Lecture 11: Dependency Graphs and SDD Types

- Dependency Graphs
- S-attributed definitions
- L-attributed definitions

section 5.2 chapter 5 of textbook.

SDD: Syntax Directed Definition

SDD = Grammar + Semantic Rules

- Attributes are associated with grammar symbols and semantic rules are associated with productions.
- Attributes may be numbers, strings, reference datatype, etc.

Dependency Graphs:

A dependency graph depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second.

Example: Consider the following production and rule.

Production: $E \to E_1 + T$ Semantic Rule: $E.val = E_1.val + T.val$

At every node N labeled E, with children corresponding to the body of this production, the synthesized attribute ual at N is computed using the values of ual at the two children, labeled E and T. Thus, a portion of the dependency graph for every parse tree in which this production is used looks like Fig. 5.6. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

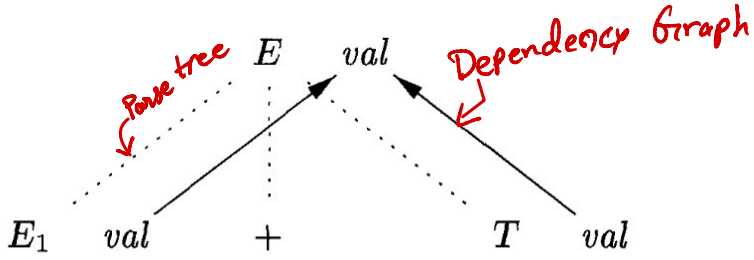


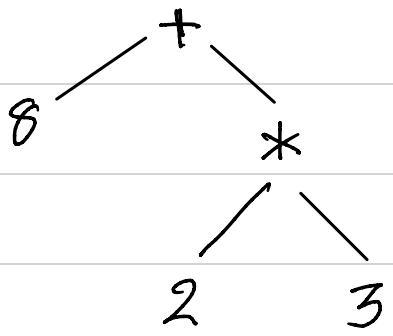
Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

Grammar Production Rule	Semantic Rule (action)
$E \rightarrow EN$	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val = E_1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$ ← non-terminal	$F.val = digit.lexval$
$N \rightarrow ;$	; terminal symbol

- For non-terminals, we use **lexval** in semantic rule.

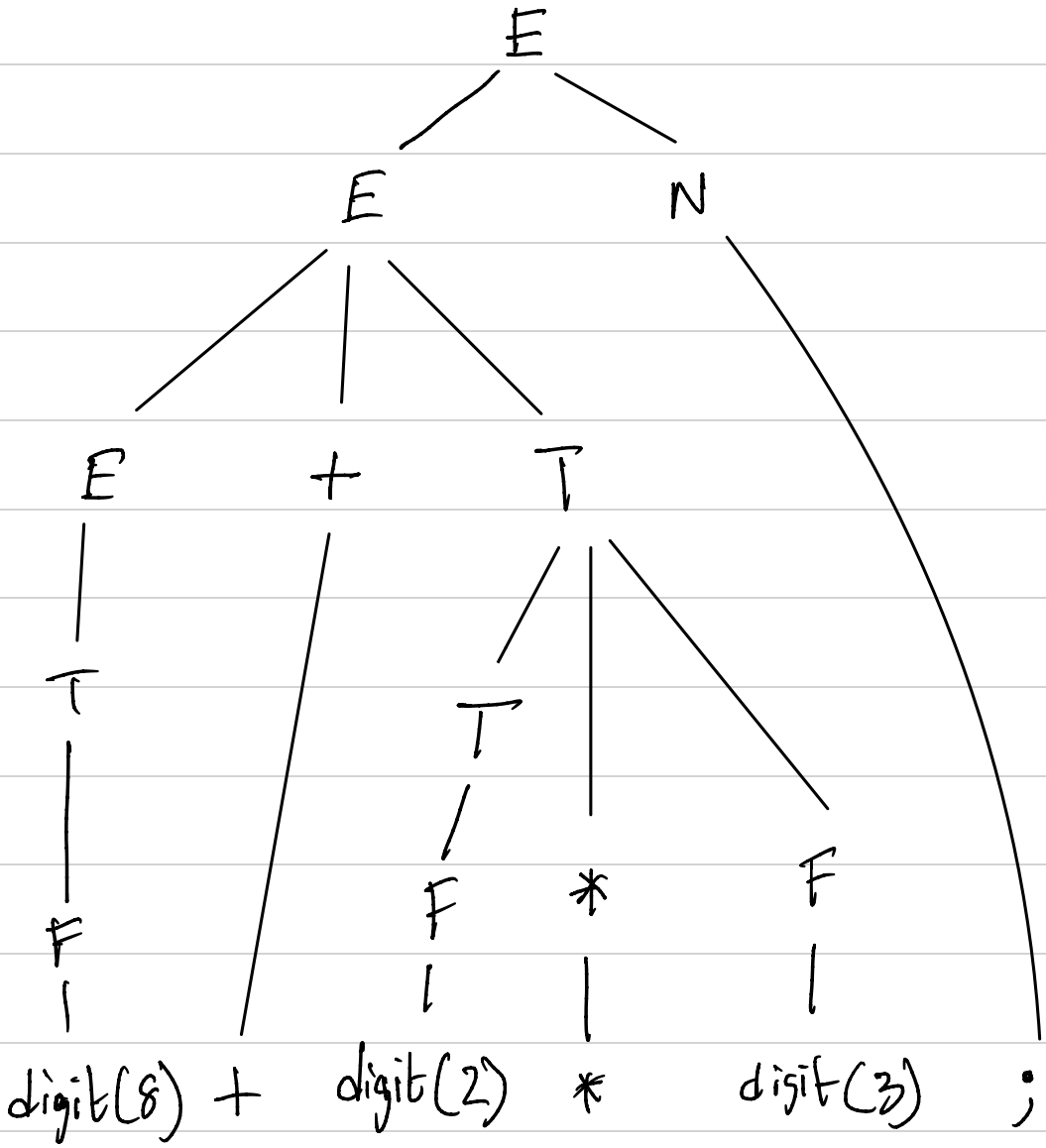
Example: Expression: $8 + 2 * 3$

Syntax Tree:



