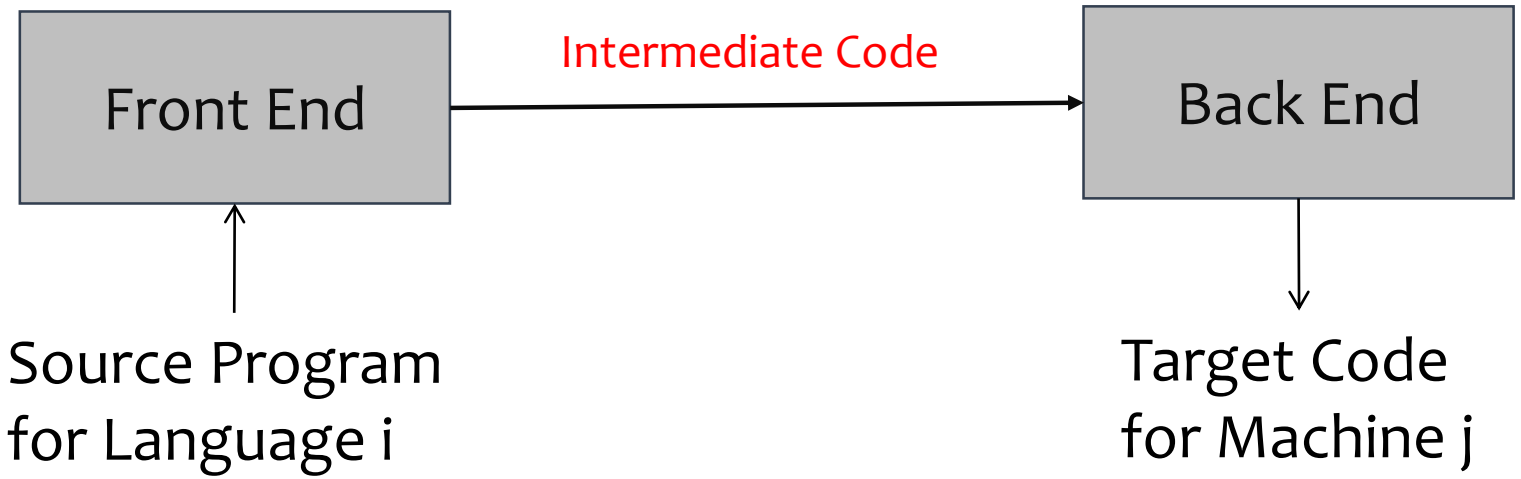
Chapter 6

Intermediate Code Generation

Intermediate Representation



Intermediate Representation

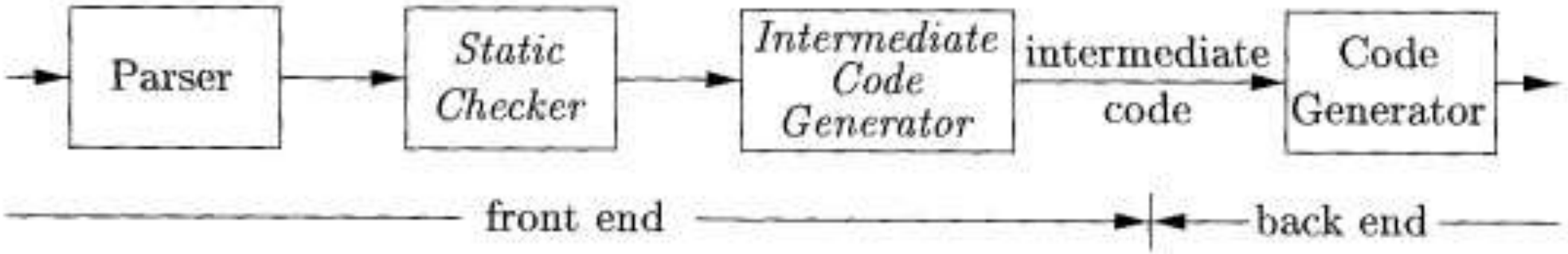


Figure 6.1: Logical structure of a compiler front end

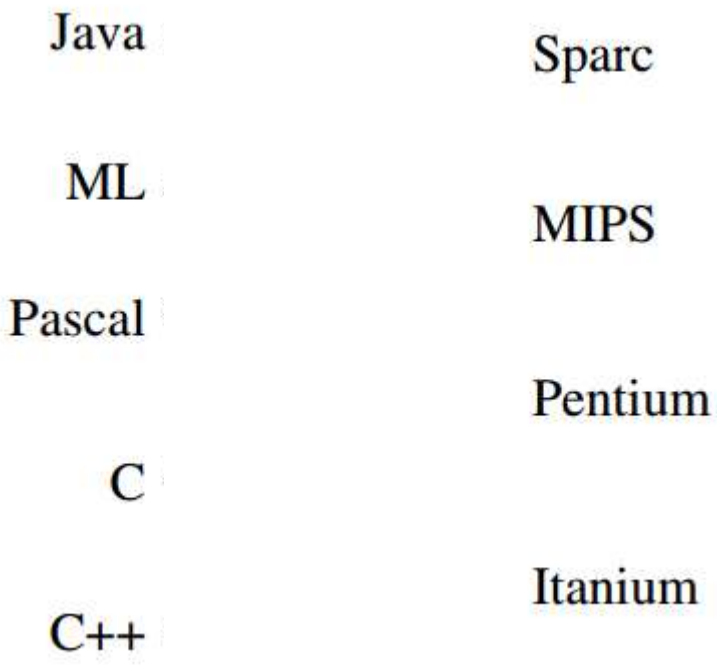
Intermediate Representation

- What is an Intermediate Representation (IR)?
  - ➢ Abstract Machine Language
  - ➢ Machine and Language independent version of source program
  - ➢ Can express target machine operation without committing too much machine specific details

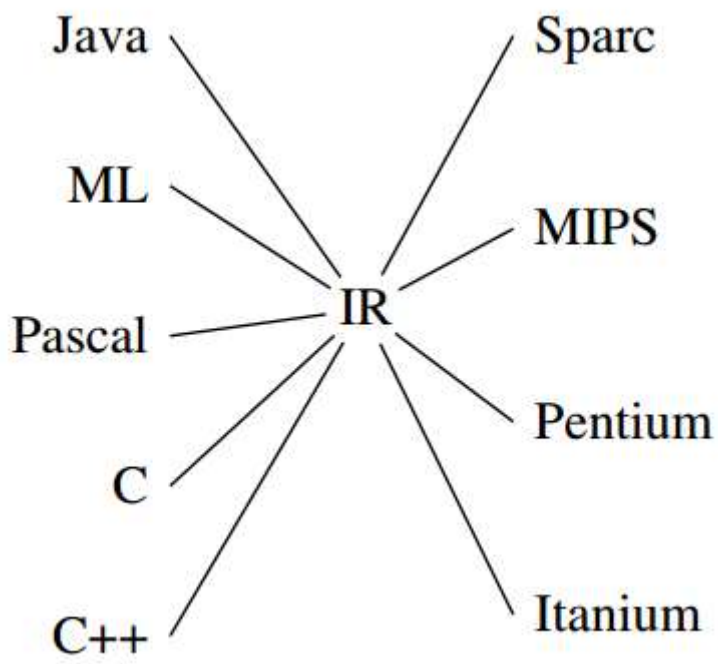
Why Need IR?

- Modularity:
  - Front end is not complicated with machine specific details
  - Back end is not bothered with information specific to the source code
- Portability & Cross-Compilation:
  - For n source language and m machines, need only n front ends and m back ends
  - Without IR, a total of $n \times m$ compilers are required
- Simplify Certain Optimization:
  - Easier and clearer optimization tasks

Why Need IR?



Without an IR

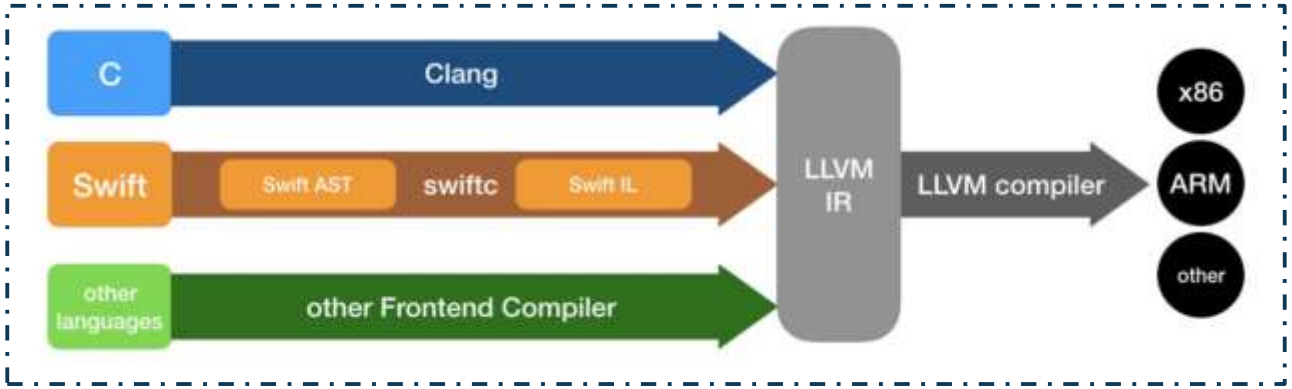


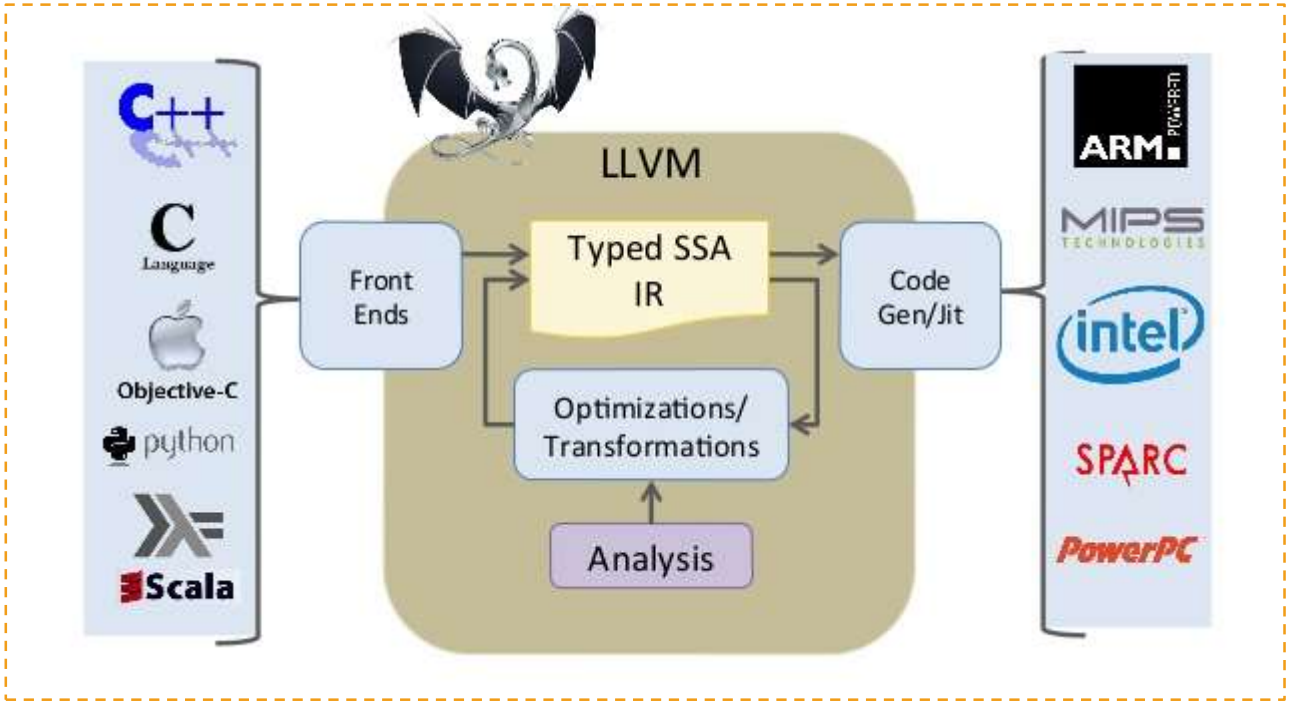
With an IR

Figure Source: Modern Compiler Implementation in Java by Andrew Appel

The LLVM Project

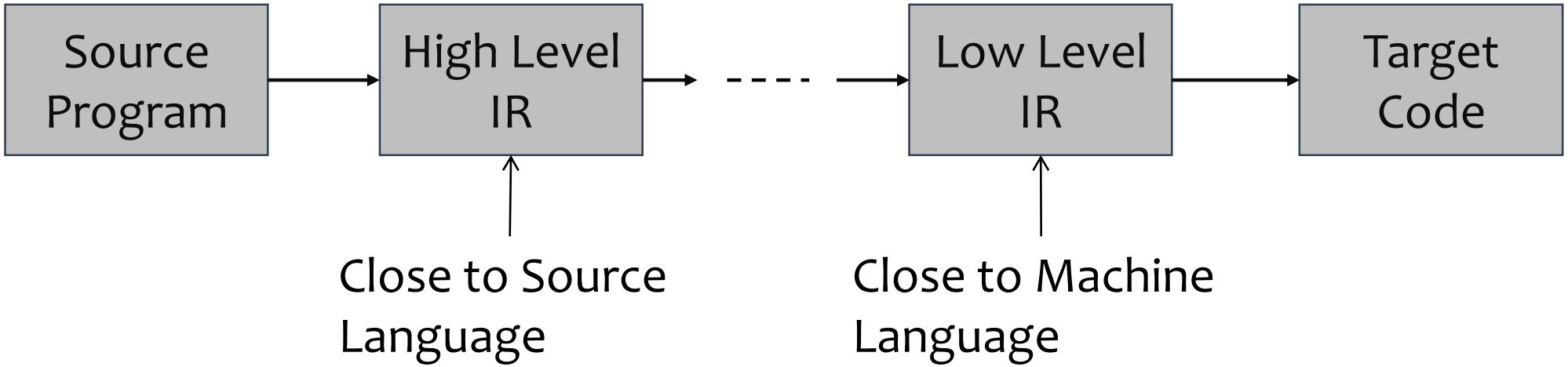
https://www.llvm.org/





Designing Good IR?

- Balance between high-level source language and low level machine language
- Often there are multiple IRs in a single compiler



Choice of an IR

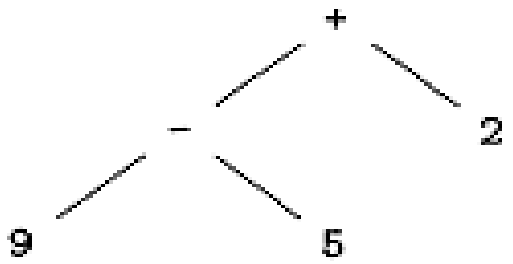
- The choice or design of an intermediate representation varies from compiler to compiler.
- An intermediate representation may either be
  - an actual language or
  - it may consist of internal data structures that are shared by phases of the compiler.
- C is a programming language, yet it is often used as an intermediate form because
  - it is flexible,
  - compiles into efficient machine code, and
  - its compilers are widely available.
- The original C + + compiler consisted of a front end that generated C, treating a C compiler as a back end.
- Another example: python compiler using C++ as an IR (https://github.com/Omyyyy/pycom)

High and Low Level IRs

- High Level IRs:
  - Syntax Tree
  - DAG
  - Three Address Code with high level operators
- Low Level IRs:
  - Three Address Code with low level operators

Syntax Trees

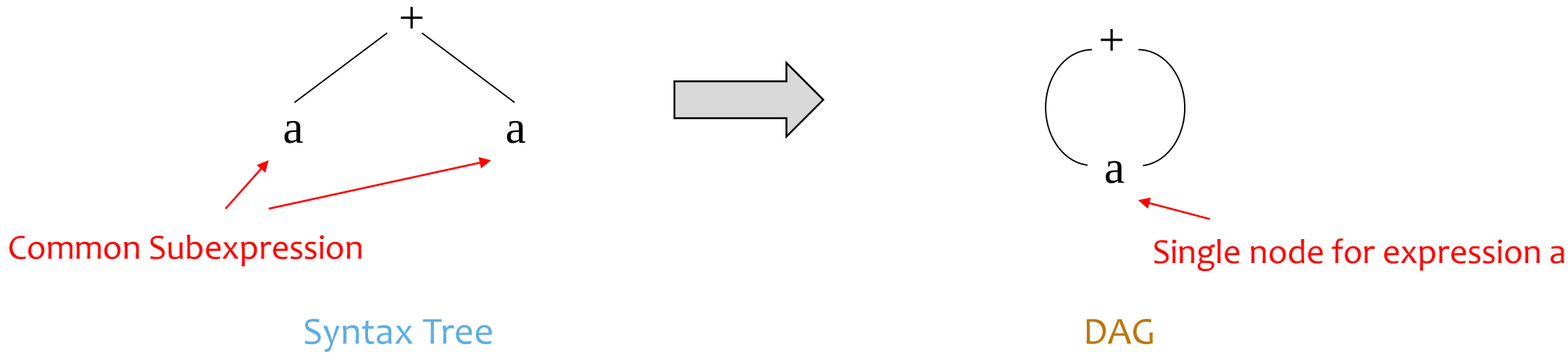
- Syntax Tree:
  - Each node represents a construct of source program
  - Each children represent a meaningful component
  - Leaves correspond to atomic operands
  - Interior nodes correspond to operators



Syntax Tree for expression 9-5+2

Directed Acyclic Graphs (DAG)

- Detect Common Subexpression
  - Produce only one subtree for each subexpression
  - Consider an expression: $a + a$

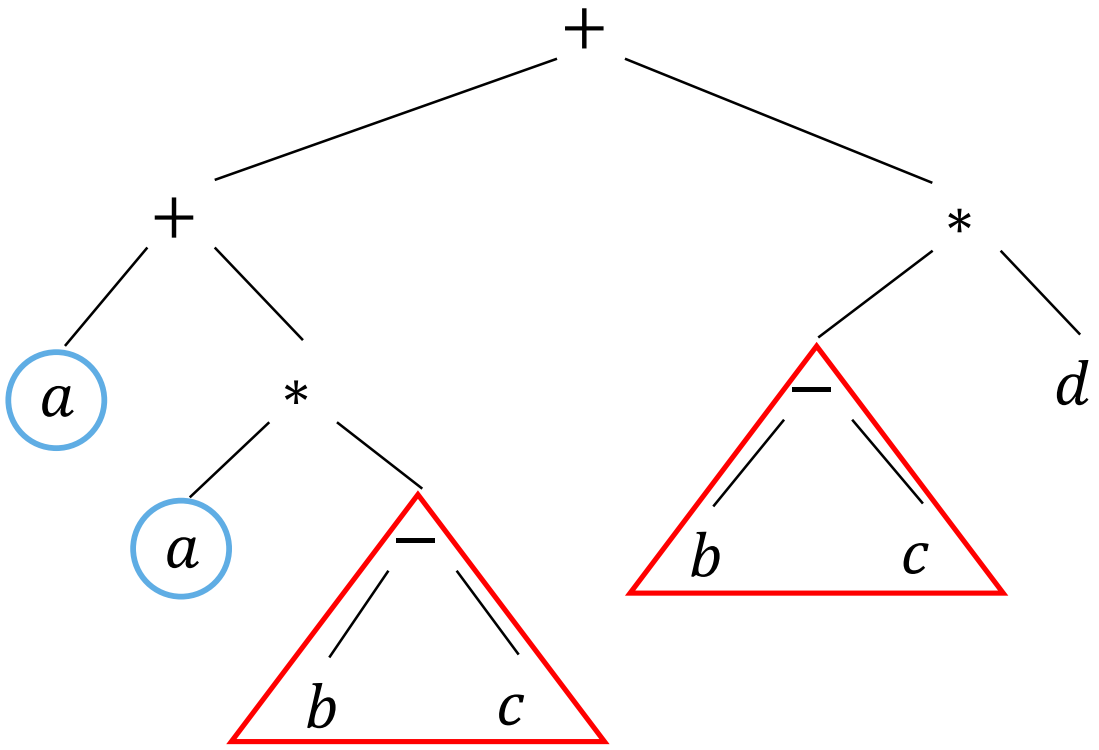


Directed Acyclic Graphs (DAG)

