ('**goto**' B.falseLabel);
$B \longrightarrow$ `true`	B.code = '**goto**' B.trueLabel
$B \longrightarrow$ `false`	B.code = '**goto**' B.falseLabel

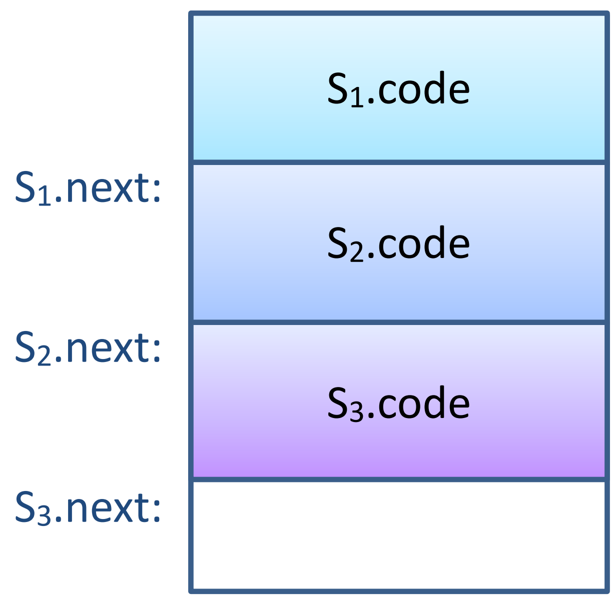
Control Structures: next

Note. We are **changing** the behavior of ‘next’ from before; ‘begin’ is no longer needed.

production	semantic action
$P \longrightarrow S$	S.next = freshLabel(); P.code = S.code || (S.next ‘:’)
$S \longrightarrow S_1 ; S_2$	S_1.next = freshLabel(); S_2.next = S.next; S.code = S_1.code || (S_1.next ‘:’) || S_2.code

$S_1 ; S_2 ; S_3$



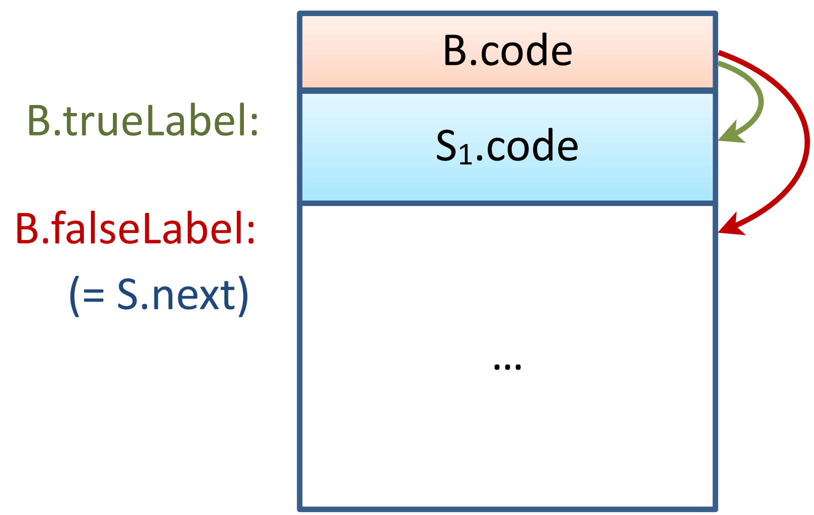


Control Structures: conditional

production	semantic action
$S \rightarrow$ if B then S_1	B.trueLabel = freshLabel(); B.falseLabel = S.next; S_1.next = S.next; S.code = B.code || (B.trueLabel ‘:’) || S_1.code

```
if a<b then x:=x+1
```

```
      if a < b goto l1
      goto l2
l1:   x := x + 1
l2:
```