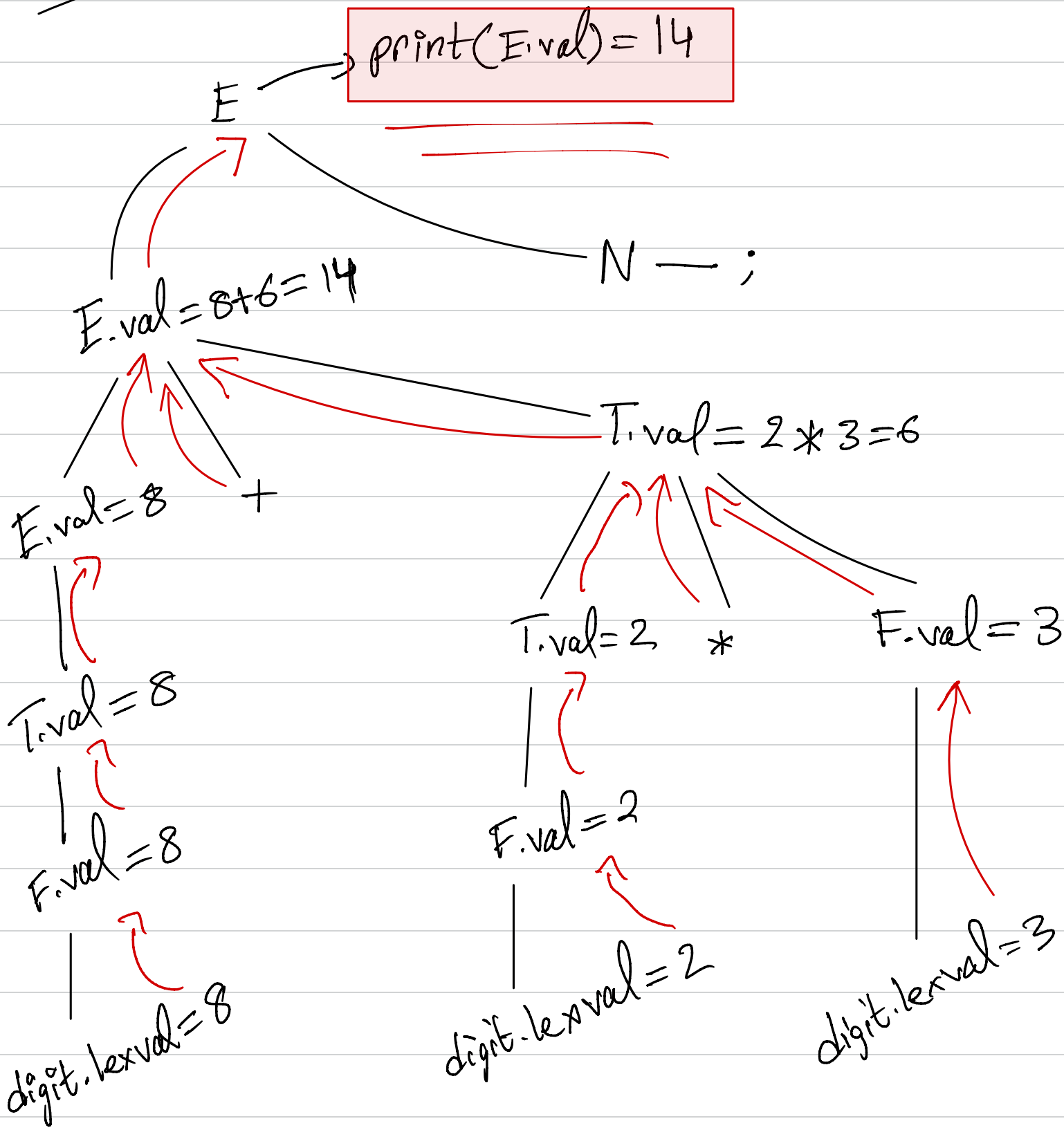
8+2*3

Parse Tree:



Annotated Parse Tree

Use bottom-up approach here



Another Example

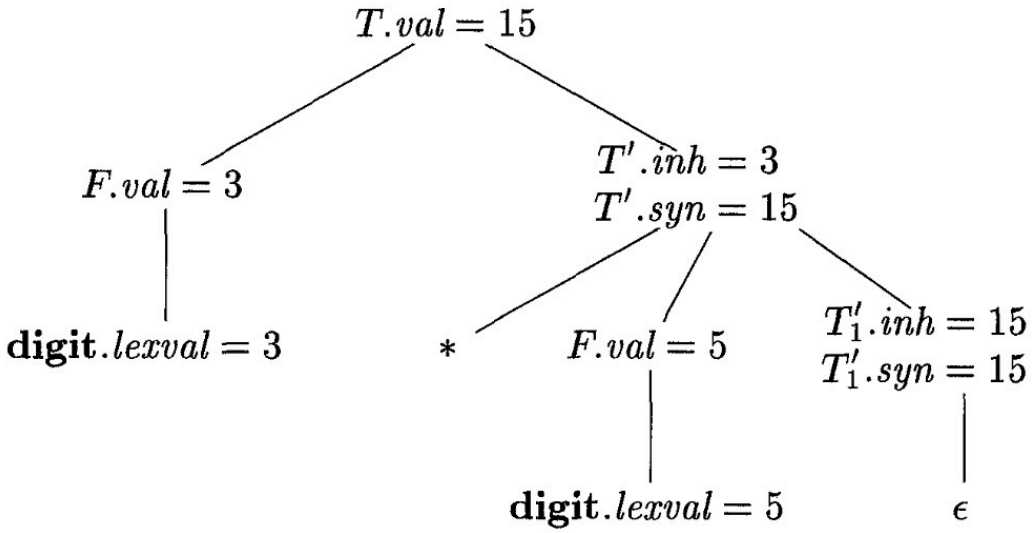


Figure 5.5: Annotated parse tree for $3 * 5$

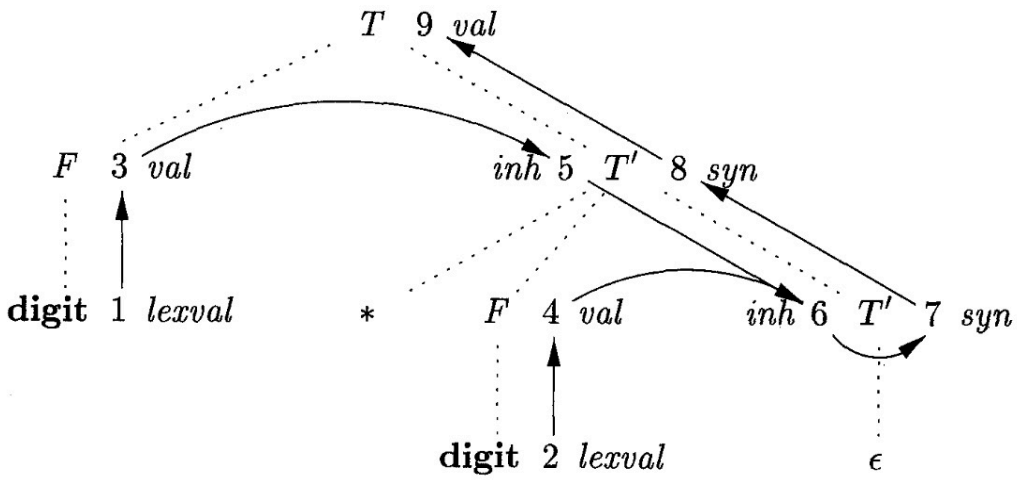


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

5.2.3 | S-attributed Definitions

- An SDD is S-attributed if every attribute is synthesized.

- An SDD that uses only synthesized attribute is called S-attributed SDD.

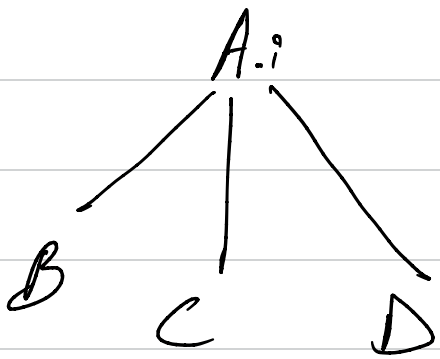
Example:

$A \rightarrow BCD$

$A.i = B.i$

$A.i = C.i$

$A.i = D.i$



- Semantic actions are always placed at right end of the production. (Postfix SDD)

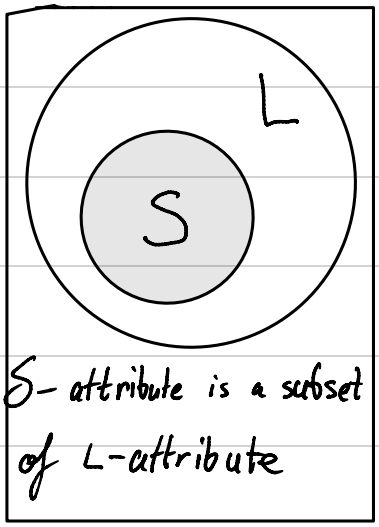
- Attributes are evaluated with Bottom Up Parsing

- Basically, parent value is being calculated in terms of its children.

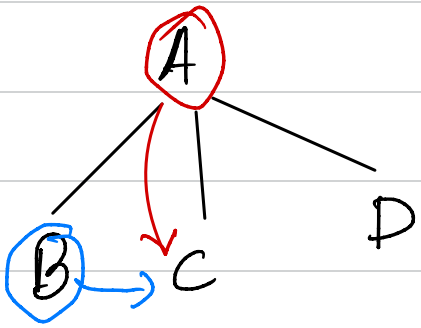
- So, to calculate value of A, we need to calculate B, C, D first!

5.2.4 | L-Attributed SDD

- An SDD that uses both synthesized and inherited attributes is called as L-attributed SDD.
  But, each inherited attribute is restricted to inherit from parent or left siblings only.



E: $A \rightarrow BCD$



To calculate value of C, we can inherit from C's left sibling or it's parent, but never its right sibling.

$C.i = B.i$

$C.i = A.i$

but, $C.i \neq D.i$

↳ because D is right sibling of C

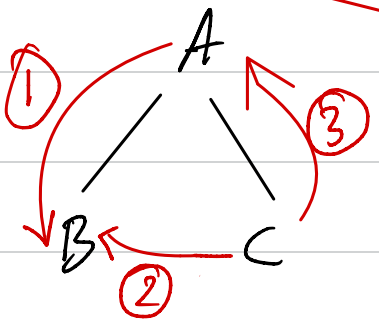
- Semantic actions are placed anywhere on the R.H.S of the production
- Attributes are evaluated by traversing parse tree depth first, left to right.

Identify whether they are S-Attributed SDD or L-attributed SDD

1) $A \rightarrow BC \quad \{B.i = A.i, \; C.i = B.i, \; A.i = C.i\}$ (① ② ③)

inherited inherited synthesized

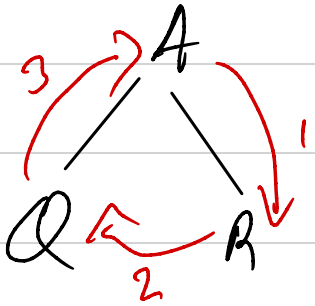
Ans-



Therfore, this is L-attributed SDD.

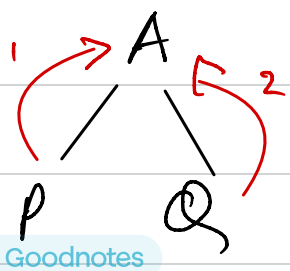
2) $A \rightarrow QR \quad \{R.i = A.i, \; Q.i = R.i, \; A.i = Q.i\}$ (① ② ③)

↳ Inheritance from right sibling is not allowed!