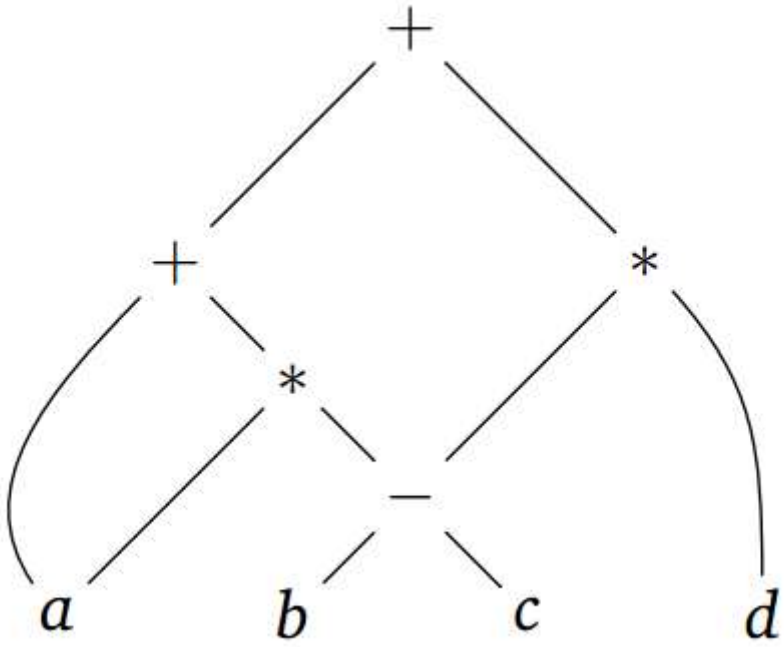
- A Node N can have more than one parent
  - If it represent a common subexpression
- Succinct Representation of Syntax Tree
- Provide clues for generating efficient code

Directed Acyclic Graphs (DAG)

- An example: $a + a * (b - c) + (b - c) * d$



Syntax Tree



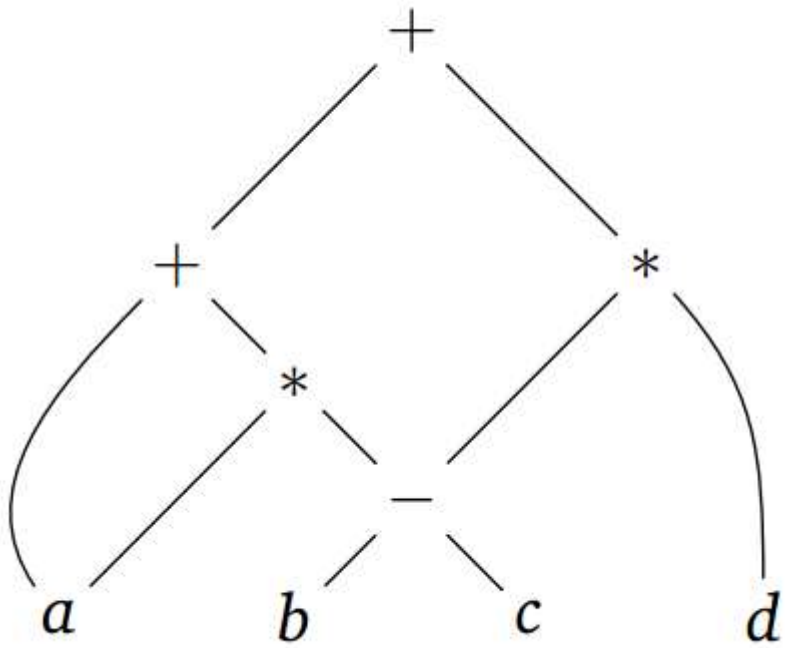
DAG

SDD for Constructing DAG

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	
2)	$E \rightarrow E_1 - T$	
3)	$E \rightarrow T$	
4)	$T \rightarrow (E)$	
5)	$T \rightarrow \mathbf{id}$	
6)	$T \rightarrow \mathbf{num}$	

- Wait!! What?? This SDD constructs Syntax Tree!
- The Leaf and Node functions will return a previously created identical node if exist, otherwise return a newly created one
- Identical Node means the subtree rooted at that node is same

SDD for constructing DAG

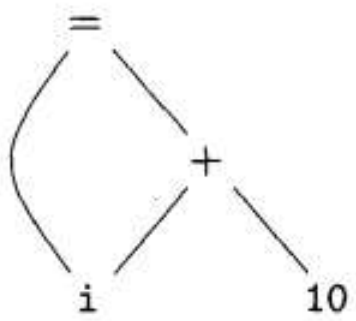


DAG for expression: $a + a * (b - c) + (b - c) * d$

1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Steps of constructing the left DAG

Value Number Method for Constructing DAG



(a) DAG

1	id		to entry for i
2	num	10	
3	+	1	2
4	=	1	3
5	...		

(b) Array.

Second Field of leaf node is lexical value

Second and Third Field of interior node contains index number of left and right child

First Field is an **operation code** (label of the node)

- Nodes of AST or DAG can be stored in array or records (structures)
- Each row represent one record (one node)
- Reference to a node is the index of the node in the array
  - The integer index is called the value number of the node
- The value number of + node is 3

Value Number Method

- Algorithm for constructing Nodes of DAG (Node and Leaf function)
  - Input: Label op, Node l and Node r (For leaf node $r = 0$)
  - Output: The value number of a node in the array with signature $< op, l, r >$
  - Method:
    - Search array for a node M with label op, left child l and right child r
    - If found return value number of M
    - Otherwise create in the array a new node N with label op, left child l and right child r
    - Return value number of N

Value Number Method

- Searching the entire array every time to check whether a node exists is expensive
- Use Hash Table
- Hash using $< op, l, r >$ as hash key

Three Address Code

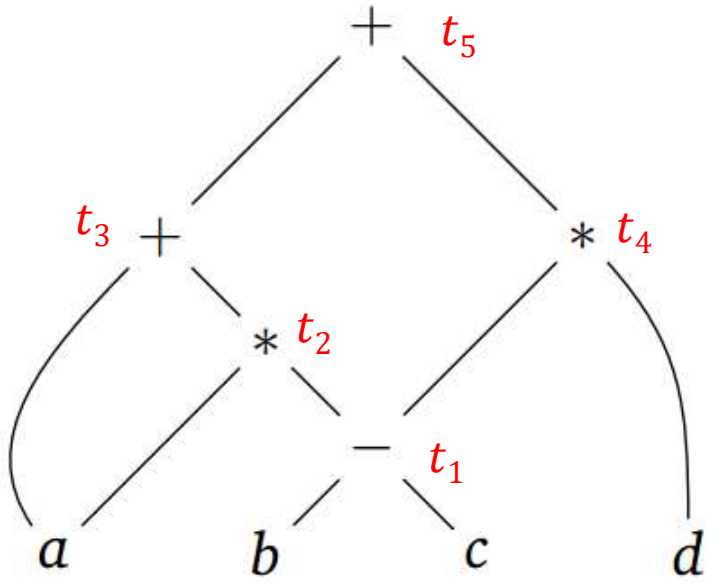
- At most one operator at the right side of an instruction
- For expression $x * y + z$ three address code is

$$t_1 = x * y$$
$$t_2 = z + t_1$$

- t_1 and t_2 are compiler generated names

Three Address Code

- Linearized representation for DAG
  - ➢Explicit name corresponds to the interior nodes
  - ➢Each instruction in the sequence has a label



DAG for expression: $a + a * (b - c) + (b - c) * d$

$$t_1 = b - c$$
$$t_2 = a * t_1$$
$$t_3 = a + t_2$$
$$t_4 = t_1 * d$$
$$t_5 = t_3 + t_4$$

Three Address Code

Three Address Code

- Based on Two concepts:
  - Address
    - What address can be used in the code
  - Instruction
    - What kind of instruction sets can be applied

Address

- Name
  - Name from source program
  - Example: x, y, z
  - May be replaced by pointer to symbol table entry in implementation
- Constant
  - Example: $2.3, 1$
- Compiler generated temporary
  - Example: t_1, t_2

Instructions

- Assignment instructions
  - ***x = y op z*** where x, y, z are addresses and op is a binary operator
  - ***x = op y*** where op is a unary operation such as unary minus, logical negation and conversion operator
- Copy instruction: ***x = y***
  - x is assigned the value of y
- An unconditional jump : *goto L*
  - Instruction with label L is executed next
- Conditional jumps:
  - ***if x goto L***
  - ***ifFalse x goto L***
  - ***if x relop y goto L***

Instructions

- Procedure calls and returns:
  - ***param x*** for parameters
  - ***call p, n*** or ***y = call p, n*** for procedure/function calls
  - ***return y*** represent returning a value
  - For procedure $proc(x_1, x_2, \dots, x_n)$

    param x_1

    param x_2

    …

    paramx_n

    call proc,n

Instructions

- Indexed copy instructions:
  - ***x = y[i]*** sets x to the value in the location *i* memory units beyond location *y*.
  - ***x[i] = y*** sets the contents of the location *i* units beyond x to the value of *y*

- Address and Pointers assignments:
  - x = & y
  - x = * y
  - * x = y

Assigning Labels

- Example: do i = i +1; while(a[i] < v);

$$
\begin{aligned}
L: \quad & t_1 = i + 1 \\
& i = t_1 \\
& t_2 = i * 8 \\
& t_3 = a[t_2] \\
& if\ t_3 < v\ goto\ L
\end{aligned}
$$

Symbolic Labels

$$
\begin{aligned}
100: \quad & t_1 = i + 1 \\
101: \quad & i = t_1 \\
102: \quad & t_2 = i * 8 \\
103: \quad & t_3 = a[t_2] \\
104: \quad & if\ t_3 < v\ goto\ 100
\end{aligned}
$$

Position Numbers

Choice of Operators

- The choice of allowable operators is an important issue
  - The operator set clearly must be rich enough to implement the operations in the source language.
  - Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine.
  - However, if the front end must generate long sequences of instructions for some source-language operations, then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations.

Representing Three Address Code

- What is the suitable data structure to represent three address code?
- Can be represented using objects or record with operators and operands as fields
- Three representations:
  - Quadruples
  - Triples
  - Indirect Triples

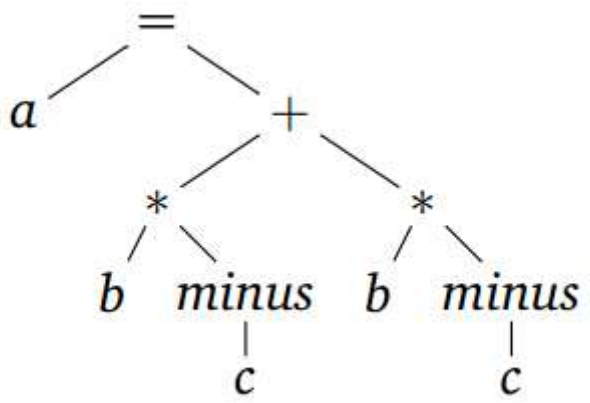
Quadruples

- Four Fields:
  - *op* – represent operator
  - *arg1*, *arg2* – holding operands
  - *result* – hold result of the operation

- Exceptions:
  - Unary Operators: no *arg2*
  - param operator: no *arg2*, no *result*
  - (Un)conditional jump: target label is the *result*

Quadruples

- Example: $a = b * -c + b * -c;$



Syntax Tree

$$t_1 = minus\ c$$
$$t_2 = b * t_1$$
$$t_3 = minus\ c$$
$$t_4 = b * t_3$$
$$t_5 = t_2 + t_4$$
$$a = t_5$$

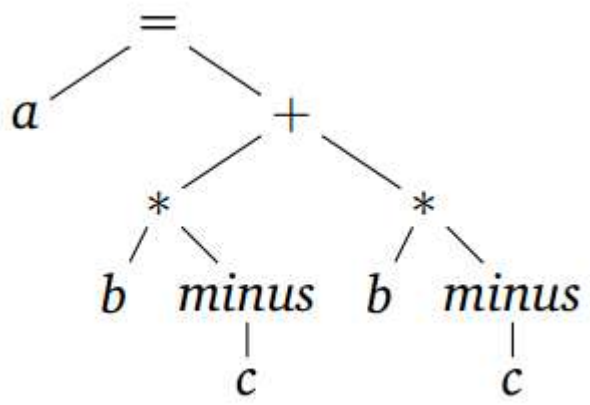
Three Address Code

	OP	ARG1	ARG2	RESULT
0	*minus*	c		t_1
1	$*$	b	t_1	t_2
2	*minus*	c		t_3
3	$*$	b	t_3	t_4
4	$+$	t_2	t_4	t_5
5	$=$	t_5		a

Quadruples

Triples

- Three fields:
  - *op*, *arg1*, *arg2*
  - No *result field*
  - *result* referred by their position
- *Example:* $a = b * -c + b * -c;$



Syntax Tree

$$t_1 = minus\ c$$
$$t_2 = b * t_1$$
$$t_3 = minus\ c$$
$$t_4 = b * t_3$$
$$t_5 = t_2 + t_4$$
$$a = t_5$$

Three Address Code

	OP	ARG1	ARG2
0	*minus*	*c*	
1	*	*b*	(0)
2	*minus*	*c*	
3	*	*b*	(2)
4	+	(1)	(3)
5	=	*a*	(4)

Triples

Triples

- Optimization by moving instructions is difficult in Triples, but easy in Quadruples
- Moving instructions forces to adjust the arguments where that position appear

	OP	ARG1	ARG2
0	*minus*	c	
1	$*$	b	(0)
2	*minus*	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)

Triples

Indirect Triples

- Listing pointers to the triples instead of triples
- Reorder the pointers instead of the tuples itself during code motion

	OP	ARG1	ARG2
0	*minus*	*c*	
1	*	*b*	(0)
2	*minus*	*c*	
3	*	*b*	(2)
4	+	(1)	(3)
5	=	*a*	(4)

Triples

	INSTRUCTION
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

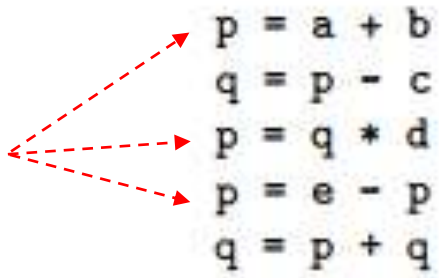
	OP	ARG1	ARG2
0	*minus*	*c*	
1	*	*b*	(0)
2	*minus*	*c*	
3	*	*b*	(2)
4	+	(1)	(3)
5	=	*a*	(4)

Indirect Triples

Static Single Assignment Form (SSA)

- Another form of IR
- All assignments are to variables with distinct names

Assignments to the same variable



(a) Three-address code.

SSA

- Facilitates code optimizations

Redundant but difficult to detect

x = 10
x = 15
y = x

SSA

~~$x_1 = 10$~~
$x_2 = 15$
$y = x_2$

Easy to detect and remove

Static Single Assignment Form (SSA)

- Converting ordinary code into SSA form
  - Replace the target of each assignment with a new variable
  - Replacing each use of a variable with the "version" of the variable reaching that point.
- What to do when branching is involved?

```
if (flag) x = -1
else x = 1
y = x*a
```

SSA

```
if (flag) x1 = -1
else x2 = 1
y = x?*a
```

- Φ functions to combine two definitions of the same variable x

```
if (flag) x = -1; else x = 1
y = x*a
```

SSA

```
if (flag) x1 = -1; else x2 = 1
x3 = φ(x1, x2)
y = x3*a
```

For further details see: https://en.wikipedia.org/wiki/Static_single-assignment_form

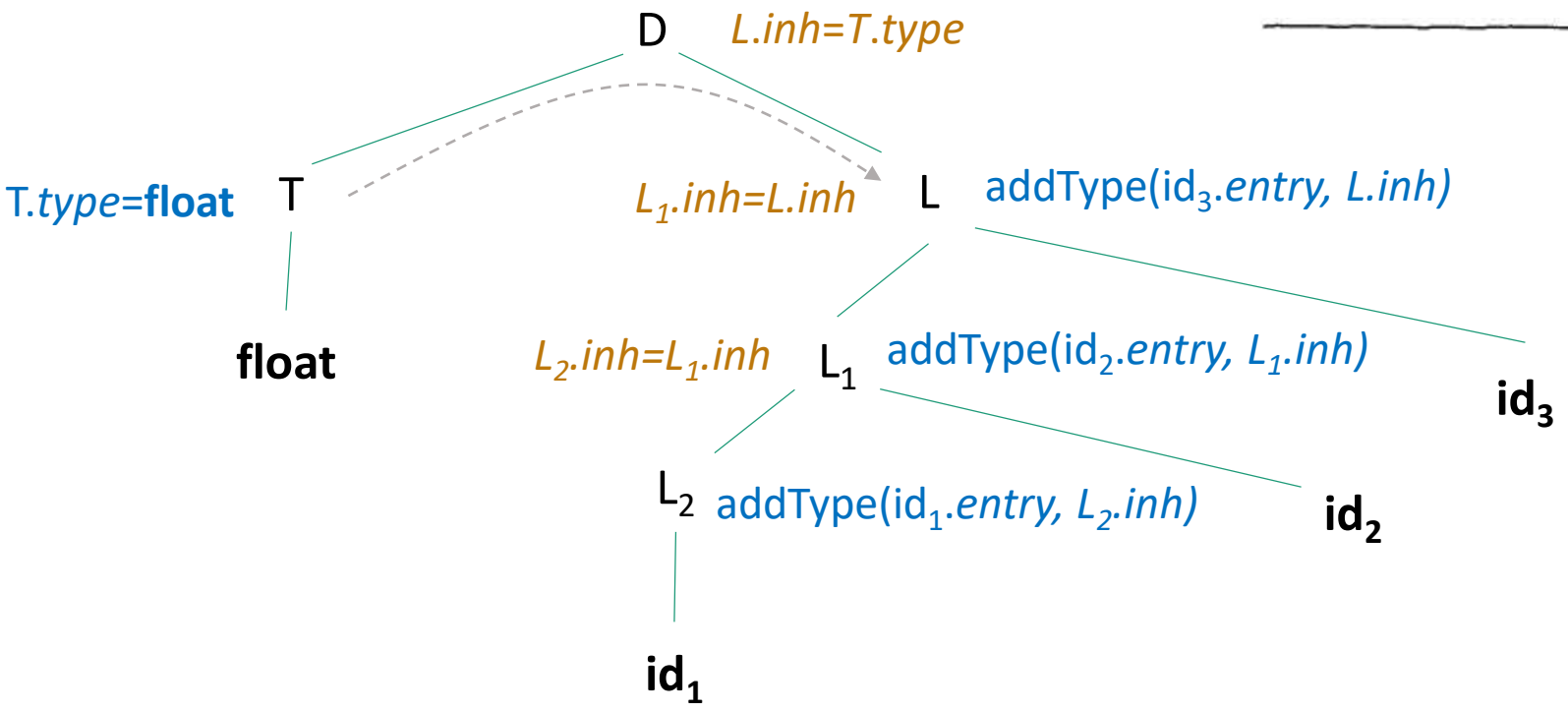
Types and Declarations

- Application of types
  - Type Checking
    - Uses logical rules to reason about the behavior of a program at run time
    - Ensures that the types of the operands match the type expected by an operator
    - **Example:** the && operator in java expects it two operands to be booleans; the result type is also boolean
  - Translation Applications
    - From the type of a name, a compiler can determine its required storage at runtime
    - Type information is also needed
      - to calculate the address denoted by an array reference
      - to insert explicit type conversion
      - to choose the right version of an arithmetic operator

Computing Types

- Let us compute an SDD to support declarations like
  - int id1, id2, id3
  - float id1, id2, id3

PRODUCTION	SEMANTIC RULES



Parse tree for the declaration:
float id1, id2, id3

Type Expressions

- Type expressions are used to represent structures of types
- A type expression is either a basic type or is formed by applying an operator called a *type constructor* to a *type expression*
- The sets of basic types and constructors depend on the language to be checked

Example: The array type `int[2][3]` can be

- read as "array of two arrays of 3 integers each"
- written as a type expression array(2,array(3,integer))
  - the operator array takes two parameters, a number and a type

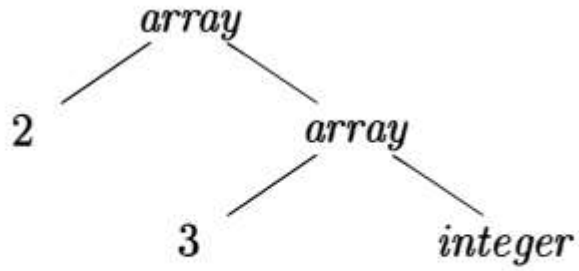


Figure 6.14: Type expression for `int[2][3]`

- Type expressions can be represented using a graph and the value-number method can used to store type expressions.