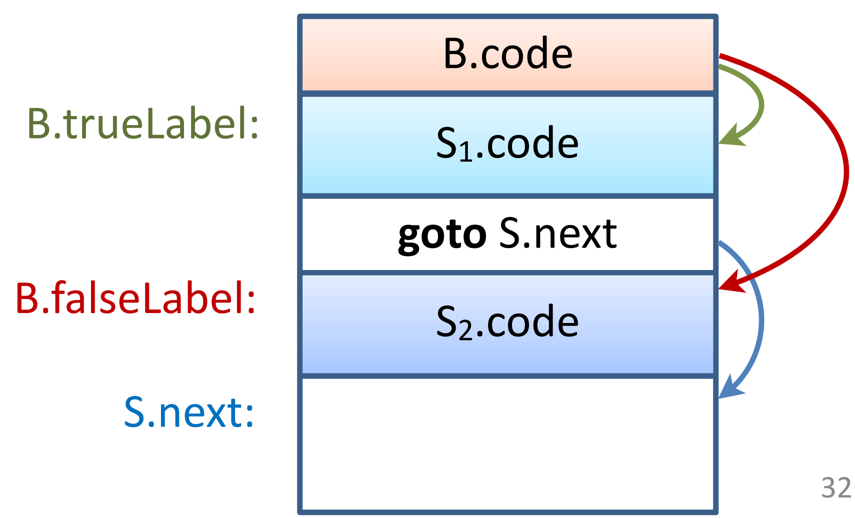


Control Structures: conditional

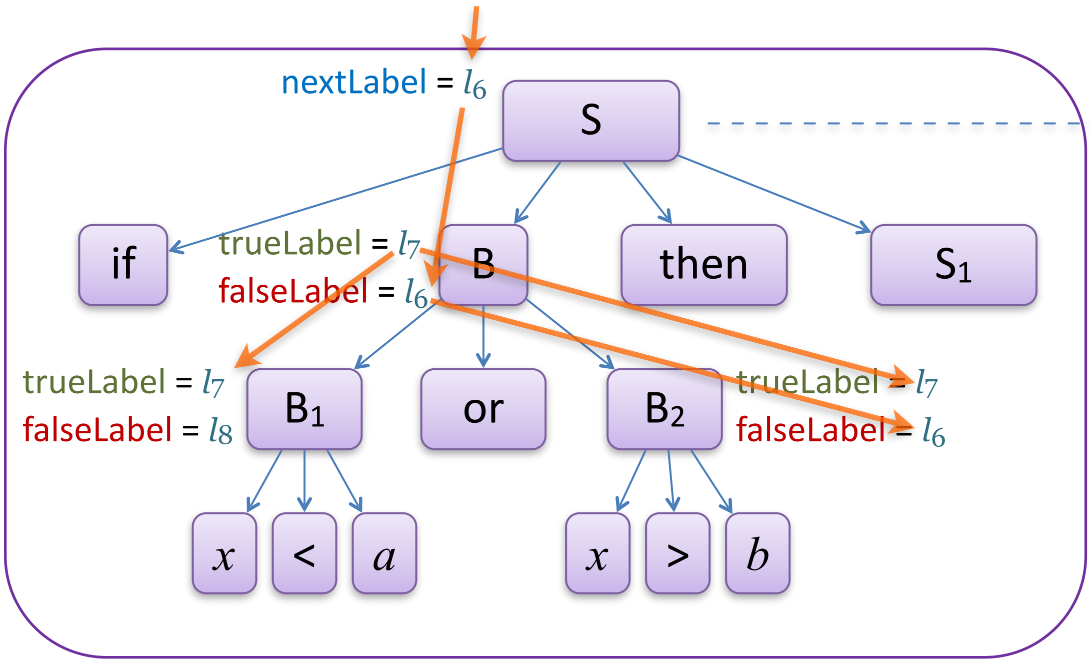
production	semantic action
$S \longrightarrow$ if B then S_1 else S_2	B.trueLabel = freshLabel(); B.falseLabel = freshLabel(); S_1.next = S.next; S_2.next = S.next; S.code = B.code || (B.trueLabel ‘:’) || S_1.code || (‘**goto**’ S.next) || (B.falseLabel ‘:’) || S_2.code

```
if a<b then x:=x+1 else x:=x-1
```

	if a < b **goto** l_1
	goto l_2
l_1:	x := x + 1
	goto l_3
l_2:	x := x – 1
l_3:	