∴ This is neither S-attributed or L-Attributed SDD

3) $A \rightarrow PQ \quad \{A.i = P.i, \; A.i = Q.i\}$ (① ②)



This is both S-attributed and L-attributed SDD

Quiz-4 Syllabus:

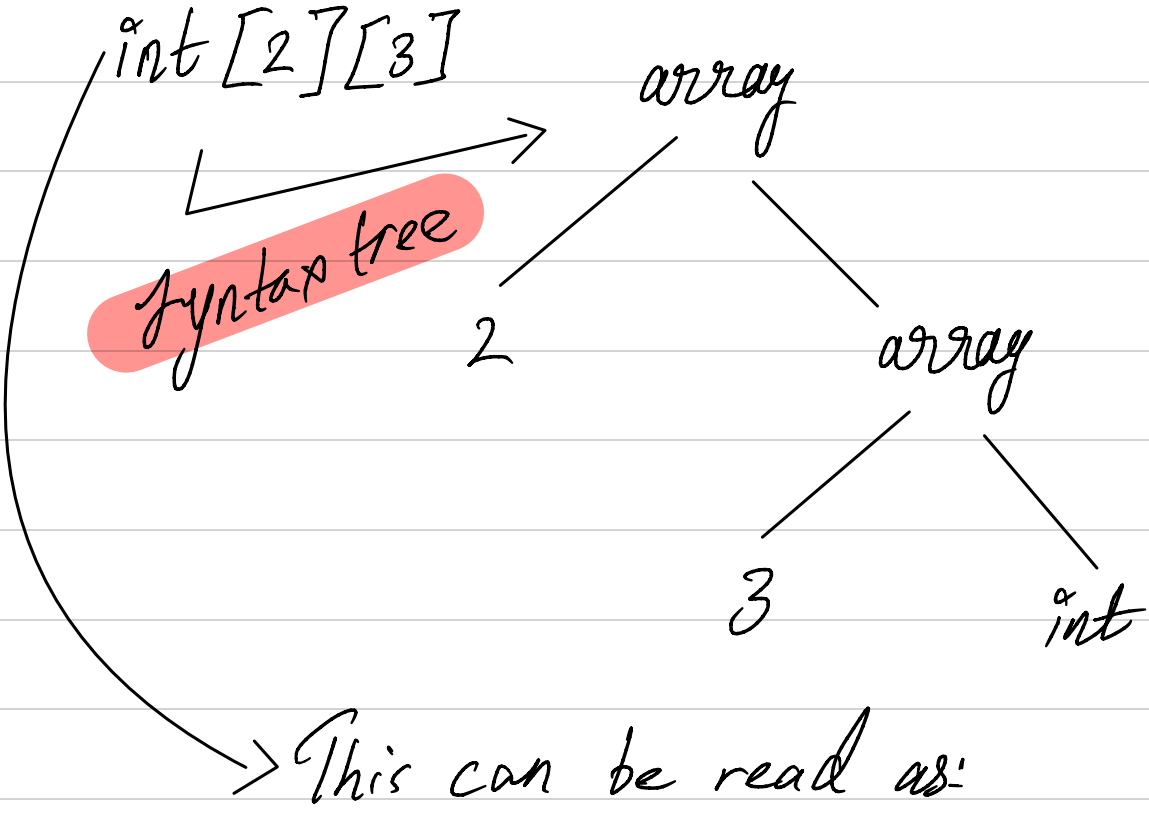
Lecture 13 – Lecture 16

Lecture 12: Type Grammar and Attributes of a Type

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

With the SDD in Fig. 5.16, nonterminal T generates either a basic type or an array type. Nonterminal B generates one of the basic types int and float. Tgenerates a basic type when T derives BC and C derives E. Otherwise, C generates array components consisting of a sequence of integers, each integer surrounded by brackets.



This can be read as:

"array of 2 arrays of 3 integers."

6.3 | Types and Declarations.

The applications of types can be grouped under checking and translation:

> Type checking uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator. For example, the && operator in Java expects its two operands to be booleans; the result is also of type boolean.

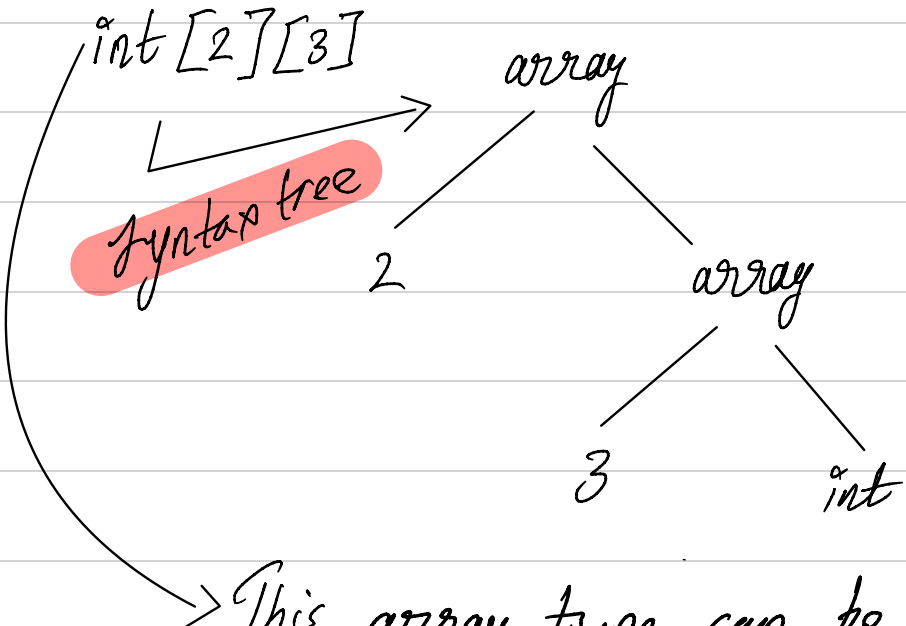
> Translation Applications. From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator, among other things.

6.3.1 Type expressions

Types have structure, which we shall represent using type expressions

A type expression is either basic type or is formed by applying an operator called a type constructor or type expression

Example 2.8



This array type can be read as:

"array of 2 arrays of 3 integers."

and written as type expression

array(2, array(3, integer)).

The operator array takes two parameters, a number and a type.

We shall use the following definition of type expressions:

- A basic type is a type expression. Typical basic types for a language include boolean, char, integer, float, and void → this denotes "the absence of a value."

- A type name is a type expression

- 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 expression 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. We write 's→t' for "function from type s to type t."

- If s and t are type expressions, then their cartesian product $s \times t$ is a type expression. Products are introduced for completeness; they can be used to represent a list or type of tuples (ex: for function parameters). We assume that $\times$ associates to the left and that it has higher precedence than $\rightarrow$.

- Type expressions may contain variables whose values are type expressions.

6.3.2 | Type Equivalence

- When are two type expressions equivalent?

→ When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions is true:

→ 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.