Definition of Type Expressions

- A basic type is a type expression such as *boolean, float, char, integer and void*
- A type name is a type expression such as name of a record, structure or class
- A type expression can be formed by applying the array type constructor to a number and a type expression
- A record is a data structure with named fields. A type name can be formed by applying the record type constructor to the field names and their types
- A type expression can be formed by using the type constructor → for function types.
  - s→t denotes "function from type s to type t
- If s and t are type expressions, then their Cartesian product s x t is a type expression.
  - Is used to represent a list or tuple of tuples (e.g., for function parameters)
- Type expressions may contain variables whose values are type expressions

Type Equivalence

- Many type-checking rules have the form, "if two type expressions are equal then return a certain type else error"
- When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions hold
  - They are the same basic type.
  - They are formed by applying the same constructor to structurally equivalent types
  - One is a type name that denotes the other

Storage Layout for Local Names

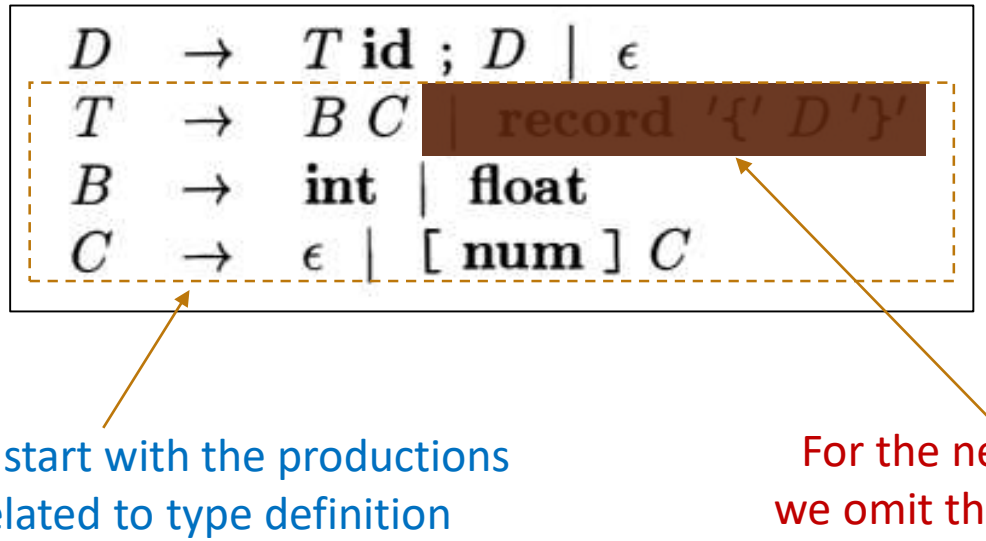
- From the type of a name, the amount of storage required for it during runtime can be determined
- At compile time, these amounts can be used to assign each name a relative address
- The type and relative address are saved in the symbol-table entry for the name
- Data of varying length, such as strings, dynamic arrays is handled by reserving a known fixed amount of storage for a pointer to the data

Storage Layout for Local Names (2)

- The width of a type is the number of storage units needed for objects of that type
  - we consider a byte as the smallest unit of addressable memory
- A basic type, such as a character, integer, or float, requires an integral number of bytes
- For easy access, storage for aggregates such as arrays and classes is allocated in one contiguous block of bytes

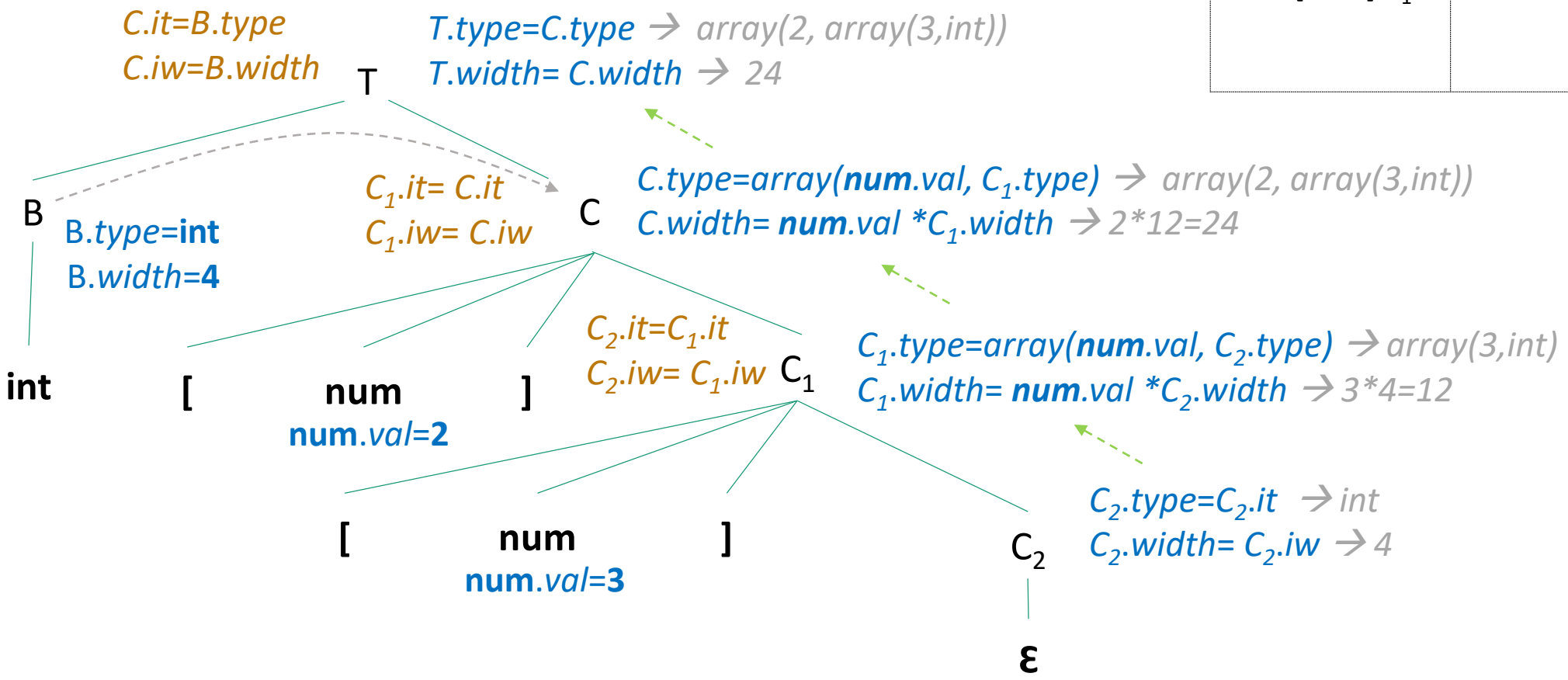
Computing Types and Their Widths

- A modified grammar
- Supports declarations like
  - int id; float id;
  - int[2][3][3] id;
  - record { int id; int[2][3] id2} id



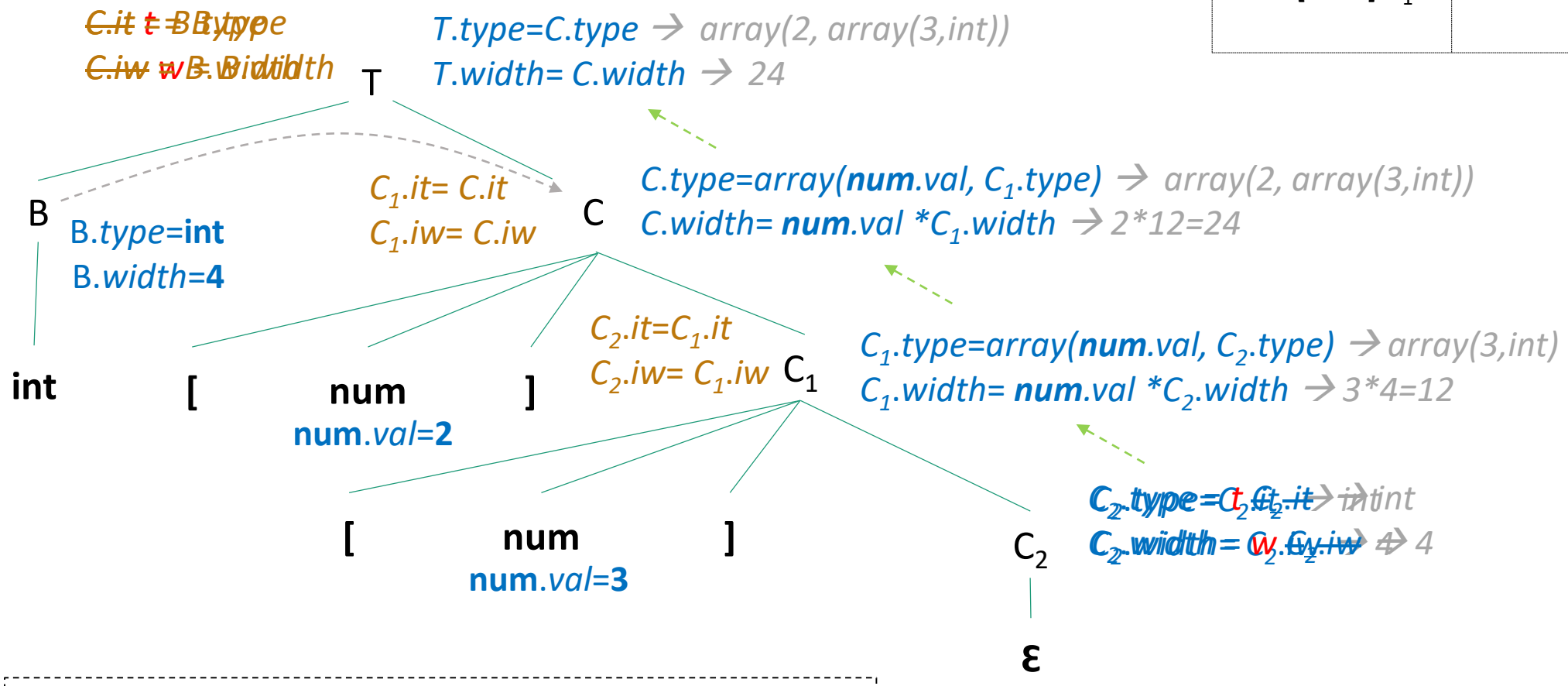
Prod.	Semantic Rules
T → B C	
B → **int**	
B → **float**	
C → ε	
C → [num] C_1	

Annotated Parse tree for **int [2][3]**



Prod.	Semantic Rules
T → B C	
B → **int**	
B → **float**	
C → ε	
C → [num] C_1	

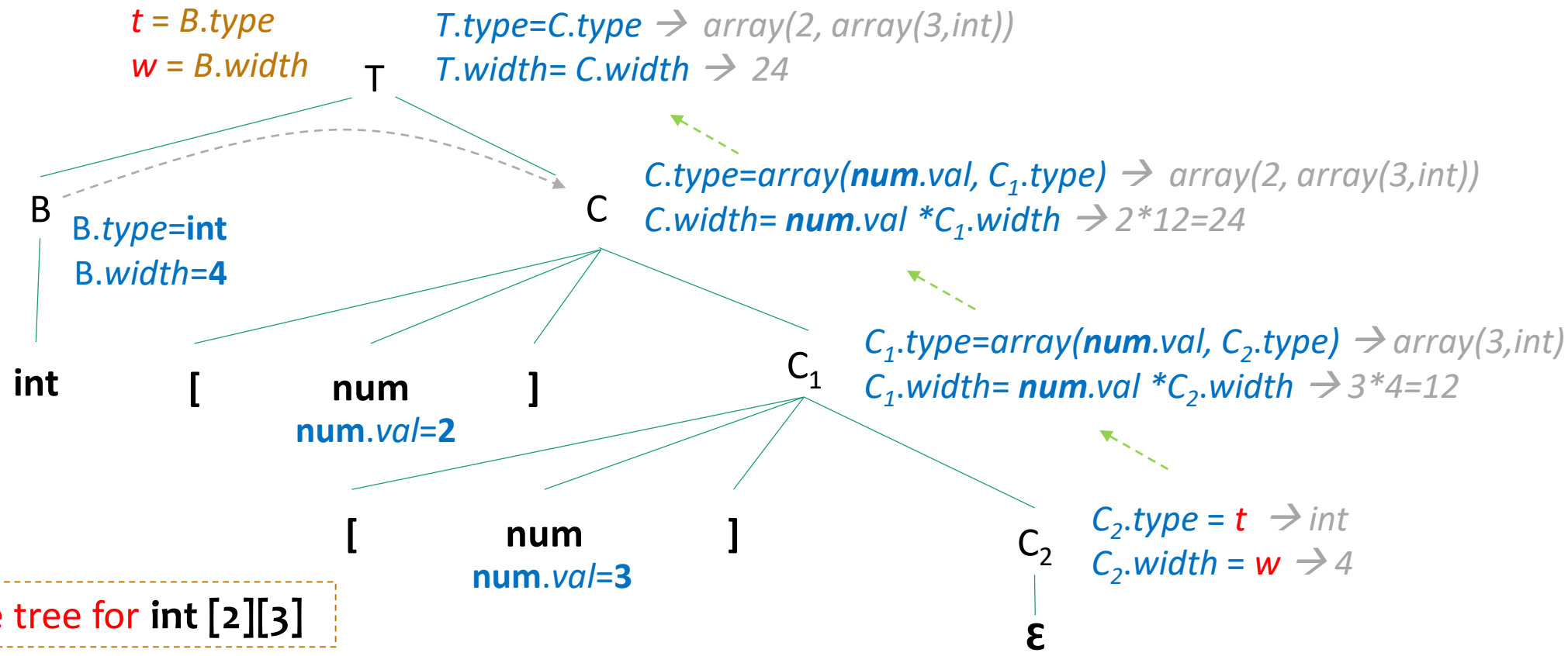
Annotated Parse tree for **int [2][3]**



Use of global variables instead of inherited attributes

SDT for Computing Type and Width	
T → B	{ *t* =*B.type*, *w*=*B.width* }
C	{ *T.type*=*C.type* *T.width*= *C.width* }
B → **int**	{ B.*type*=**float,** B.*width*=**4** }
B → **float**	{ B.*type*=**int,** B.*width*=**8** }
C → ε	{ *C.type* = *t*, *C.width* = *w* }
C → [num] C_1	{ *C.type*=*array*(***num***.*val*, C_1.*type*) *C.width*= ***num***.*val* *C_1.*width* }

Prod.	Semantic Rules
T → B C	*t*=*B.type*, *w*=*B.width* *T.type*=*C.type* *T.width*= *C.width*
B → **int**	B.*type*=**float,** B.*width*=**4**
B → **float**	B.*type*=**int,** B.*width*=**8**
C → ε	*C.type* = *t*, *C.width* = *w*
C → [num] C_1	*C.type*=*array*(***num***.*val*, C_1.*type*) *C.width*= ***num***.*val* *C_1.*width*



Annotated Parse tree for **int [2][3]**

Computing Relative Addresses

```cpp
int main(){
    int a,b,c;
    a=1;
    b=2;
    c=1;
    return 0;
}
```

CPP

```asm
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 1
        mov     DWORD PTR [rbp-8], 2
        mov     DWORD PTR [rbp-12], 1
        mov     eax, 0
        pop     rbp
        ret
```

ASM (x86-64 gcc 11.1)

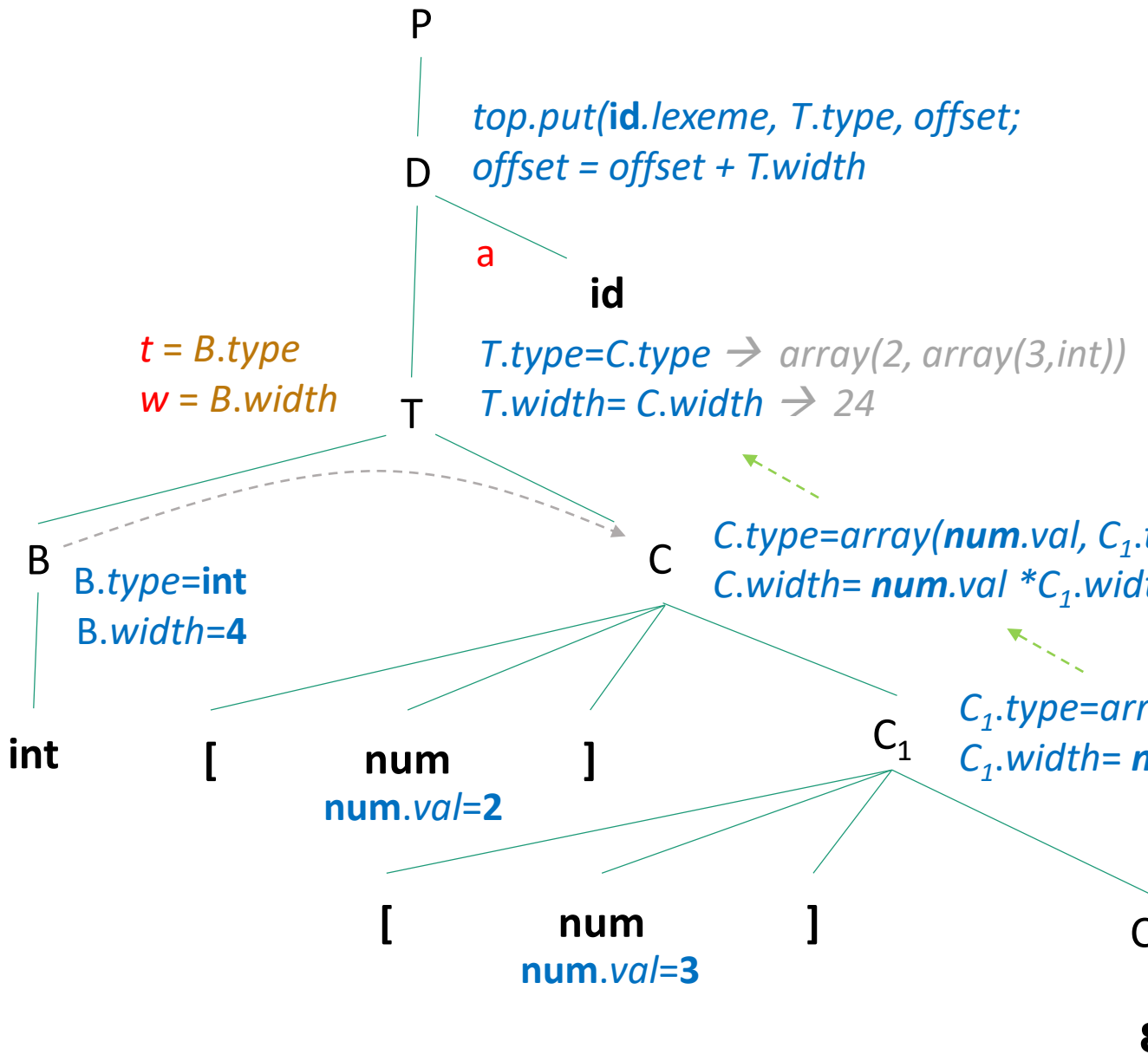
$$P \rightarrow \{\ offset = 0;\ \}\ D$$