Boolean Expressions



B.trueLabel = freshLabel();

B.falseLabel = S.next;

S_1.next = S.next;

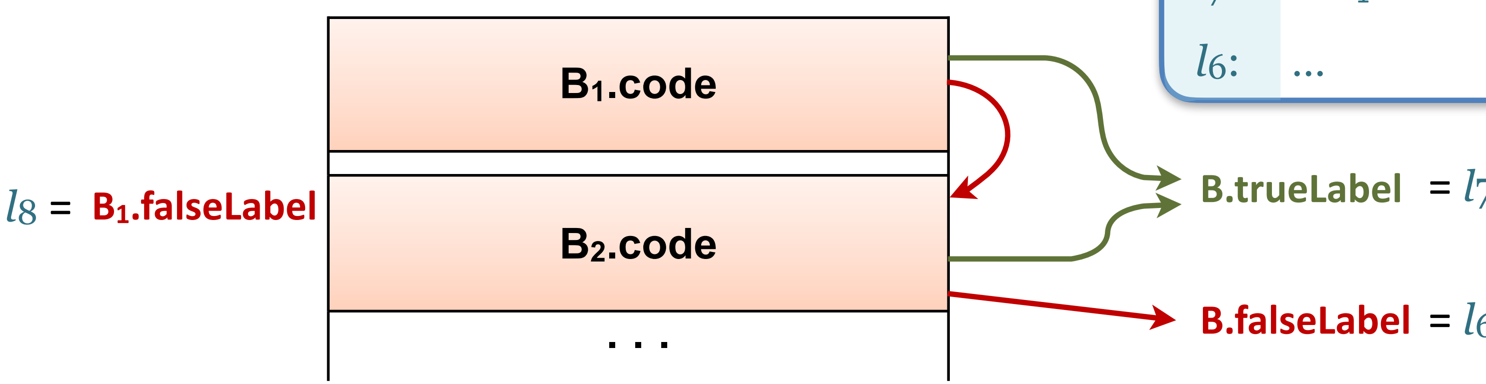
S.code = B.code ||

(B.trueLabel ':') || S_1.code

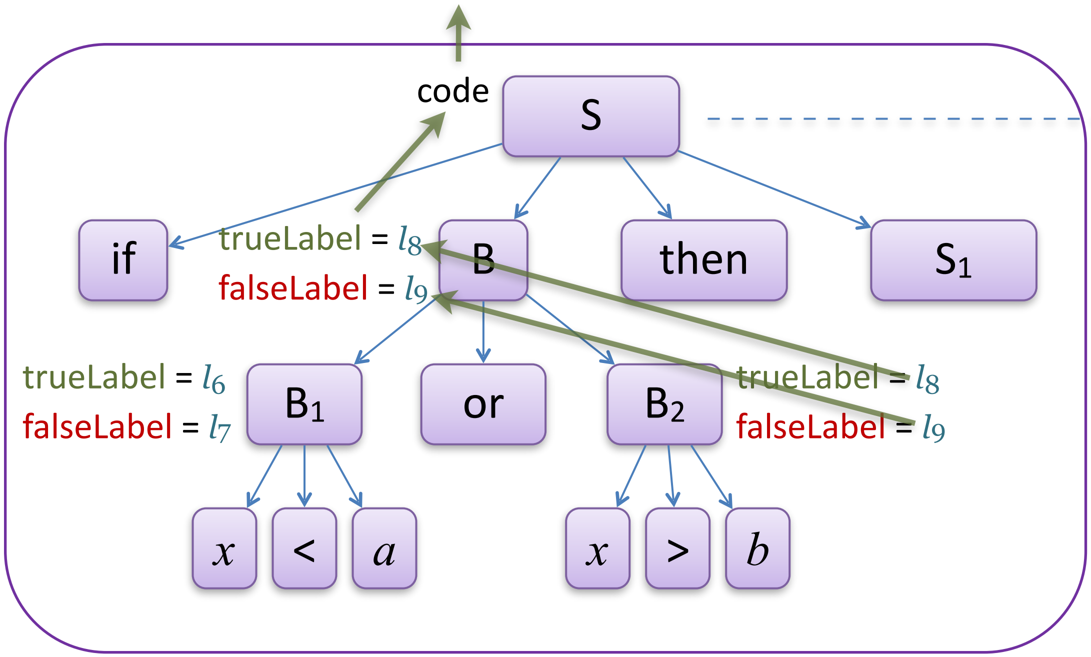
```
if   x < a   or   x > b   then  S1
```