6.3.3 | Declarations

Suppose a grammar:

$D \rightarrow T\ id\ ;\ D \mid \epsilon$

$T \rightarrow B\ C \mid record\ '\{'\ D\ '\}'$

$B \rightarrow int \mid float$

$C \rightarrow \epsilon \mid [\ num\ ]\ C$

Non-terminal D generates a sequence of declarations. Nonterminal T generates basic, array, or record types. Nonterminal B generates one of the basic types int or float. Nonterminal C, for "component", generates strings of zero or more integers, each integer surrounded by brackets. An array type consists of a basic type specified by B,

followed by array components specified by nonterminal C. A record type (2nd production by T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

Array Width and Offset Calculation

↳ Lecture 14

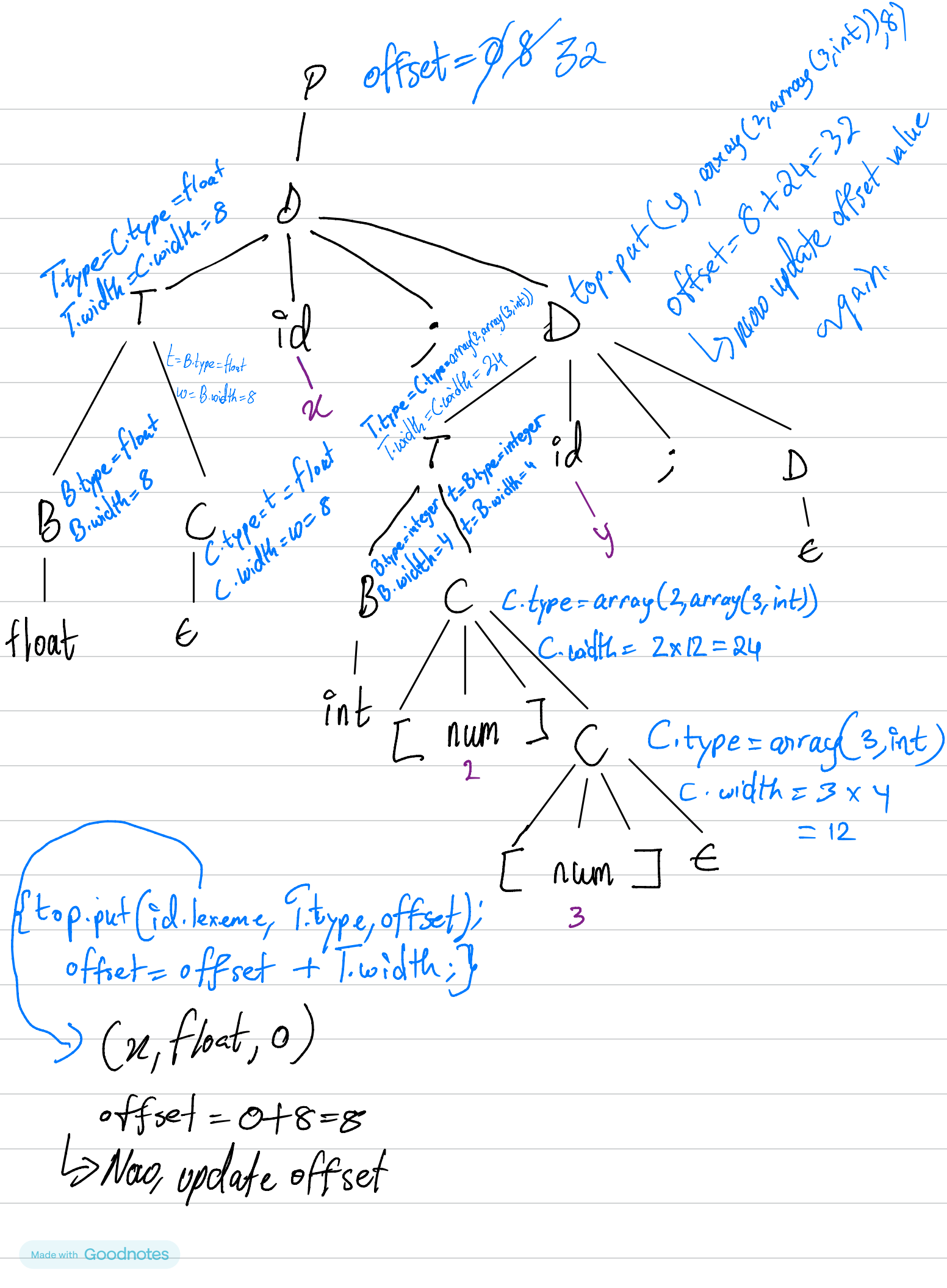
Lecture 13: Type information Processing During Bottom-Up parsing

Reading references same as lecture 13

$P \rightarrow$ D		$\{ offset = 0; \}$
$D \rightarrow T$ **id** ; D_1		$\{ top.put(\textbf{id}.lexeme, T.type, offset);$ $offset = offset + T.width; \}$
$D \rightarrow \epsilon$		
$T \rightarrow B$ C		$\{ t = B.type; w = B.width; \}$ $T.type = C.type; T.width = C.width$
$B \rightarrow$ **int**		$\{ B.type = integer; B.width = 4; \}$
$B \rightarrow$ **float**		$\{ B.type = float; B.width = 8; \}$
$C \rightarrow \epsilon$		$\{ C.type = t; C.width = w; \}$
$C \rightarrow$ [**num**] C_1		$\{ array(\textbf{num}.value, C_1.type);$ $C.width = \textbf{num}.value \times C_1.width; \}$

Calculate the width for input : float x;
int[2][3] y;

① Draw the parse tree.
② Find type and width using bottom up approach
③ Update offset.



Lecture 14 Introduction to intermediate code generation.