$$D \rightarrow T\ \mathbf{id}\ ;\ \{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset);\ offset = offset + T.width;\ \}\ D_1$$

$$D \rightarrow \epsilon$$

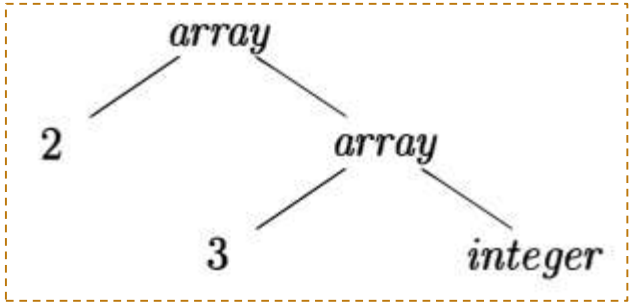
SDT for computing relative addresses

Annotated Parse tree for int [2][3] a

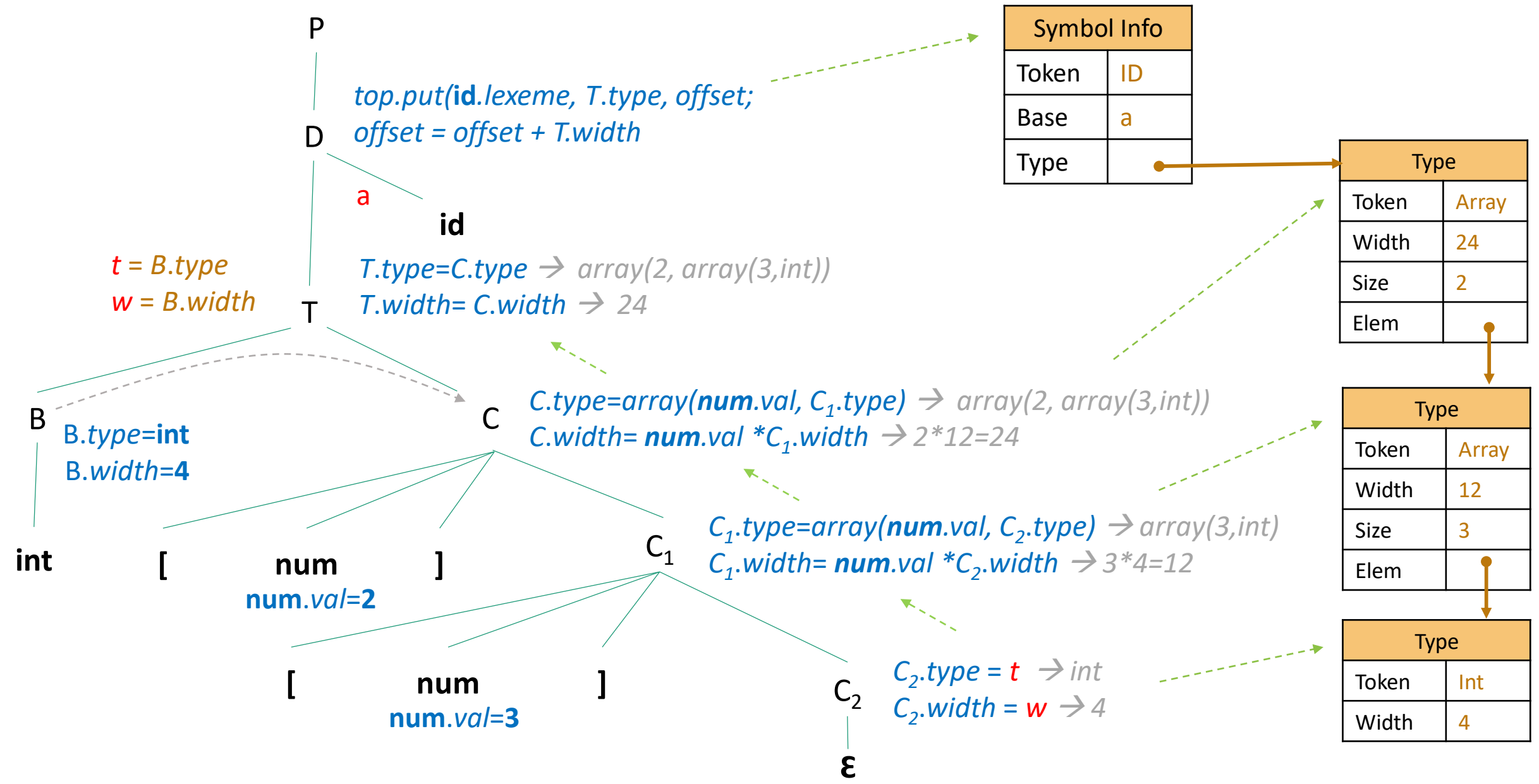


SDT for Computing Type and Width	
P → D	*{offset = 0;}*
D → T **id**	{ *top.put(**id**.lexeme, T.type, offset; offset = offset + T.width*}
T → B C	{ *t =B.type, w=B.width* } { *T.type=C.type T.width= C.width* }
B → **int**	{ B.*type*=**float,** B.*width*=**4** }
B → **float**	{ B.*type*=**int,** B.*width*=**8** }
C → ε	{ *C.type = t, C.width = w* }
C → [num] C_1	{ *C.type=array(**num**.val, C_1.type) C.width= **num**.val *C_1.width* }

Type expression for int[2][3]



Annotated Parse tree for **int [2][3] a**



Fields in Records and Classes

```cpp
struct Info {
    int x, y, z;
};
int main(){
    struct Info a;
    struct Info b;
    int x;
    x=a.x+b.x;
    return 0;
}
```

CPP

```asm
main:
        push    rbp
        mov     rbp, rsp
        mov     edx, DWORD PTR [rbp-16]
        mov     eax, DWORD PTR [rbp-28]
        add     eax, edx
        mov     DWORD PTR [rbp-4], eax
        mov     eax, 0
        pop     rbp
        ret
```

ASM (x86-64 gcc 11.1)

$$T \rightarrow \textbf{record}\ '\{'\ D\ '\}'$$

$$T \rightarrow \textbf{record}\ '\{'\quad \{\ Env.push(top);\ top = \textbf{new}\ Env();\ Stack.push(offset);\ offset = 0;\ \}$$

$$D\ '\}'\quad \{\ T.type = record(top);\ T.width = offset;\ top = Env.pop();\ offset = Stack.pop();\ \}$$

type constructor

symbol table object

Handling of field names in records

Three Address Code Generation

An Example

```
1) {                        // File test
2)     int i; int j; float v; float x; float[100] a;
3)     while( true ) {
4)         do i = i+1; while( a[i] < v);
5)         do j = j-1; while( a[j] > v);
6)         if( i >= j ) break;
7)         x = a[i]; a[i] = a[j]; a[j] = x;
8)     }
9) }
```

```
 1) L1:L3:   i = i + 1
 2) L5:      t1 = i * 8
 3)          t2 = a [ t1 ]
 4)          if t2 < v goto L3
 5) L4:      j = j - 1
 6) L7:      t3 = j * 8
 7)          t4 = a [ t3 ]
 8)          if t4 > v goto L4
 9) L6:      iffalse i >= j goto L8
10) L9:     goto L2
11) L8:     t5 = i * 8
12)         x = a [ t5 ]
13) L10:    t6 = i * 8
14)         t7 = j * 8
15)         t8 = a [ t7 ]
16)         a [ t6 ] = t8
17) L11:    t9 = j * 8
18)         a [ t9 ] = x
19)         goto L1
20) L2:
```

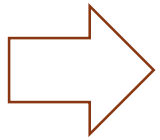
Translation of Expressions

- Need to convert an expression with more than one operator in source program to Three Address Codes with only one operator
  - $a * b + c$
- An array reference will expand in sequence of Three Address Codes that calculate the address for the reference
  - $A[i][j]$

Translation of Expressions

```
int initial = 32;
float rate = .8;
float position = initial + rate * 60 ;
                 \_____E_____/
```

Source code

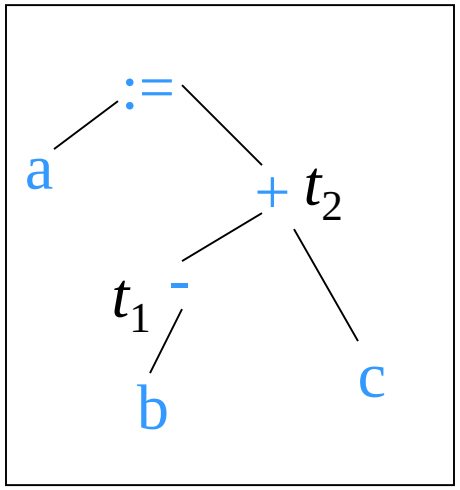


```
int t1 = 32
int initial = t1
float t2 = .8
float rate = t2
int t3 = initial
float t4 = rate
int t5 = 60                          E.code
float t6 = (float) t3
float t7 = t4 * t6
float t8 = t6 + t7
                    ↓ E.addr
float position = t8
```

Corresponding Three Address Code

Translation of Expressions

An Example Expression: $a := -(b) + c$

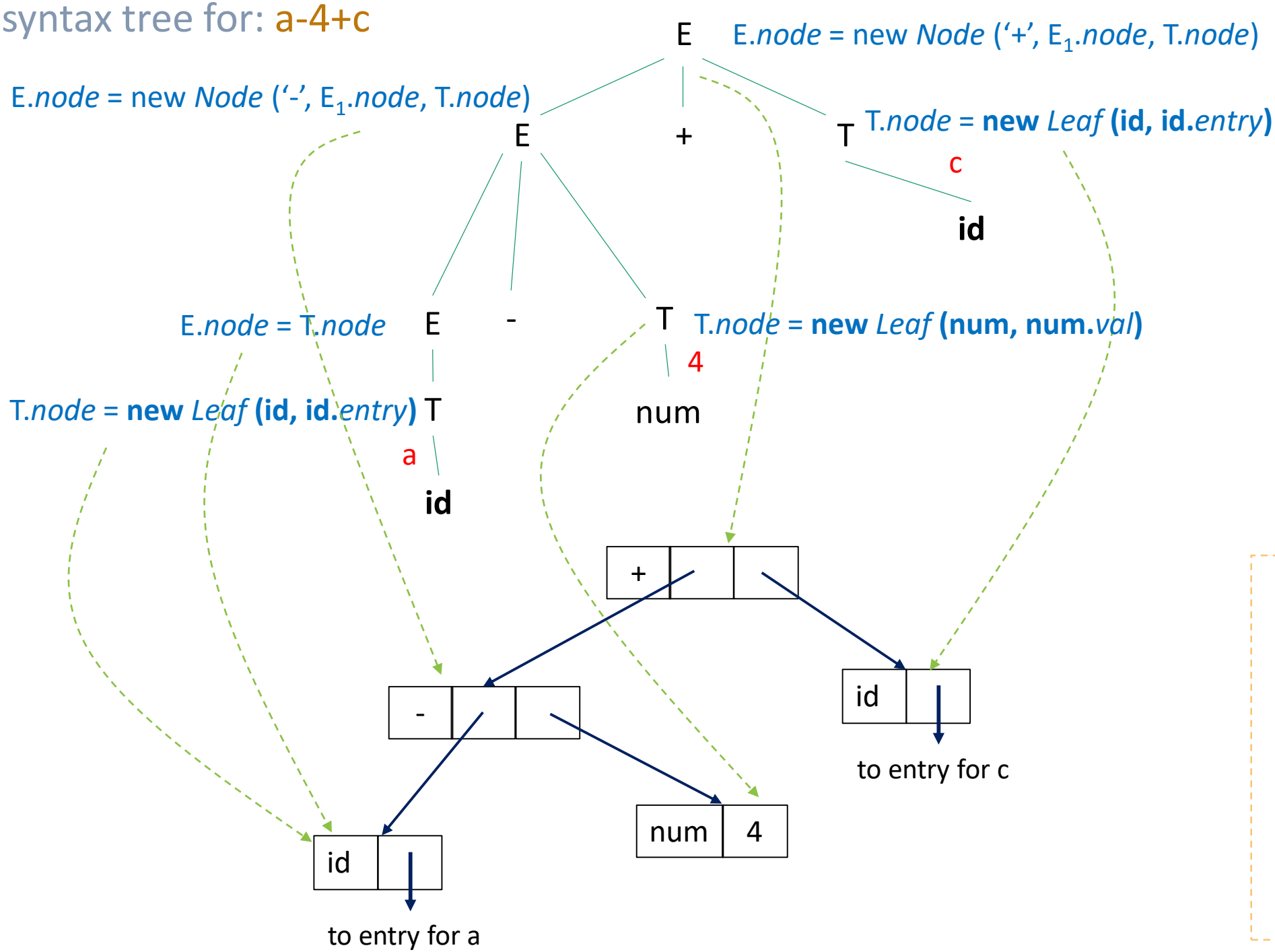


Syntax Tree

$$t_1 = \textbf{minus}\ b$$
$$t_2 = t_1 + c$$
$$a = t_2$$

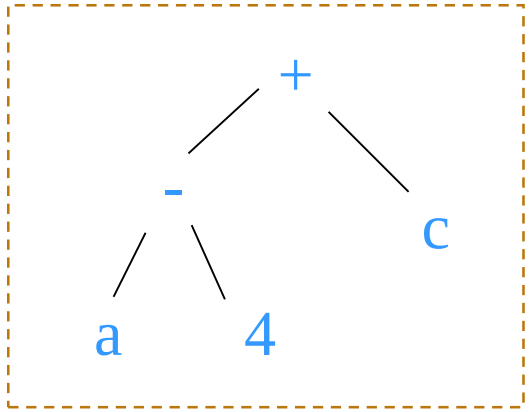
Three Address Code

Construct syntax tree for: a-4+c

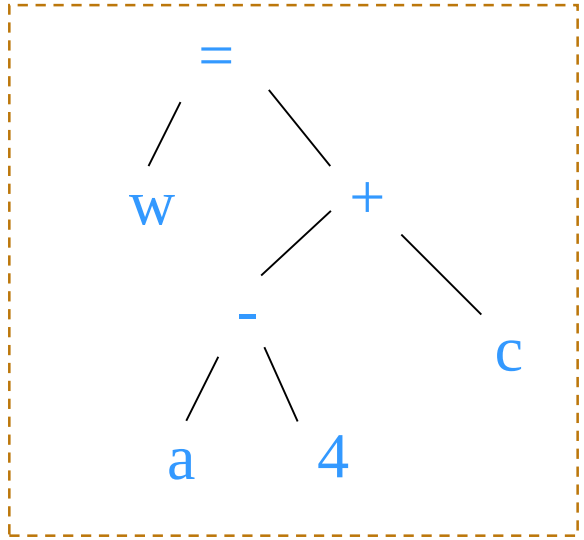


Production
$E \rightarrow E_1 + T$
$E \rightarrow E_1 - T$
$E \rightarrow T$
$T \rightarrow (E)$
$T \rightarrow$ **id**
$T \rightarrow$ num

AST for a-4+c



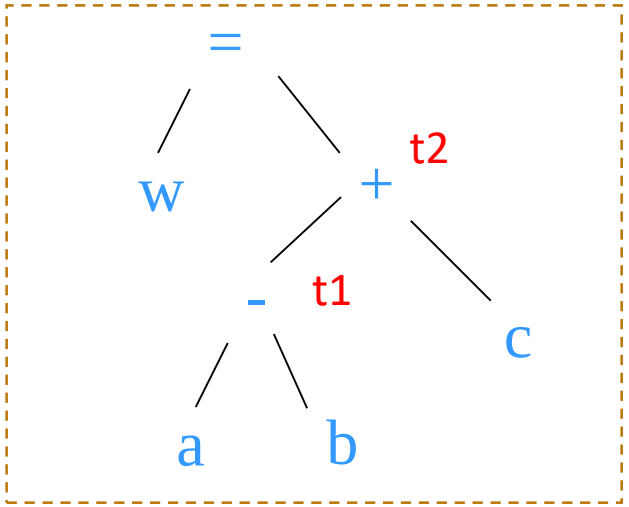
AST for w=a-4+c



Prod.	Semantic Rules
E → E_1 + T	E.*node* = new *Node* ('+', E_1.*node*, T.*node*)
E → E_1 – T	E.*node* = new *Node* ('-', E_1.*node*, T.*node*)
E → T	E.*node* = T.*node*
T → (*E*)	E.*node* = E.*node*
T → **id**	T.*node* = **new** *Leaf* **(id, id.***entry***)**
T → **num**	T.*node* = **new** *Leaf* **(num, num.***val***)**

Prod.	Semantic Rules
S → **id** = E	S.*node* = new *Node* (':=', new Leaf (id, id.entry), E.*node*)
E → E_1 + T	E.*node* = new *Node* ('+', E_1.*node*, T.*node*)
E → E_1 – T	E.*node* = new *Node* ('-', E_1.*node*, T.*node*)
E → T	E.*node* = T.*node*
T → (*E*)	E.*node* = E.*node*
T → **id**	T.*node* = **new** *Leaf* **(id, id.***entry***)**
T → **num**	T.*node* = **new** *Leaf* **(num, num.***val***)**

AST for w:=a-b+c



Three address code for w:=a-4+c

t1 = a - b

t2 = t1 + c

w = t2

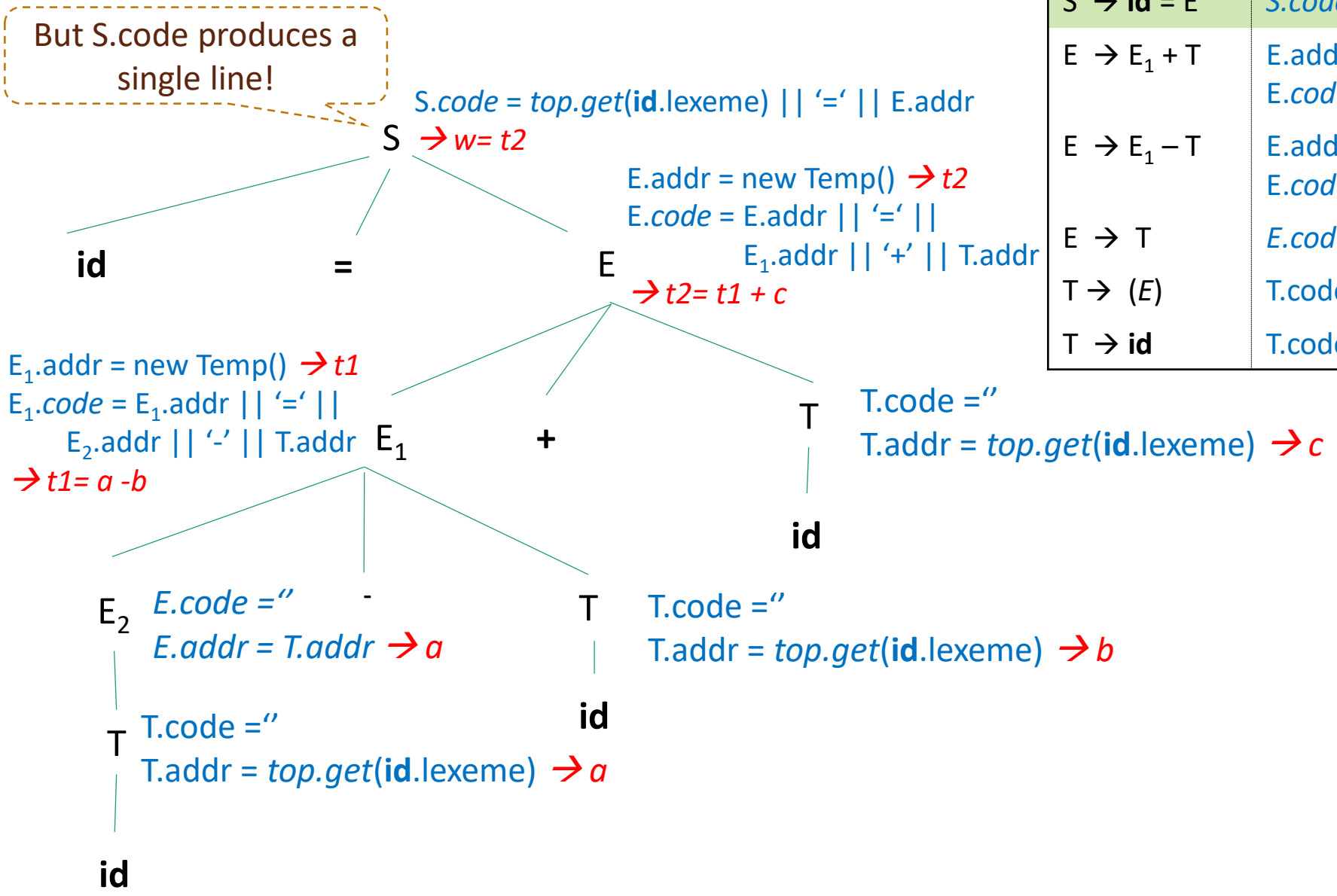
Prod.	Semantic Rules for AST
S → **id** = E	S.*node* = new *Node* (':=', new Leaf (id, id.entry), E.*node*)
E → E_1 + T	E.*node* = new *Node* ('+', E_1.*node*, T.*node*)
E → E_1 – T	E.*node* = new *Node* ('-', E_1.*node*, T.*node*)
E → T	E.*node* = T.*node*
T → (*E*)	T.*node* = E.*node*
T → **id**	T.*node* = **new** *Leaf* **(id, id.***entry***)**