	if x < a **goto** l_7
	goto l_8
l_8:	if x > b **goto** l_7
	goto l_6
l_7:	... S_1.code ...
l_6:	...

B.code



Boolean Expressions – S-Attributed



S.code = B.code
|| (B.trueLabel ‘:’) || S_1.code
|| (B.falseLabel ‘:’)

if x < a **or** x > b then S_1

	if x < a **goto** l_6	B.code
	goto l_7	
l_7:	**if** x > b **goto** l_8	
	goto l_9	
l_6:	**goto** l_8	
l_8:	… S_1.code …	
l_9:	…	

B ⟶ id_1 ⊜ id_2	B.trueLabel = freshLabel(); B.falseLabel = freshLabel(); B.code = (‘**if**’ id_1.var ⊜ id_2.var ‘**goto**’ B.trueLabel) || (‘**goto**’ B.falseLabel)
B ⟶ B_1 or B_2	B.trueLabel = B_2.trueLabel; B.falseLabel = B_2.falseLabel; B.code = B_1.code || (B_1.falseLabel ‘:’) || B_2.code || (B_1.trueLabel ‘:’) || (‘**goto**’ B.trueLabel)

Boolean Expressions – S-Attributed

production	semantic action
$B \longrightarrow$ `true`	B.trueLabel = freshLabel(); B.falseLabel = freshLabel(); B.code = ‘**goto**’ B.trueLabel
$B \longrightarrow$ `false`	B.trueLabel = freshLabel(); B.falseLabel = freshLabel(); B.code = ‘**goto**’ B.falseLabel
$B \longrightarrow id_1 \circleddash id_2$	B.trueLabel = freshLabel(); B.falseLabel = freshLabel(); B.code = (‘**if**’ id_1.var $\circleddash$ id_2.var ‘**goto**’ B.trueLabel) || (‘**goto**’ B.falseLabel)
$B \longrightarrow B_1$ `or` B_2	B.trueLabel = B_2.trueLabel; B.falseLabel = B_2.falseLabel; B.code = B_1.code || (B_1.falseLabel ‘:’) || B_2.code || (B_1.trueLabel ‘:’) || (‘**goto**’ B.trueLabel);
$S \longrightarrow$ if B then S_1	S.code = B.code || (B.trueLabel ‘:’) || S_1.code || (B.falseLabel ‘:’)
$S \longrightarrow$ if B then S_1 else S_2	S.next = freshLabel(); S.code = B.code || (B.trueLabel ‘:’) || S_1.code || (‘**goto**’ S.next) (B.falseLabel ‘:’) || S_2.code || (S.next ‘:’)

Boolean Expressions – S-Attributed

with **emit**(..)