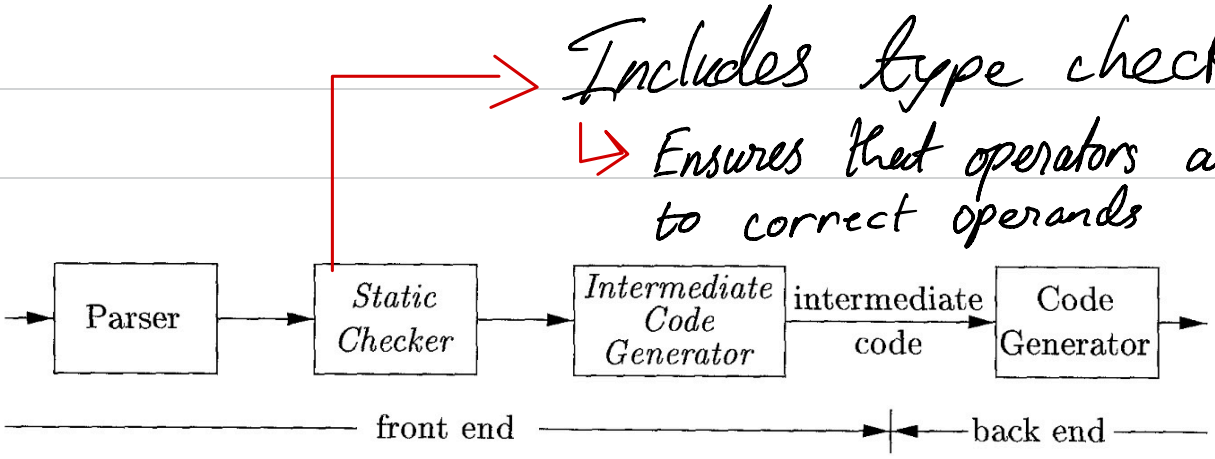


Figure 6.1: Logical structure of a compiler front end

6.1 Variants of Syntax Tree

DAG: Directed Acrylic Graph identifies the common subexpressions of the expression.

Priority

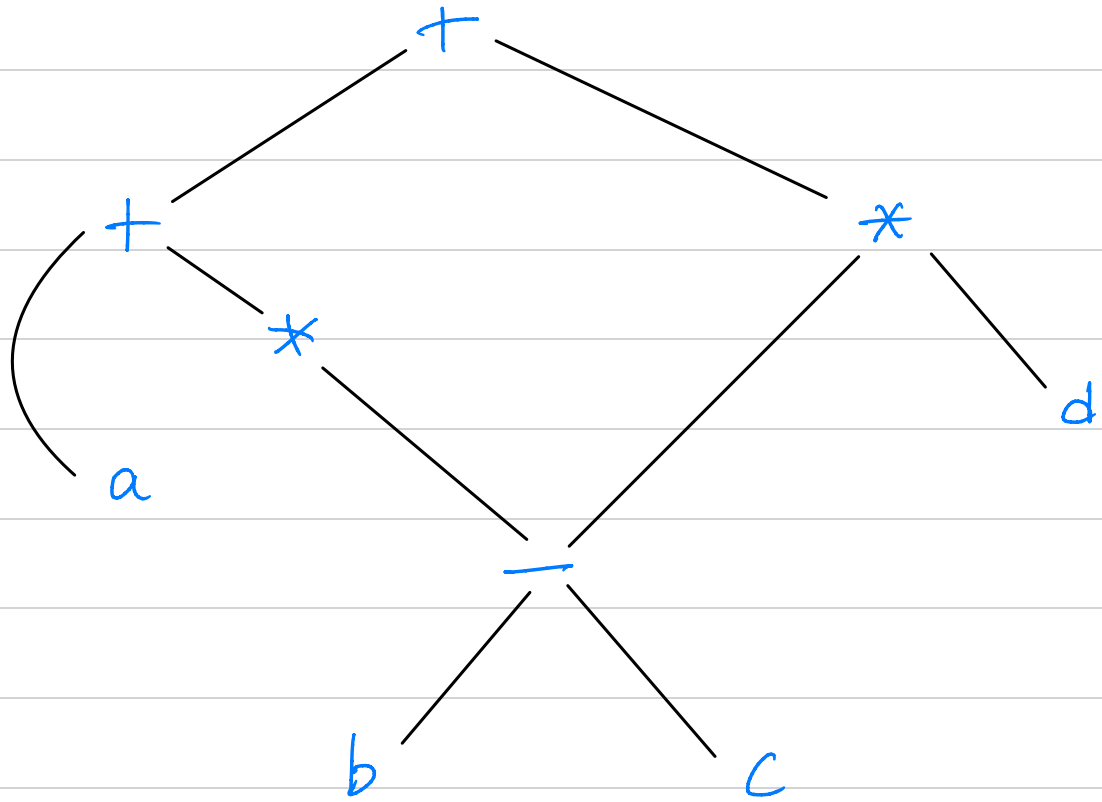
\-
*
\+

DAG's can be constructed by using the same techniques that construct syntax trees.

6.1.1 DAGs for Expressions

ex: a + a * (b − c) + (b − c) * d

DAG:



DAG - Example (Youtube Video)

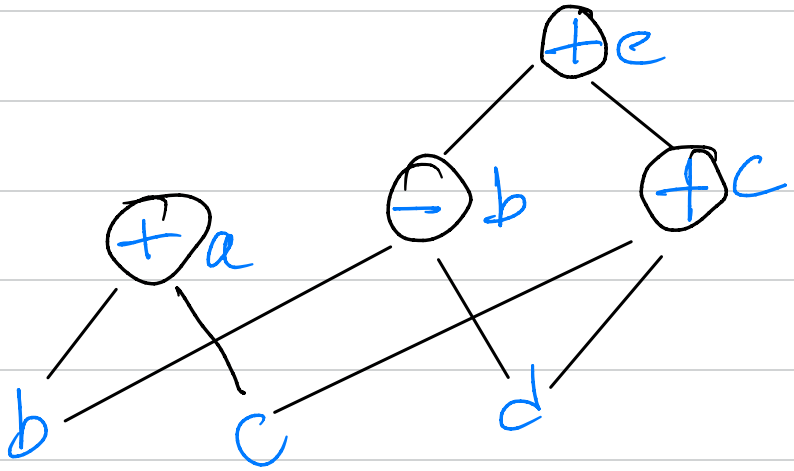
Ex:

① $a = b + c$

$b = b - d$

$c = c + d$

$e = b + c$



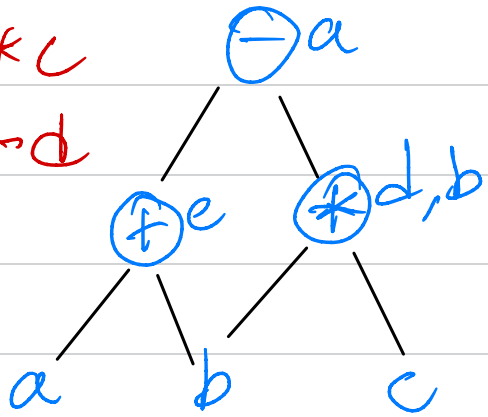
Ex 2:

$d = b * c$

$e = a + b$

$b = b * c$

$a = e - d$

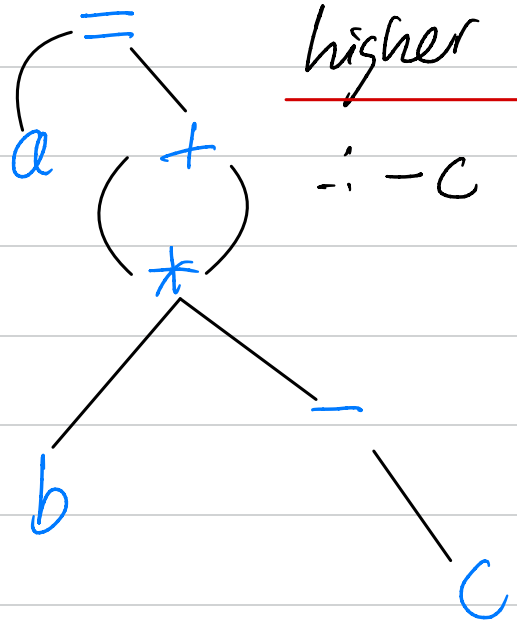


Ex 3:

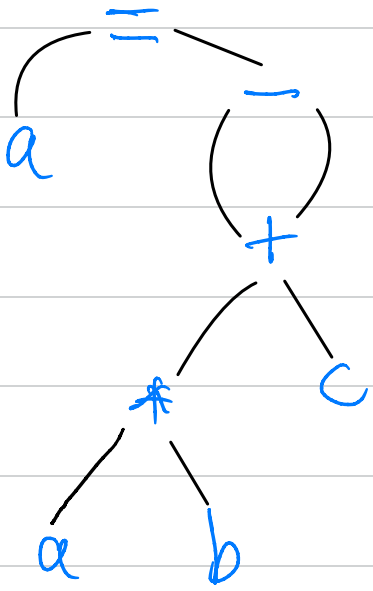
$a = b * -c + b * -c$

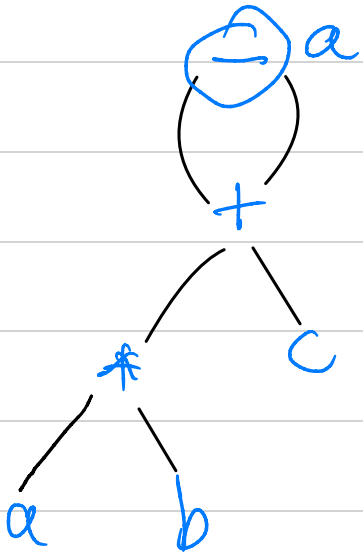
— operator is unary operator, and has higher priority than * or +

∴ −c operation takes place first!



Ex 4: $a = (a * b + c) - (a * b + c)$





Ex 5: $(a+b)*(a+b+c)$

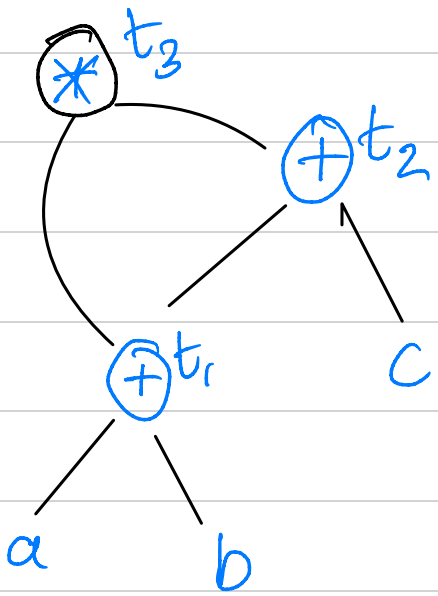
We can also solve this in another way using Three Address Code Generation or TAC.

TAC ⇒

$t_1 = a+b$

$t_2 = t_1+c$

$t_3 = t_1 * t_2$



Three Address Code (TAC)

→ It is an intermediate code. It is used by the optimizing compiler.

→ In TAC, there must be at most one operator at the right hand side of an instruction.

ex ① $x + y * 2$

$t_1 = y * 2$

$t_2 = x + t_1$

* The temp variables are generated by the compiler!

ex ② $a + a * (b + c) + d * (b - c)$

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = d * t_1$

$t_4 = t_2 + t_3$

$t_5 = a + t_4$

① Quadruples

4 fields — operator, source 1, source 2, destination or result

ex: $a = -b * c + d$

↳ IAC ⇒

$t_1 = -b$ — ①

$t_2 = c + d$ — ②

$t_3 = t_1 * t_2$ — ③

$a = t_3$ — ④

Now convert this into a quadruple

	operator	source 1	source 2	result
0	uminus	b		t_1
1	+	c	d	t_2
2	*	t_1	t_2	t_3
3	=	t_3		a

② Triples

3 fields — operator, source1, source2

ex: $a = -b * c + d$

→ TAC ⇒

unary minus → (uminus)