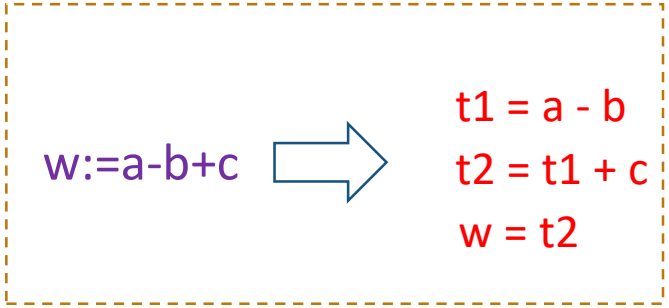
Prod.	Semantic Rules Three Address Code
S → **id** = E	
E → E_1 + T	
E → E_1 – T	
E → T	
T → (*E*)	
T → **id**	

Returns a pointer to symbol table of id

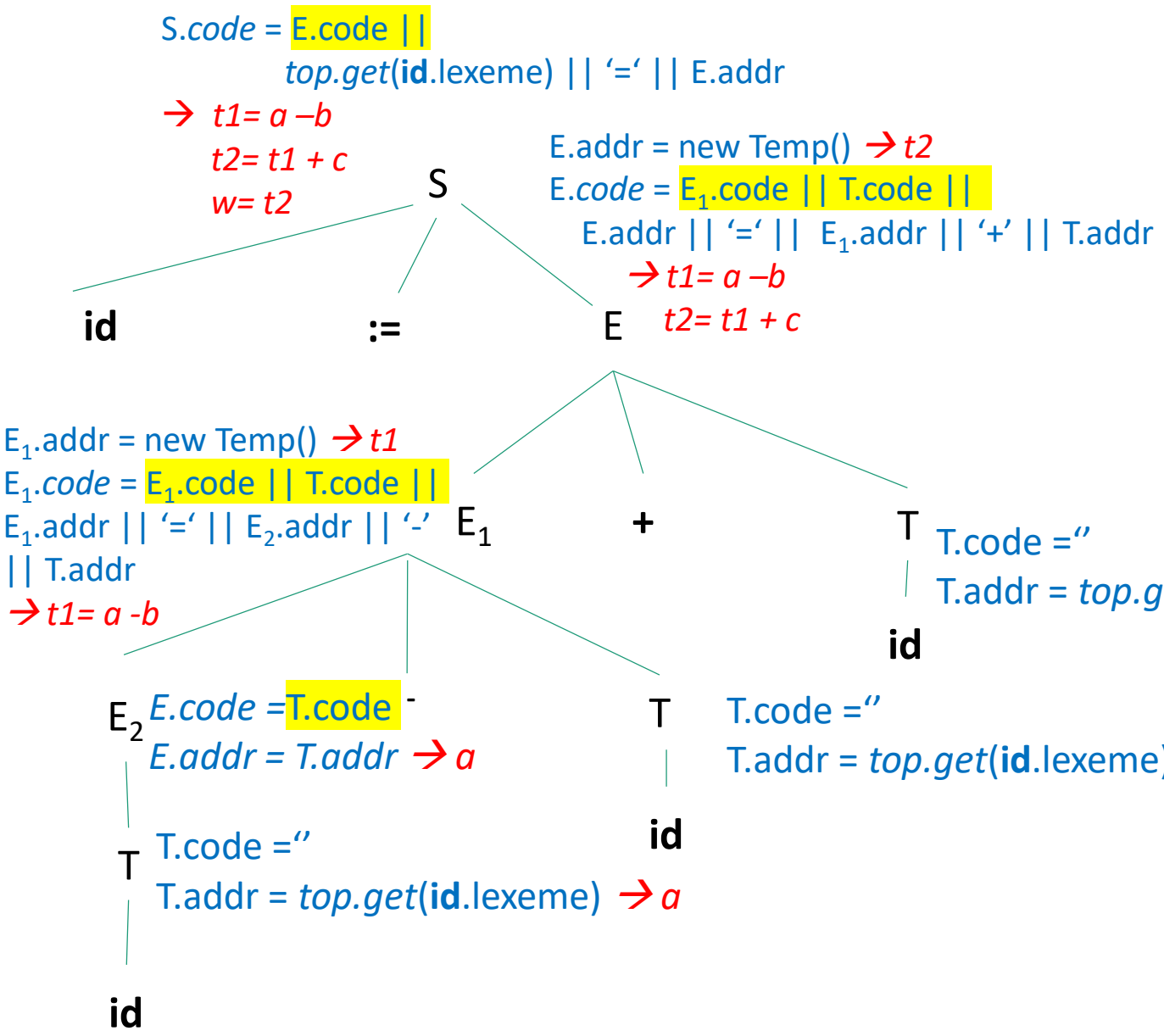
Three address code for w:=a-b+c

Prod.	Semantic Rules Three Address Code
S → **id** = E	*S.code = top.get*(**id**.lexeme) || '=' || E.addr
E → E_1 + T	E.addr = new Temp() E.*code* = E.addr || '=' || E_1.addr || '+' || T.addr
E → E_1 – T	E.addr = new Temp() E.*code* = E.addr || '=' || E_1.addr || '-' || T.addr
E → T	*E.code* = '', *E.addr = T.addr*
T → (*E*)	T.code = '', T.addr = *E.addr*
T → **id**	T.code = '', T.addr = *top.get*(**id**.lexeme)





Three address code for w:=a-b+c

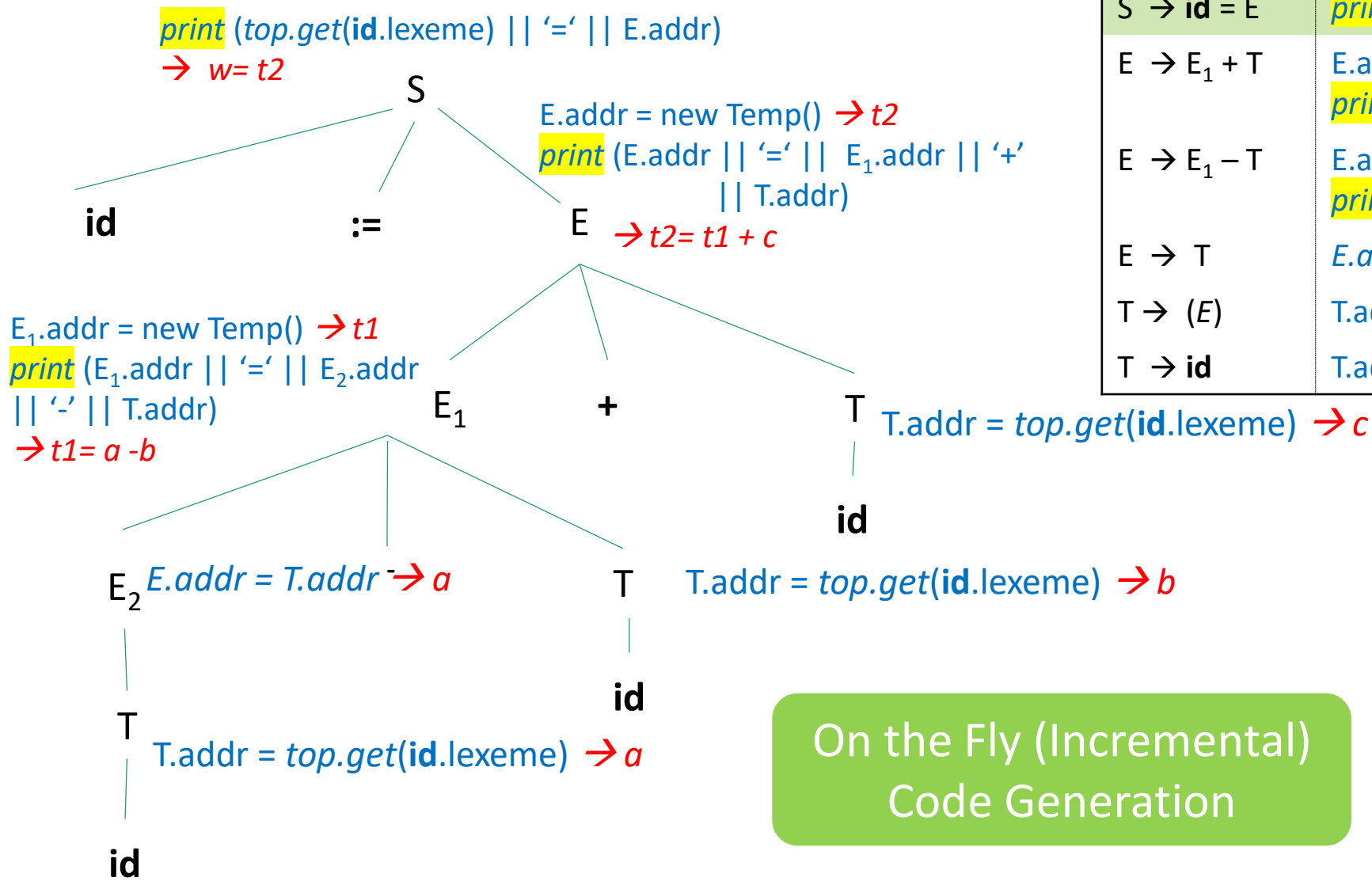


Prod.	Semantic Rules Three Address Code
S → **id** = E	*S.code* = *E*.code || *top.get*(**id**.lexeme) || '=' || E.addr
E → E_1 + T	E.addr = new Temp() E.*code* = E_1.code || T.code || E.addr || '=' || E_1.addr || '+' || T.addr
E → E_1 – T	E.addr = new Temp() E.*code* = E_1.code || T.code || E.addr || '=' || E_1.addr || '-' || T.addr
E → T	*E.code* =T.code, *E.addr* = *T.addr*
T → (*E*)	T.code =E.code, T.addr = *E.addr*
T → **id**	T.code ='', T.addr = *top.get*(**id**.lexeme)

w:=a-b+c ⇨

t1 = a - b
t2 = t1 + c
w = t2

Three address code for w:=a-b+c



Prod.	Semantic Rules Three Address Code
S → **id** = E	*print* (*top.get*(**id**.lexeme) || '=' || E.addr)
E → E_1 + T	E.addr = new Temp() *print* (E.addr || '=' || E_1.addr || '+' || T.addr)
E → E_1 – T	E.addr = new Temp() *print* (E.addr || '=' || E_1.addr || '-' || T.addr)
E → T	*E.addr = T.addr*
T → (*E*)	T.addr = *E.addr*
T → **id**	T.addr = *top.get*(**id**.lexeme)

On the Fly (Incremental) Code Generation

w:=a-b+c ⇨

t1 = a - b
t2 = t1 + c
w = t2

Translation of Expressions

CFG and SDD from your text

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

gen(…) builds an instruction and returns it

Incremental Translation

- Code attributes can be long strings, so they are usually generated incrementally
- Instead of building up E.code, generate only the new three address instructions
- The function gen(…), in addition to constructing new instruction, append to the stream

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \,\|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \,\|\| E_2.code \,\|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \,\|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

$S \rightarrow \mathbf{id} = E\ ;$ { $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr);$ }

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

$\mid\ - E_1$ { $E.addr = \mathbf{new}\ Temp();$
$gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr);$ }

$\mid\ (\ E_1\ )$ { $E.addr = E_1.addr;$ }

$\mid\ \mathbf{id}$ { $E.addr = top.get(\mathbf{id}.lexeme);$ }

Generating three-address code for expressions incrementally

Addressing Array Element

- In C and Java, array elements are numbered *0,1,…,n-1,* for an array of *n* elements
- If the width of each array element is *w*, then the *i*-th element of array A begins in location

$$base + i \times w$$

Here base is the relative address of A[0]

- The above formula generalized to *k*-dimensions as

$$base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$$

- For the array, int A[5][7][8], location of A[2][3][4] will be

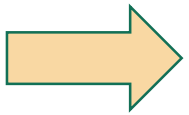
$$base + 2 \times 224 + 3 \times 32 + 4 \times 4$$

Here size of integer is assumed as 4 byte

Translation of Array References

- The chief problem in generating code for array references is to related the address calculation formulas to a grammar for array references
- For the array *a* declared as int[2][3], three-address translation of the expression –