production	semantic action
$B \longrightarrow id_1 \ominus id_2$	B.trueLabel = freshLabel(); B.falseLabel = freshLabel(); B.code = (‘**if**’ id_1.var ⊜ id_2.var ‘**goto**’ B.trueLabel) || (‘**goto**’ B.falseLabel)
$S \longrightarrow$ if B then S_1	S.code = B.code || (B.trueLabel ‘:’) || S_1.code || (B.falseLabel ‘:’)

production	semantic action
$B \longrightarrow id_1 \ominus id_2$	B.trueLabel = freshLabel(); B.falseLabel = freshLabel(); **emit**(‘**if**’ id_1.var ⊜ id_2.var ‘**goto**’ B.trueLabel || ‘**goto**’ B.falseLabel)
$S \longrightarrow$ if B M then S_1	**emit**(B.falseLabel ‘:’)
$M \longrightarrow \varepsilon$	**emit**(B.trueLabel ‘:’)

LLVM IR

- LLVM = Low-Level Virtual Machine
  - A well-known misnomer: it is (mostly) a **compiler** framework
- Flat IR, similar to 3-address code in nature
- All values are ***typed*** (unlike assembly)

```
%num = add i32 %inp, 48
```

target opcode type operands

https://llvm.org/docs/LangRef.html

LLVM IR — Hello World

global symbol

```
@str = internal constant [14 x i8] c"hello, world\0A\00"
```

external symbol

```
declare i32 @printf(i8*, ...)
```

start function

start block

type annotations

```
define i32 @main() {
entry:
    %tmp1 = getelementptr [14 x i8], [14 x i8]* @str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...) @printf( i8* %tmp1 )

    ret i32 42
}
```

local symbol

end block

end function

LLVM IR — Types

• Numeric	`i1`
	`i32`
‣ Integers — any bit width!	`i1942652`
‣ Floating point: 16, 32, 64 bit	`half, float, double`
• Pointer	`i8 *`
• Label (= code address)	`label`
• Aggregate	
	`[40 x i32]`
‣ Array (fixed dimensions)	`[12 x [10 x float]]`
‣ Struct	`{ float, [ 4 x i32 ] }`

https://llvm.org/docs/LangRef.html

LLVM IR — Memory

```
%a = alloca i32
store i32 5, i32* %a
%rd = load i32, i32* %a
```

allocate on stack
return type is a pointer

write to memory address   *a = 5

read from memory address   rd = *a

aggregate type

```
%a = alloca [4 x i32]
%el = getelementptr [4 x i32],
          [4 x i32]* %a, i32 0, i32 1
store i32 5, i32* %el
%rd = load i32, i32* %el
```

base pointer idx_1 idx_2

compute address of element in aggregate
return type is a pointer

LLVM IR — Memory

getelementptr

All Maps Videos News Images More Settings Tools

About 33,900 results (0.30 seconds)

The Often Misunderstood GEP Instruction — LLVM 10 ...

https://llvm.org › docs › GetElementPtr

This document seeks to dispel the mystery and confusion surrounding LLVM's **GetElementPtr** (GEP) instruction. Questions about the wily GEP instruction are ...

LLVM — Gotchas

- LLVM blocks must start with a label and end with a *Terminator instruction*

`ret` `br` `switch` `indirectbr` `invoke` `callbr` `resume`
`catchswitch` `catchret` `cleanupret` `unreachable`

- LLVM variables are *Single Static Assignment* (SSA)
  - Wait, what?

Single Static Assignment

- Variables can only appear on the left-hand side of ***one*** assignment.

```
cond:
   %b = icmp slt i32 %i, %j
   br i1 %b, label %then,
             label %else
then:
   %max = or i32 0, %j
   br label %exit

else:
   %max = or i32 0, %i
   br label %exit

exit:
   ret i32 %max
```

✗

```
cond:
   %b = icmp slt i32 %i, %j
   br i1 %b, label %then,
             label %else
then:
   %max1 = or i32 0, %j
   br label %exit

else:
   %max2 = or i32 0, %i
   br label %exit

exit:                          %j
   %max = phi i32 [ %max1, %then ],
                  [ %max2, %else ]
   ret i32 %max                %i
```

✓

Coming Up