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

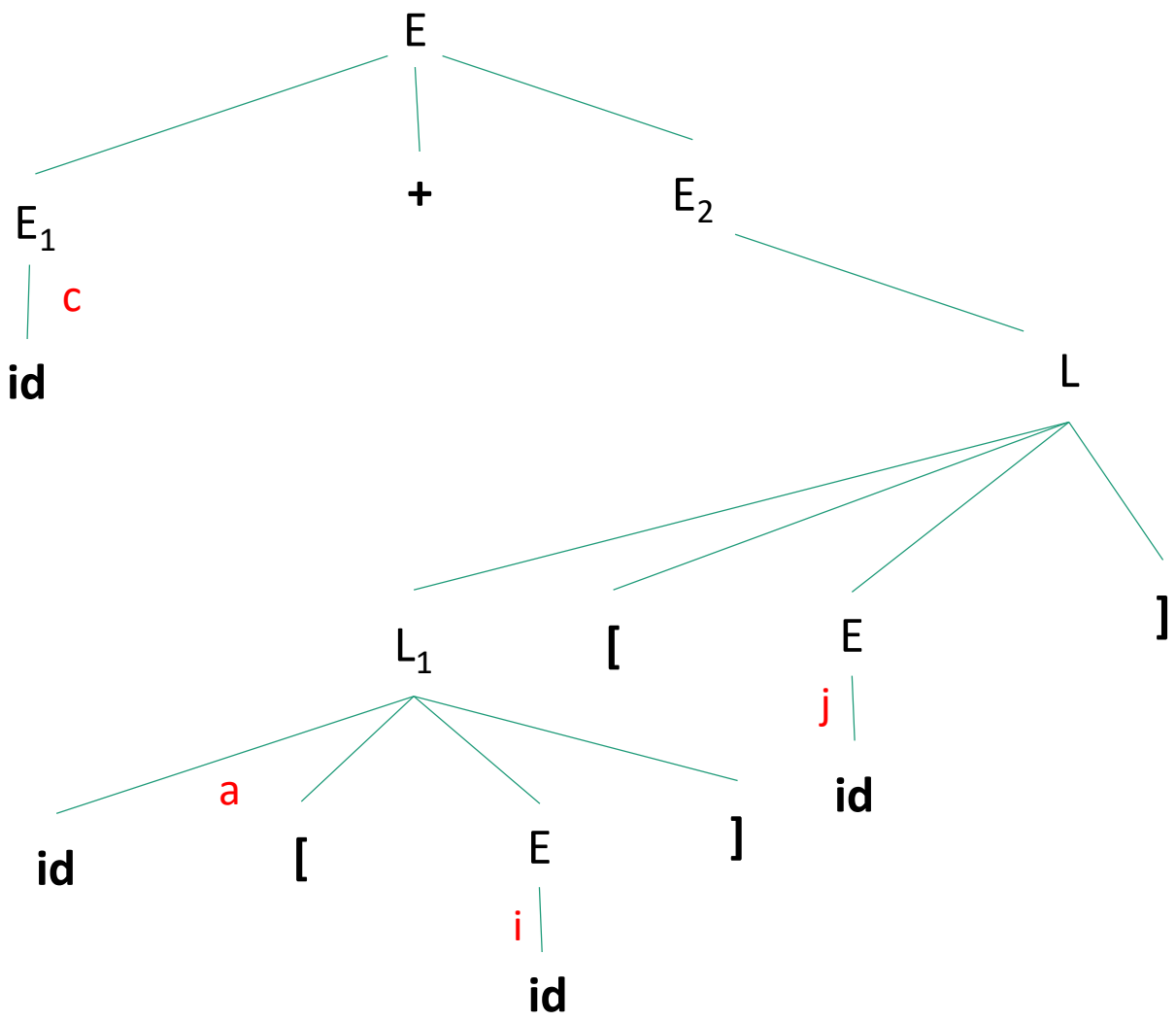


```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

- We will use the following grammar production rule where nonterminal *L* generates an array name followed by a sequence of index expression.

$$L \rightarrow L\,[\,E\,] \mid \mathbf{id}\,[\,E\,]$$

Translation of array references for c+a[i][j]

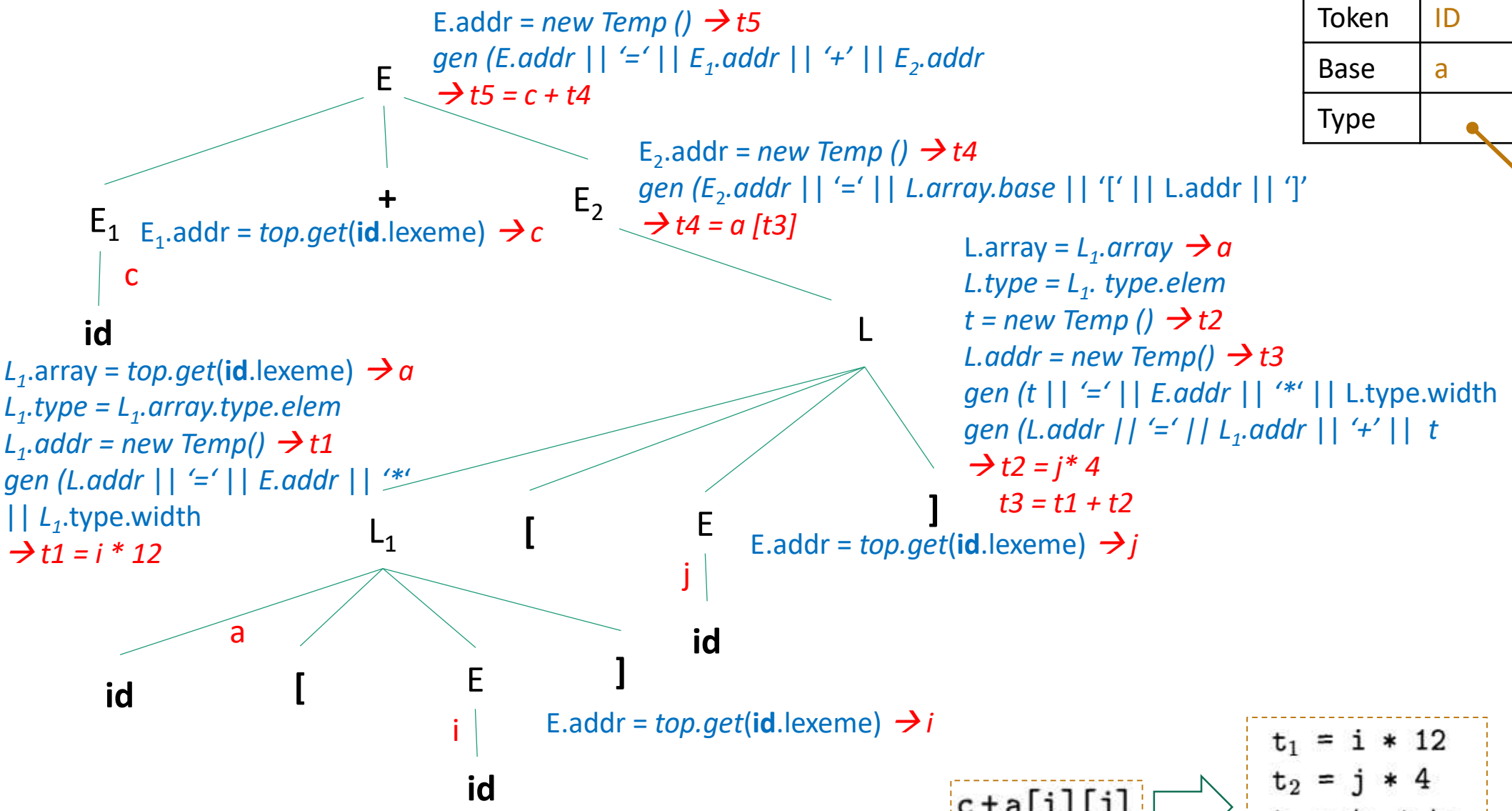


Prod.
S → **id** = E
S → L = E
E → E_1 + E_2
E → **id**
E → L
L → **id** [E]
L → L_1 [E]

c+a[i][j]

t_1 = i * 12
t_2 = j * 4
t_3 = t_1 + t_2
t_4 = a [t_3]
t_5 = c + t_4

Translation of array references for c+a[i][j]



Symbol Info	
Token	ID
Base	a
Type	

Type	
Token	Array
Width	24
Size	2
Elem	

Type	
Token	Array
Width	12
Size	3
Elem	

Type	
Token	Int
Width	4

c+a[i][j]

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

Translation of Array References (2)

Attributes of *L*

L.addr - a temporary that holds the sum of the terms $i_k \times w_k$

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

L.type - type of the subarray generated by *L*

$S \rightarrow$	**id** = E ;	{ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$; }
	L = E ;	{ $gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr)$; }
$E \rightarrow$	E_1 + E_2	{ $E.addr = \textbf{new}\ Temp()$; $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }
	id	{ $E.addr = top.get(\textbf{id}.lexeme)$; }
	L	{ $E.addr = \textbf{new}\ Temp()$; $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }
$L \rightarrow$	**id** [E]	{ $L.array = top.get(\textbf{id}.lexeme)$; $L.type = L.array.type.elem$; $L.addr = \textbf{new}\ Temp()$; $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; }
	L_1 [E]	{ $L.array = L_1.array$; $L.type = L_1.type.elem$; $t = \textbf{new}\ Temp()$; $L.addr = \textbf{new}\ Temp()$; $gen(t\ '='\ E.addr\ '*'\ L.type.width)$; } $gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; }

Translation of Array References (2)

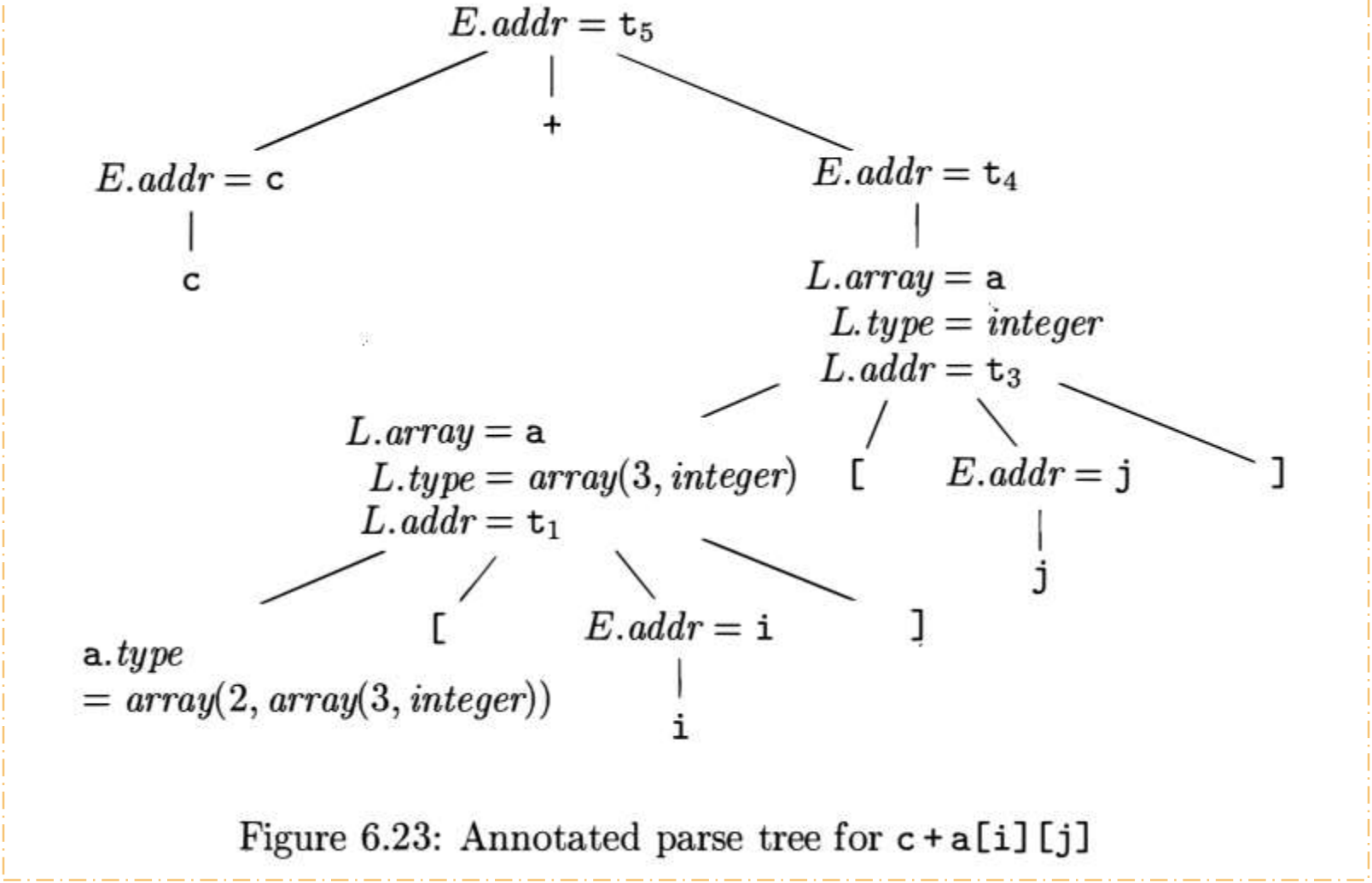


Figure 6.23: Annotated parse tree for c + a[i][j]

Control Flow

- We will translate control flow statements such as *if-else*, *while*
- Translation of control flow statements are tied with Boolean expressions
- Boolean expressions are composed of Boolean operators
- Boolean operators are applied to
  - Boolean variables
  - Relational expression with relational operator

Boolean Expression

- We consider Boolean expressions generated by following grammar

$$B \rightarrow B \,||\, B \mid B \,\&\&\, B \mid !\, B \mid (\, B \,) \mid E \,\mathbf{rel}\, E \mid \mathbf{true} \mid \mathbf{false}$$

- **rel** is relational operator which is any one of < , > , <= , >= , ==, !=

- We assume &&, || are left associative and precedence order is !, && then ||

Boolean Expression

- Boolean expressions are used to
  - Alter flow of control
    - If (B) then S: B must be true to reach S

```
if ( x < 100 || x > 200 && x != y )
      x = 0;
```

  - To compute logical values
    - True or False as value

```
p = x<a && (x>b || y<a);
```

Short Circuit Code

- Also called jumping code
- &&, || and ! are translated into jumps
- The operators don’t appear in the code
- The value of a boolean expression is represented by a position in the code

Short Circuit Code

- Example:

A statement with Boolean expressions

```
if ( x < 100 || x > 200 && x != y )
     x = 0;
```

Jumping Code

```
    if x < 100 goto L2
    ifFalse x > 200 goto L1
    ifFalse x != y goto L1
L2: x = 0
L1:
```

Flow of Control Statements

- We will consider flow of control statements generated by following grammar

$$S \rightarrow if\ (B)\ S_1$$

$$S \rightarrow if\ (B)\ S_1\ else\ S_2$$

$$S \rightarrow while\ (B)\ S_1$$

- B represent Boolean expression
- S represent Statements

Flow of Control Statements

- The Grammar

Productions	Semantic Rules
$P \rightarrow S$	
$S \rightarrow S_1 S_2$	
$S \rightarrow$ **assign**	
$S \rightarrow if\ (B)\ S_1$	
$S \rightarrow if\ (B)\ S_1\ else\ S_2$	
$S \rightarrow while\ (B) S_1$	

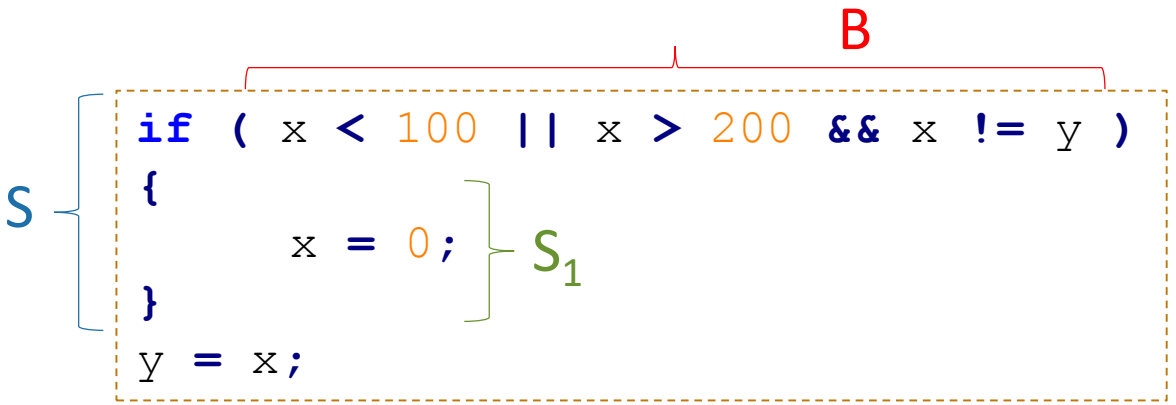
assign is a Placeholder for an assignment statement

We need to find out the Semantic Rules

Flow of Control Statements

- Semantic Rules for the production

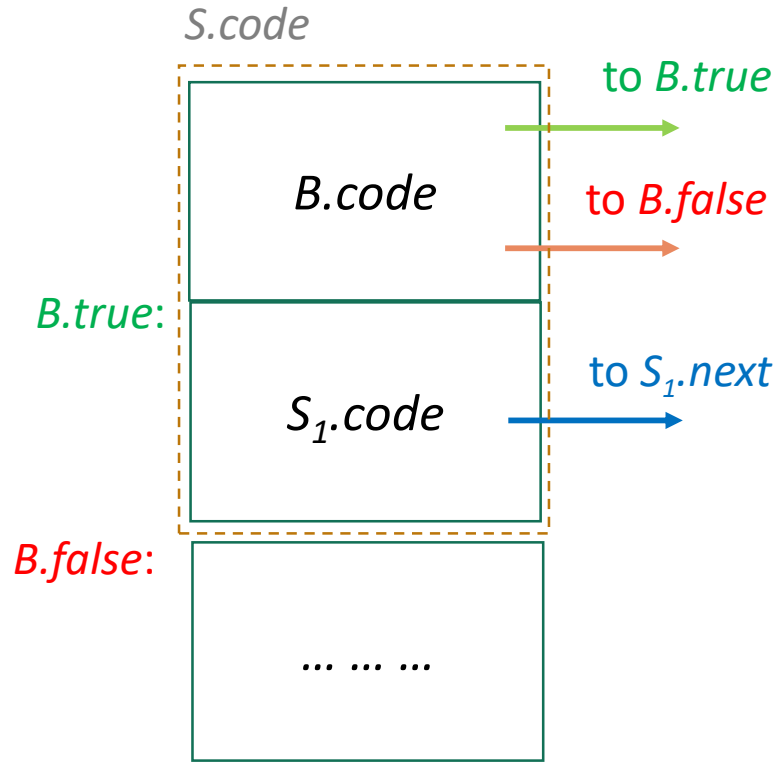
$$S \rightarrow if\ (B)\ S_1$$



$newlabel()$ creates new label each time it is called

Productions	Semantic Rules
$S \rightarrow if\ (B)\ S_1$	

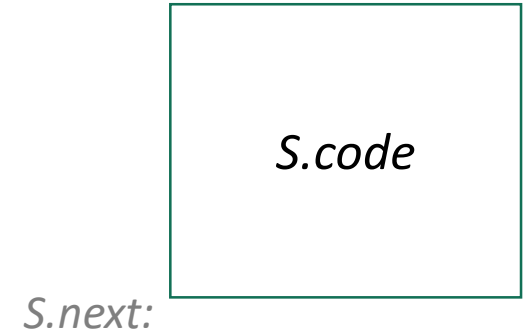
$label(L)$ attaches label L to the next instruction to be generated



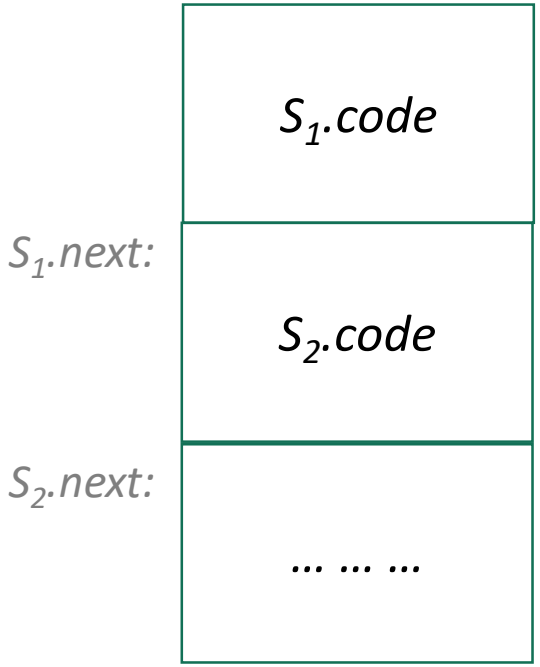
Flow of Control Statements

- Semantic Rules for the first two productions

Productions	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$



$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code$ $= S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$



Flow of Control Statements

- Attributes of SDD:
  - Codes are managed using
    - B.code : contains codes of B
    - S.code : contains codes of S

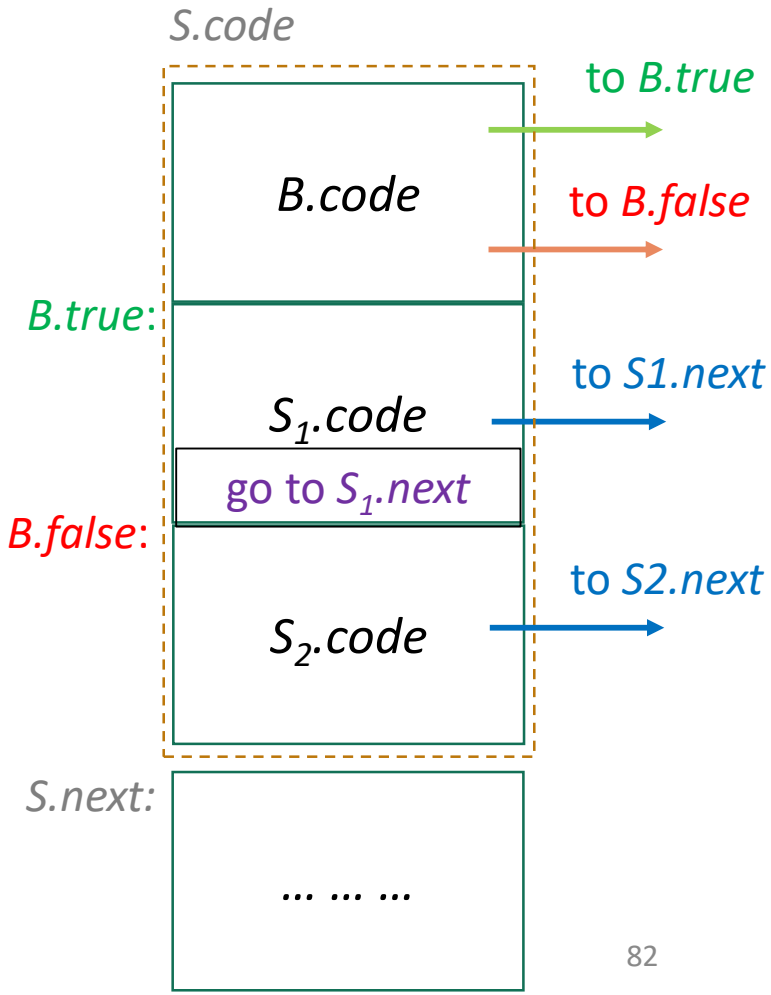
    } Synthesized
  - Jumping Labels are managed using
    - B.true : label to which control flows if B is true
    - B.false : label to which control flows if B is false
    - S.next : label of an instruction immediately after S

    } Inherited

Flow of Control Statements

- Semantic Rules for the production $S \rightarrow if\ (B)\ S_1\ else\ S_2$

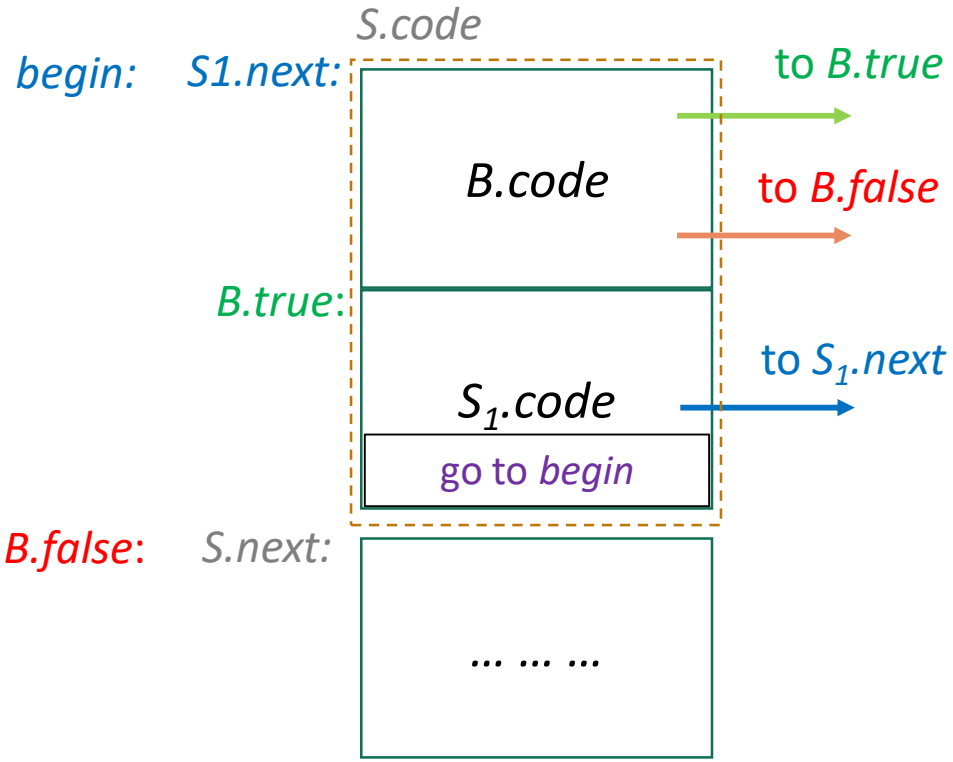
Productions	Semantic Rules
$S \rightarrow if\ (B)\ S_1\ else\ S_2$	



Flow of Control Statements

- Semantic Rules for the production $S \rightarrow while\ (B)\ S_1$

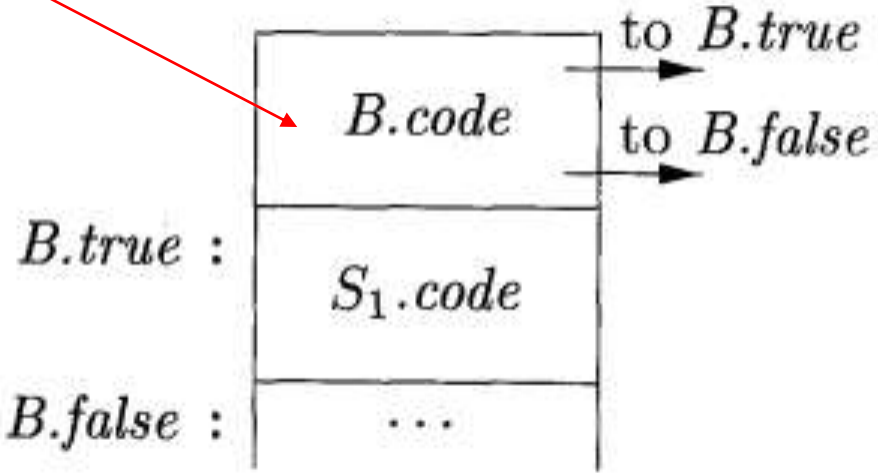
Productions	Semantic Rules
$S \rightarrow while\ (B) S_1$	



Translation of Boolean Expression

- Now we will consider productions containing B in their head
  - We want to know how we produce $B.code$

How to produce B.code?



Translation of Boolean Expression

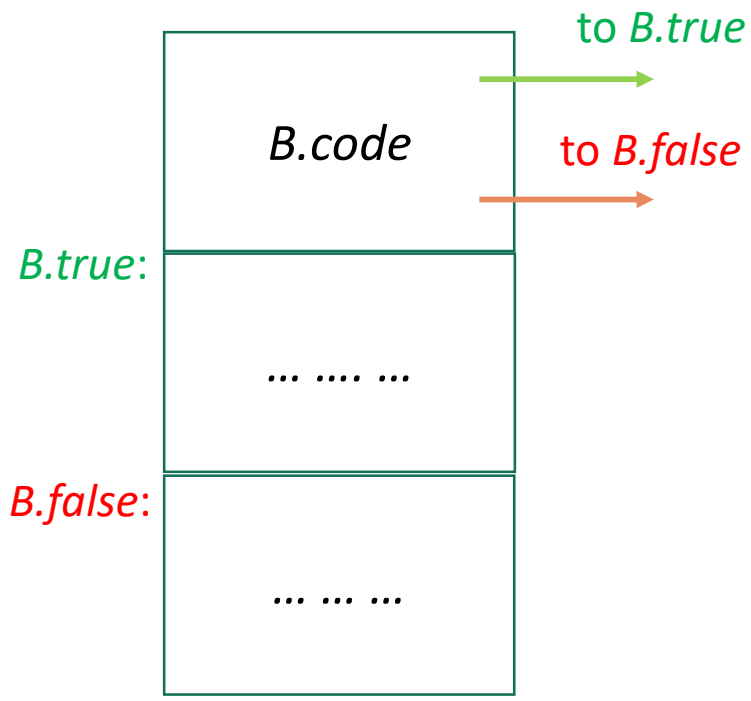
- The Grammar

Productions	**Semantic Rules**
$B \rightarrow B_1 \mid\mid B_2$	
$B \rightarrow B_1 \,\&\&\, B_2$	
$B \rightarrow !\, B_1$	
$B \rightarrow E_1 \ \mathbf{rel}\ E_2$	
$B \rightarrow \mathbf{true}$	
$B \rightarrow \mathbf{false}$	

Translation of Boolean Expression

- Semantic Rules for the production $B \rightarrow B_1 \,||\, B_2$

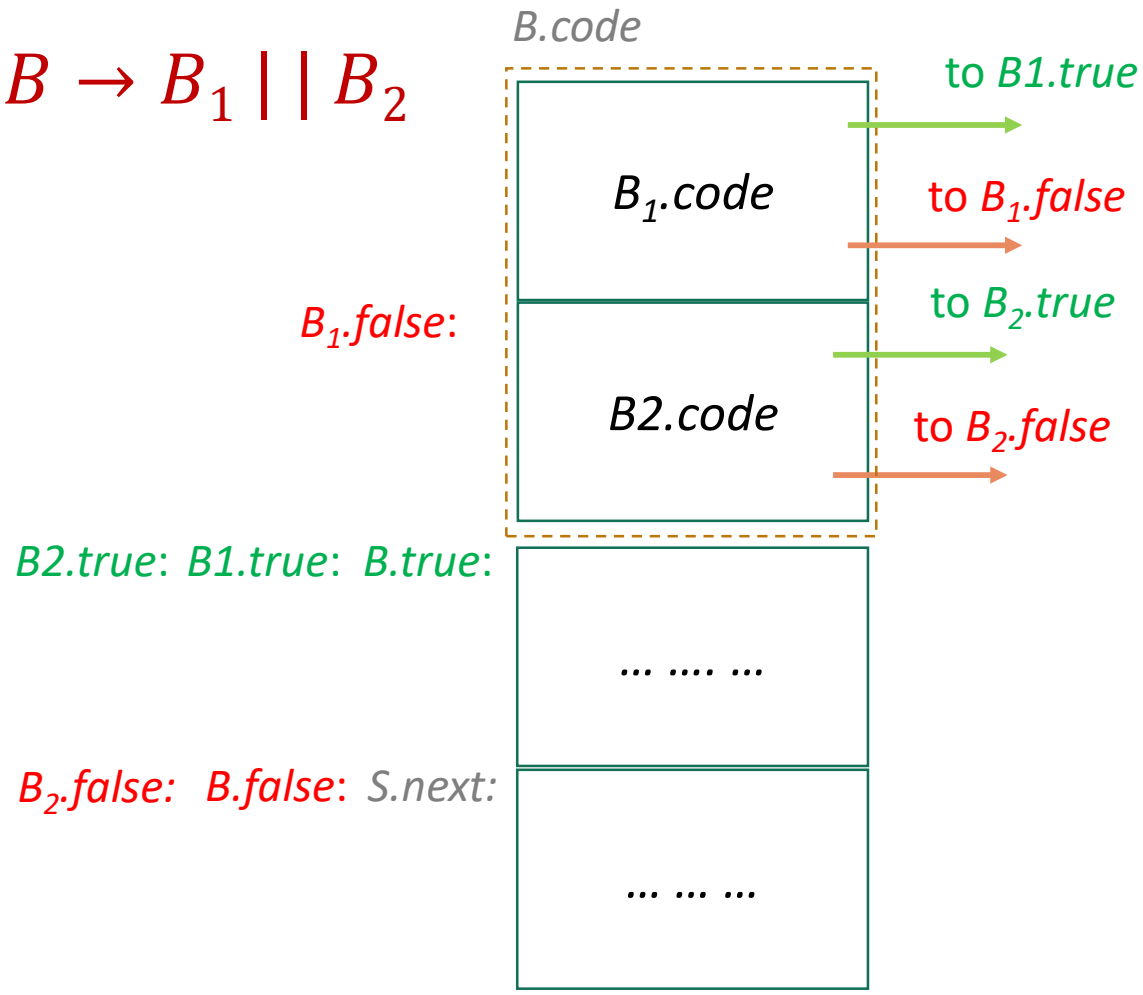
PRODUCTION	SEMANTIC RULE
$B \rightarrow B_1 \,\|\|\, B_2$	



Translation of Boolean Expression

- Semantic Rules for the production $B \rightarrow B_1 \,||\, B_2$

PRODUCTION	SEMANTIC RULE
$B \rightarrow B_1 \,\|\| B_2$	



Translation of Boolean Expression

- Consider $B \rightarrow B_1 \mid\mid B_2$
- If B_1 is true then B is also true. So label of $B_1.true$ is same as $B.true$
- If B_1 is false, then we need to evaluate B_2. So label of $B_1.false$ is the label of first instruction of B_2
- $B_2.true$ is same as $B.true$
- $B_2.false$ is same as $B.false$

Translation of Boolean Expression

- Semantic Rules for the production $B \rightarrow B_1 \,\&\&\, B_2$

PRODUCTION	SEMANTIC RULE
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true =$ $B_1.false =$ $B_2.true =$ $B_2.false =$ $B.code =$

Translation of Boolean Expression

- Semantic Rules for the production $B \rightarrow !\, B_1$

PRODUCTION	SEMANTIC RULE
$B \rightarrow !\, B_1$	

Translation of Boolean Expression

- Semantic Rules for the production $B \rightarrow E_1 \ \mathbf{rel} \ E_2$

PRODUCTION	SEMANTIC RULE
$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	

Translation of Boolean Expression

- Semantic Rules for the productions $B \rightarrow \mathbf{true}$ and $B \rightarrow \mathbf{false}$

PRODUCTION	SEMANTIC RULE
$B \rightarrow \mathbf{true}$	
$B \rightarrow \mathbf{false}$	

SDD for Generating Three Address Code for Flow-of-Control Statements

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

SDD for Generating Three Address Code for Booleans

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen('\texttt{goto}'\ B.false)$

Code Generation

SDD for Flow-of-Control and Booleans (*partial*)

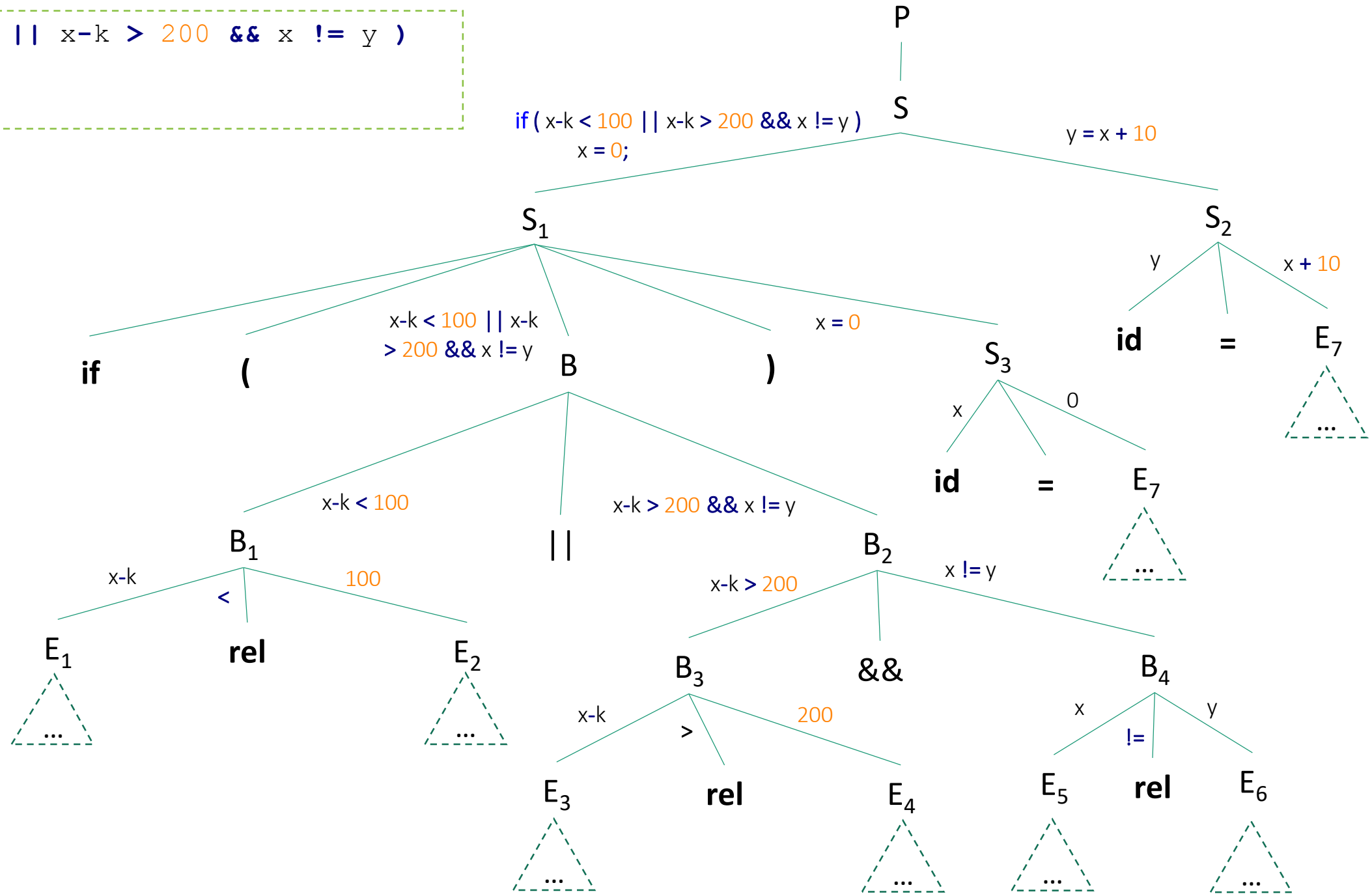
Production	Semantic Action
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$
$S \rightarrow$ **if** $(B)\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'}\ E_1.addr$ **rel** $E_2.addr\ \text{'goto}\ B.true\text{'})$ $\mid\mid gen(\text{'goto}\ B.false\text{'})$

SDT for Flow-of-Control and Booleans (on the fly)

Translation of statement

```
if ( x-k < 100 || x-k > 200 && x != y )
    x = 0;
y = x + 10;
```

Prod.
P → S
S → **id** = E;
S → S_1 S_2
S → **if** (B) S_1
B → B_1 **||** B_2
B → B_1 **&&** B_2
B → E_1 **rel** E_2
E → …

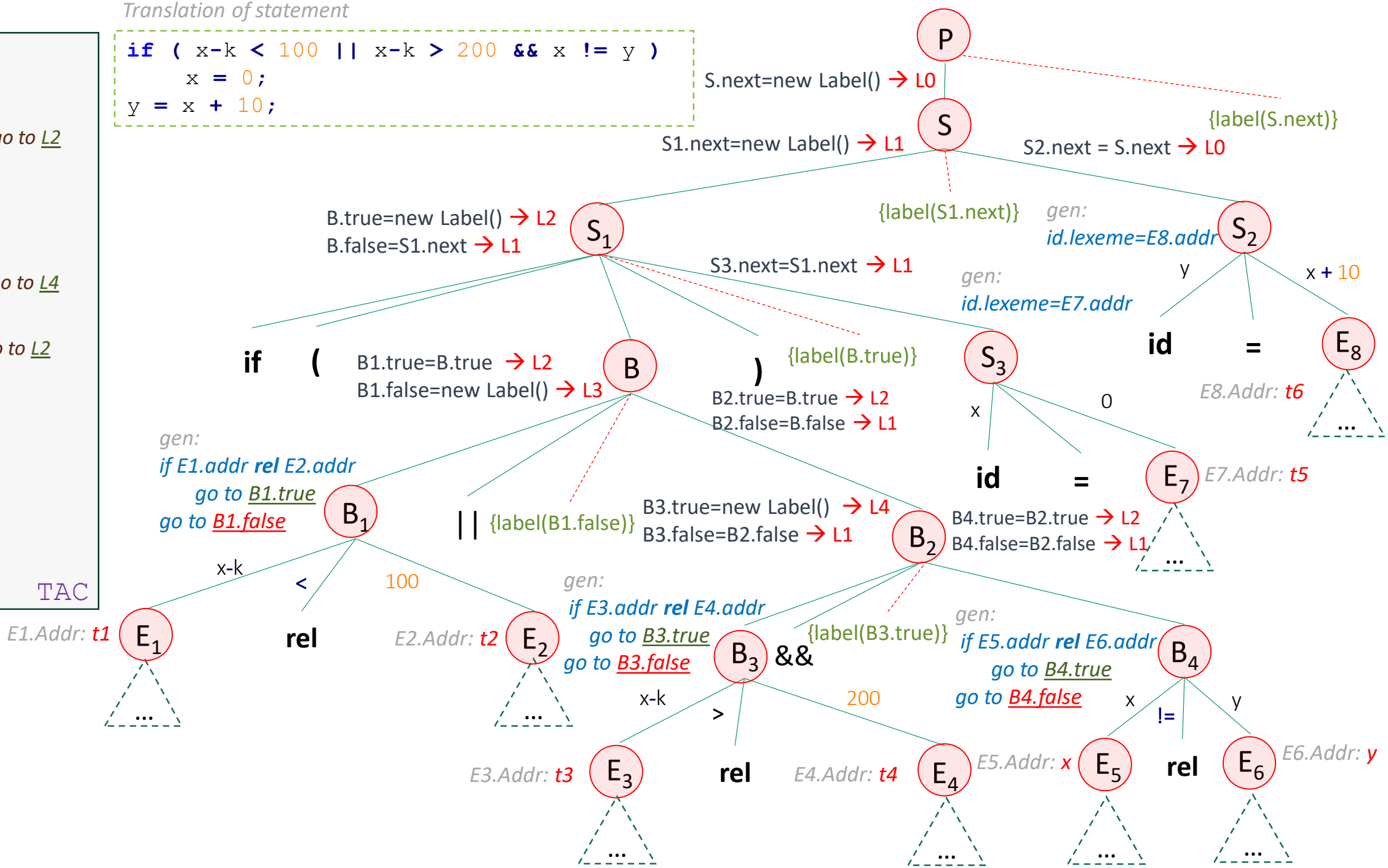


Translation of statement

```
if ( x-k < 100 || x-k > 200 && x != y )
     x = 0;
y = x + 10;
```

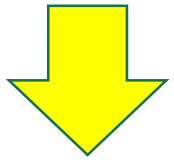
```
     t1 = x-k
     t2=100
     if t1 < t2  go to L2
     go to L1
L3:  t3 = x-k
     t4=200
     if t3 < t4  go to L4
     go to L1
L4:  if x != y  go to L2
     go to L1
L2:  t5=0
     x = t5
L1:  t6 = x+10
     y = t6
L0:
```

TAC



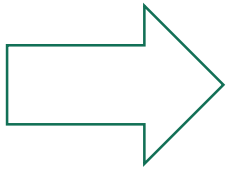
Avoiding Redundant Gotos

```
if( x < 100 || x > 200 && x != y ) x = 0;
```



```
        if x < 100 goto L2
Redundant  goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

Code with redundant goto



```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

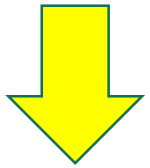
Code without redundant goto

Self Study (6.6.5)

Avoiding Redundant Gotos

$$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$$

$$B.true = newlabel()$$
$$B.false = S_1.next = S.next$$
$$S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$$



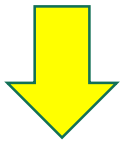
$$B.true = fall$$
$$B.false = S_1.next = S.next$$
$$S.code = B.code \mid\mid S_1.code$$

Self Study (6.6.5)

Avoiding Redundant Gotos

$$B \rightarrow B_1 \;||\; B_2$$

$$
\begin{aligned}
&B_1.true = B.true \\
&B_1.false = newlabel() \\
&B_2.true = B.true \\
&B_2.false = B.false \\
&B.code = B_1.code \;||\; label(B_1.false) \;||\; B_2.code
\end{aligned}
$$



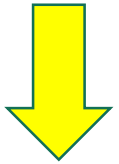
$$
\begin{aligned}
&B_1.true = \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel() \\
&B_1.false = fall \\
&B_2.true = B.true \\
&B_2.false = B.false \\
&B.code = \textbf{if } B.true \neq fall \textbf{ then } B_1.code \;||\; B_2.code \\
&\qquad\qquad \textbf{else } B_1.code \;||\; B_2.code \;||\; label(B_1.true)
\end{aligned}
$$

Self Study (6.6.5)

Avoiding Redundant Gotos

$$B \rightarrow E_1 \ \mathbf{rel} \ E_2$$

$$\begin{aligned} B.code = {} & E_1.code \ || \ E_2.code \\ & || \ gen(\texttt{'if'} \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ \texttt{'goto'} \ B.true) \\ & || \ gen(\texttt{'goto'} \ B.false) \end{aligned}$$



$$test = E_1.addr \ \mathbf{rel}.op \ E_2.addr$$

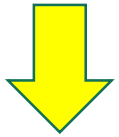
$$\begin{aligned} s = {} & \mathbf{if} \ B.true \neq fall \ \mathbf{and} \ B.false \neq fall \ \mathbf{then} \\ & \quad gen(\texttt{'if'} \ test \ \texttt{'goto'} \ B.true) \ || \ gen(\texttt{'goto'} \ B.false) \\ & \mathbf{else \ if} \ B.true \neq fall \ \mathbf{then} \ gen(\texttt{'if'} \ test \ \texttt{'goto'} \ B.true) \\ & \mathbf{else \ if} \ B.false \neq fall \ \mathbf{then} \ gen(\texttt{'ifFalse'} \ test \ \texttt{'goto'} \ B.false) \\ & \mathbf{else} \ '' \end{aligned}$$

$$B.code = E_1.code \ || \ E_2.code \ || \ s$$

Boolean Values and Jumping Code

- So far, we have used boolean expressions to alter the flow of control in statements
- A boolean expression may also be evaluated for its value such as

```
x = a<b && c<d
```



```
        ifFalse a < b goto L1
        ifFalse c > d goto L1
        t = true
        goto L2
L1:     t = false
L2:     x = t
```

```
if ( a < b && c < d )
    x=true;
else x=false;
```

Self Study (6.6.6)

Boolean Values and Jumping Code

- A clean way of handling both roles of boolean expressions is to first build a syntax tree for expressions, using either of the following approaches
  - Use two passes
    - Construct a complete syntax tree for the input, and
    - then traverse in depth-first order computing the translations specified by the semantic rule
  - Use one pass for statements, but two passes for expressions
    - In the approach, we would first translate *E* in ***while*** *(E) S1* before *S1* is examined.
    - The translation of *E*, however would be done by building its syntax tree and then walking the tree
- Use separate code generation functions for expressions
  - One for generating jumping code
  - The other for computing the value

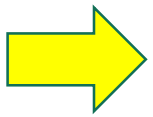
Backpatching

- A key problem when generating code for boolean expression and flow-of-control statement is that of matching a jump instruction with the target of the jump
- The SDD we have used earlier generates symbolic labels
- A separate pass is needed to bind labels to address

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

1st Pass

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```



2nd Pass

```
100:  if x < 100 goto 106
101:  goto 102
102:  if x > 200 goto 104
103:  goto 107
104:  if x != y goto 106
105:  goto 107
106:  x = 0
107:
```

Backpatching

- Backpatching is used to generate code for boolean expressions and flow-of-control statements in one pass
- In Backpatching, list of jumps are passed as synthesized attributes
- Specifically, when a jump is generated, the target of the jump label is left unspecified
- Each such jump is put on a list of jumps whose labels are to be filled in when proper label can be determined
- All of the jumps on a list have the same target label

Code Generation Using Backpatching

- Two synthesized attributes for boolean expression *B*
  - *B. truelist* – a list of jumps or conditional jumps into which we must insert the label to which control goes if *B* is true
  - *B. falselist* – a list of jumps or conditional jumps into which we must insert the label to which control goes if *B* is false
- A synthesized attribute for statement *S*
  - *S. nextlist* – a list of jumps to the instruction immediately following the code for *S*
- Three functions
  - *makelist(i)* – creates a new list containing only *i*, an index into the array of instructions. Returns a pointer to the newly created array
  - *merge(p1, p2)* – concatenates the lists pointed to by *p1* and *p2*, and returns a pointer to the concatenated list
  - *backpatch(p, i)* – inserts *i* as the target label for each of the instructions on the list pointed to by *p*

Code Generation Using Backpatching

SDT for Flow-of-Control and Booleans (on the fly)	SDT for Backpatching

$L \rightarrow$ { $L_1.next = newlabel()$; }
L_1 {$S.next = L.next$; $label(L_1.next)$; }
S

$L \rightarrow$ { $S.next = L.next$; }
S

$S \rightarrow$ **if** ({ $B.true = newlabel()$; $B.false = S_1.next = S.next$; }
B) { $label(B.true)$;}
S_1

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

$B \rightarrow E_1$ **rel** E_2 { gen('if' $E_1.addr$ **rel** $E_2.addr$ ' goto $B.true$');
gen(' goto $B.false$'); }

Backpatching for Boolean Expression

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

2) $B \rightarrow B_1 \;\&\&\; M\; B_2$

3) $B \rightarrow\; !\, B_1$

4) $B \rightarrow (\; B_1\;)$

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

6) $B \rightarrow$ **true**

7) $B \rightarrow$ **false**

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

Backpatching for Flow-of-Control Statements

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

2) $S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$

3) $S \rightarrow \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1$

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

7) $N \rightarrow \epsilon$

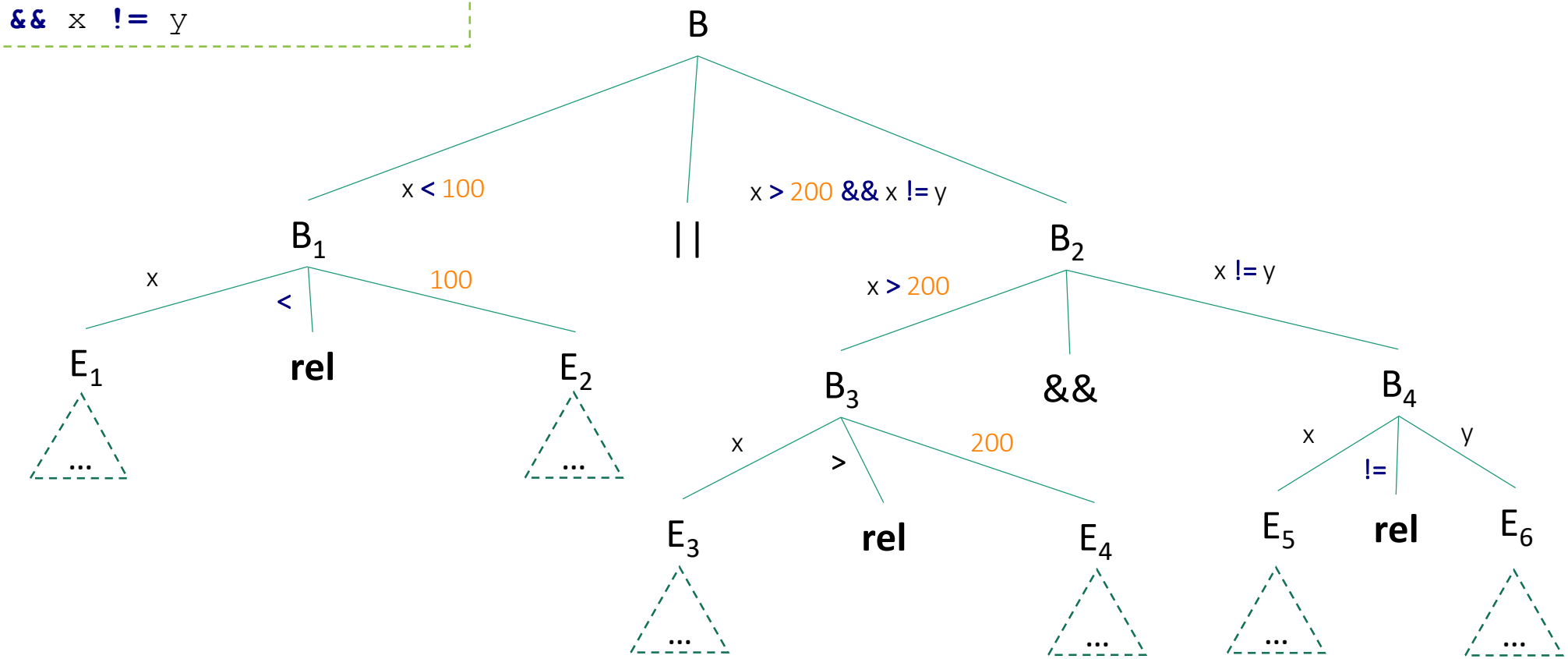
Backpatching for Flow-of-Control Statements

4)	$S \rightarrow \{\ L\ \}$	
5)	$S \rightarrow A\ ;$	
8)	$L \rightarrow L_1\ M\ S$	$\{\ backpatch(L_1.nextlist,\ M.instr);$ $L.nextlist\ =\ S.nextlist;\ \}$
9)	$L \rightarrow S$	$\{\ L.nextlist\ =\ S.nextlist;\ \}$

Translation of a boolean expression using backpatching

```
x < 100 || x > 200 && x != y
```

Prod
$B \rightarrow B_1$ && B_2
$B \rightarrow B_1$ || B_2
$B \rightarrow E_1$ **rel** E_2



Translation of a boolean expression using backpatching

`x < 100 || x > 200 && x != y`

Prod
$B \rightarrow B_1$ && M B_2
$B \rightarrow B_1$ || M B_2
$B \rightarrow E_1$ **rel** E_2
$M \rightarrow \varepsilon$

nextinstr = 106

Instr. #	Instructions
102	
104	

B.truelist={100, 104}
B.falselist={103,105}

B

B.falselist= *B2.falselist*
B.truelist=*merge*(*B1.truelist, B2.truelist*)
backpatch(*B1.falselist, M.instr*)

B1.truelist={100}
B1.falselist={101}

B1.truelist=*makelist*(*nextinstr*)
B1.falselist=*makelist*(*nextinstr*+1)
B1.Code:
if E1.addr ***rel*** *E2.addr*
go to __
go to __

B_1 || M ε *M.instr*= *nextinstr* → 102

B2.truelist={104}
B2.falselist={103,105}

B2.truelist= *B4.truelist*
B2.falselist=*merge*(*B3.falselist, B4.falselist*)
backpatch(*B3.truelist, M.instr*)

B_2

x < 100

E1.Addr: x E_1 ...

rel

E2.Addr: t1 E_2 ...

B3.truelist={102}
B3.falselist={103}

B3.truelist=*makelist*(*nextinstr*)
B3.falselist=*makelist*(*nextinstr*+1)
B3.Code:
if E3.addr ***rel*** *E4.addr*
go to __
go to __

B_3 && M ε *M.instr*= *nextinstr* → *104*

x > 200

E3.Addr: x E_3 ...

rel

E4.Addr: t2 E_4 ...

B4.truelist={104}
B4.falselist={105}

B4.truelist=*makelist*(*nextinstr*)
B4.falselist=*makelist*(*nextinstr*+1)
B4.Code:
if E5.addr ***rel*** *E6.addr*
go to __
go to __

B_4

x != y

E1.Addr: x E_5 ...

rel

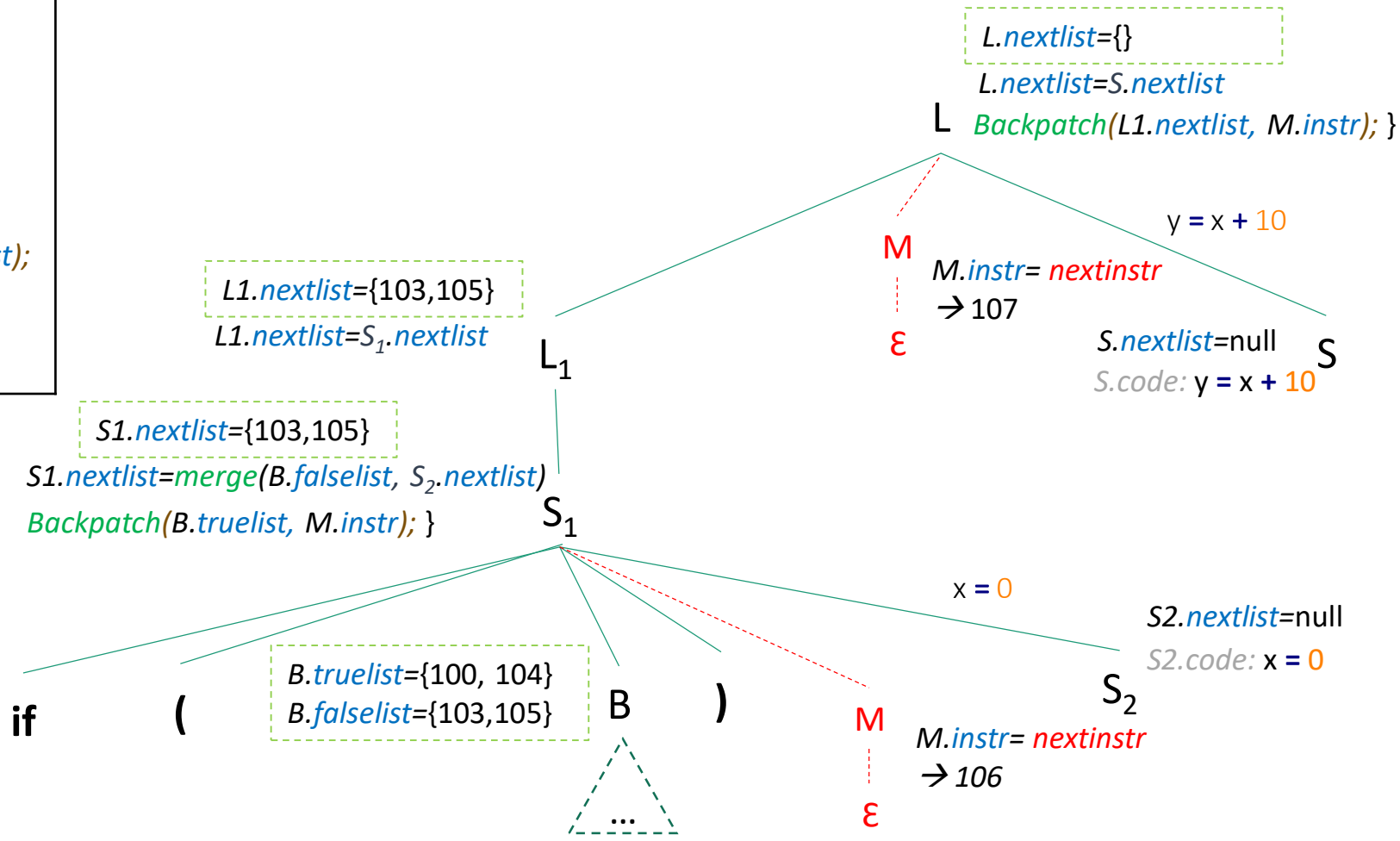
E6.Addr: y E_6 ...

```
if ( x < 100 || x > 200 && x != y )
    x = 0;
y = x + 10;
```

$L \rightarrow L_1\ M\ S$	{ *Backpatch*(L_1 *nextlist*, *M.instr*); } *L.nextlist*=*S nextlist*; }
$L \rightarrow S$	{ *L.nextlist*=*S.nextlist*; }
$S \rightarrow A$	{ *S.nextlist*=*null*}
$S \rightarrow$ **if** (B) $M\ S_1$	{ *S.nextlist*=*merge*(*B.falselist*, S_1*.nextlist*); *Backpatch*(*B.truelist*, *M.instr*); }
$M \rightarrow \varepsilon$	{ *M.instr*=*nextinstr*; }

nextinstr = 108

Instr. #	Instructions
100	*if x < 100 go to* 106
101	*go to* 102
102	*if x > 200 go to* 104
103	*go to* 107
104	*if x != y go to* 106
105	*go to* 107



Self Study (6.7.4)

Break- and Continue- Statements

```
1)  for ( ; ; readch() ) {
2)      if( peek == ' ' || peek == '\t' ) continue;
3)      else if( peek == '\n' ) line = line + 1;
4)      else break;
5)  }
```

- If *S* is the enclosing loop construct, then a break-statement is a jump to the first instruction after the code for *S*
- Code for break can be generated by
  (1) Keeping track of the enclosing loop statement *S*
  (2) generating an unfilled jump for the break-statement, and
  (3) putting this unfilled jump on *S.nextlist*
- In a two-pass front end that builds syntax trees, *S.nextlist* can be implemented as a field in the node for S
- We can keep track of S by using the symbol table to map a special identifier break to the node for the enclosing loop statement

Self Study (6.7.4)

Break-, Continue-, and Goto-Statements

- Alternatively, a pointer to the *S.nextlist* can be put in the symbol table.
- When a break-statement is reached,
  - a unfilled jump is generated,
  - the *nextlist* is looked up in the symbol table,
  - the jump added to the list, and
  - finally, the list is backpatched
- Continue-statement can be handled in a manner analogous to break-statement.
  - However, the target of the generated jump is different.

Self Study (6.8)

Switch- Statements

- The ‘switch’ or ‘case’ statement is available in a variety of languages
- Syntax:

Switch Syntax

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ...
        case Vn−1: Sn−1
        default: Sn
}
```

E: Selector Expression

V1, V2, … , Vn-1: constant values

default: any value other than V1, V2, … , Vn-1

S1, S2, …. , Sn: Statements

- Translation of Switch-Statements
  1. Evaluate the expression of E
  2. Find the value of Vj in the list of cases that is the same as the value of the expression. If not match found, match to the default values
  3. Execute the statement Sj associated with the value found

Self Study (6.8)

Translation of Switch- Statements

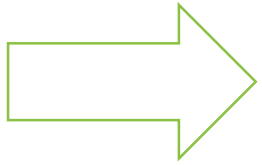
- Step (2) is a n-way branch that can be implemented in several ways
- If the number of cases is small, use a sequence of conditional jumps, each of which tests for an individual value and transfer to the code for the corresponding statement.
- A more compact way is to create a table of pairs, each pair consisting of a value and a label for the corresponding statement's code
  - The value for the expression E is paired with the label for the default statement and placed at the last of the table
  - A loop generated by the compiler compares the value of the expression to the values in the table
  - If the number of values is high, use a hash table
  - If the values all lies in some small range and the number of different values used is a reasonable fraction of the number of values in the range, then use 'buckets'

Self Study (6.8)

SDT for Translation of Switch- Statements

Switch Syntax

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ...
        case Vn−1: Sn−1
        default: Sn
}
```



```
                code to evaluate E into t
                if t != V1 goto L1
                code for S1
                goto next
L1:             if t != V2 goto L2
                code for S2
                goto next
L2:
                ...
Ln−2:           if t != Vn−1 goto Ln−1
                code for Sn−1
                goto next
Ln−1:           code for Sn
next:
```

Three Address Code (V.1)

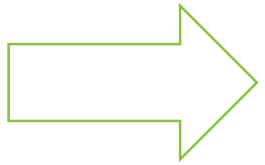
- Requires the compiler to do extensive analysis to find the most efficient implementation
- It is inconvenient in a one-pass compiler to place the branching statements at the beginning, because the compiler could not then emit code for each of statements *Si* as it saw them

Self Study (6.8)

SDT for Translation of Switch- Statements

Switch Syntax

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ...
        case Vn−1: Sn−1
        default: Sn
}
```



```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln−1:   code for Sn−1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn−1 goto Ln−1
        goto Ln
next:
```

1. When the keyword ‘switch’ is found, generate two labels test and next and a new temporary t
2. Then while E is parsed, generate code to evaluate E into t
3. After processing of E, generate the jump goto test
4. Then for each case keyword, create a new label *Li* and enter in a queue in the symbol table value-label pair <*Vi*, *Li*>
5. Process each statement case *Vi*:*Si* by emitting the label *Li* attached to the code of *Si*, followed by the jump to goto next
6. When the end of the switch is found, generate the code for the n-way branch by reading from the queue of value-label pairs in the symbol table

Reference

- Compilers: Principles, Techniques, & Tools, by
  Alfred B. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

  - Chapter 6
    - 6.1
    - 6.2
    - 6.3
    - 6.4
    - 6.6
    - 6.7
    - 6.8 (Self study)

Thank You!