$t_1 = -b$ — 0

$t_2 = c + d$ — 1

$t_3 = t_1 * t_2$ — 2

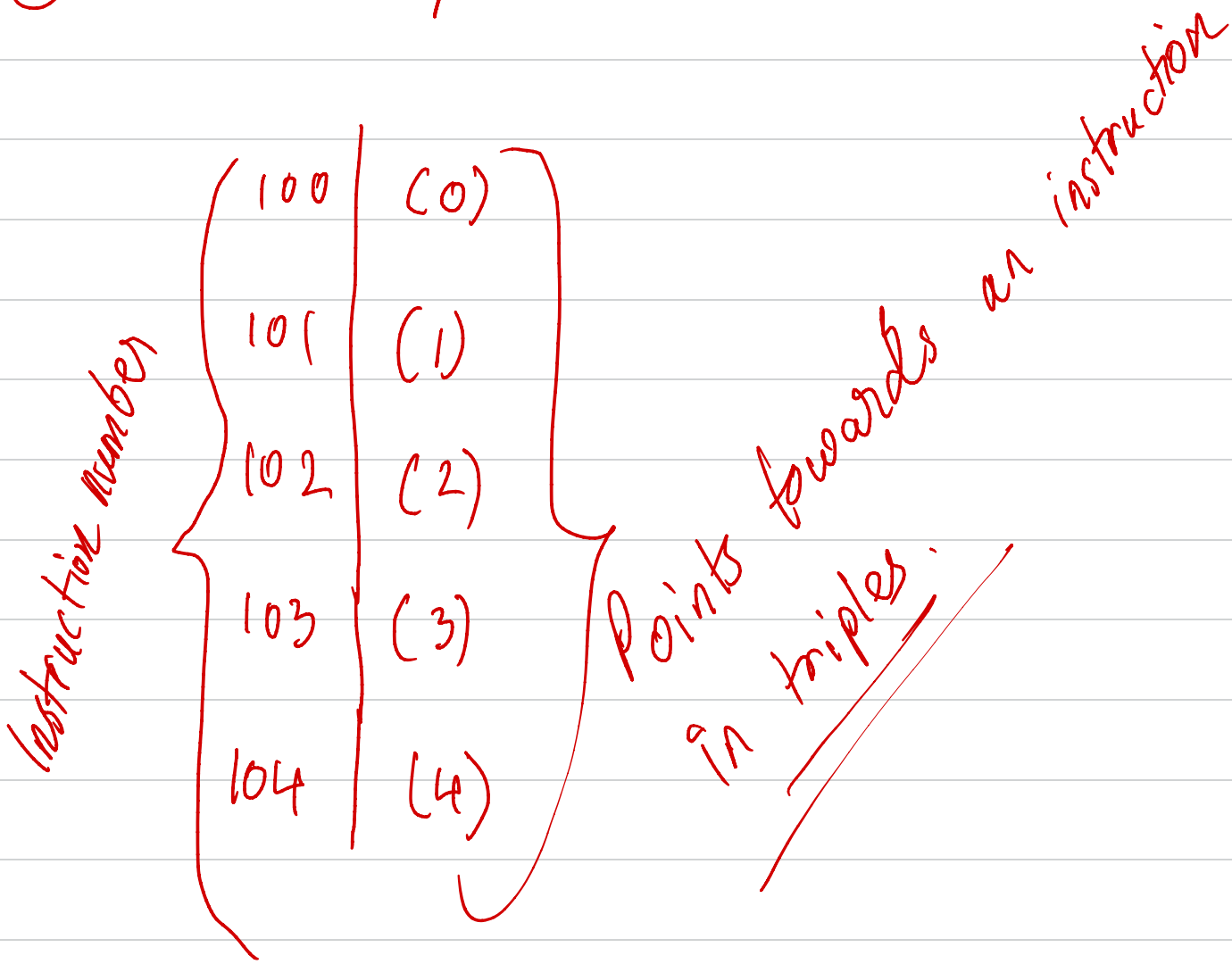
$a = t_3$ — 3

Now convert this into a triple

	operator	source1	source2
0	uminus	b	
1	+	c	d
2	*	(0)	(1)
3	=	(2)	

Instead of t_n, write the number of that production rule.

(3) Indirect Triple



Lecture 15: Array Access logic and SDT for Array Access.

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

|| ← concatenation sign

generates temp variables

Refers to entry of E in symbol table.

Figure 6.19: Three-address code for expressions

The semantic rules for this production define E.addr to point to the symbol-table entry for this instance of id. Let top denote the current symbol table. Function top.get retrieves the entry when it is applied to the string representation id.lexeme of this instance of id. E.code is set to the empty string.

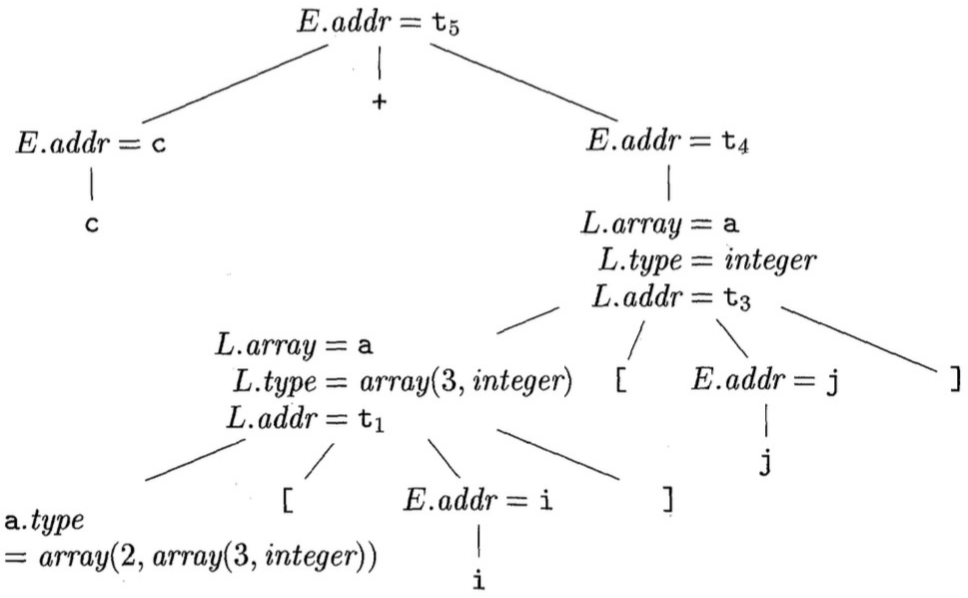


Figure 6.23: Annotated parse tree for `c + a[i][j]`

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

Figure 6.24: Three-address code for expression `c + a[i][j]`

SDT = Syntax Directed Translation

SDD = Syntax Directed Definition

Example 6.11: The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c ;` into the three-address code sequence

```
t1 = minus c
t2 = b + t1
a = t2
```

6.4.2 Incremental Translation

Code attributes can be long strings, so they are usually generated incrementally, as discussed in Section 5.5.2. Thus, instead of building up $E.code$ as in Fig. 6.19, we can arrange to generate only the new three-address instructions, as in the translation scheme of Fig. 6.20. In the incremental approach, *gen* not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far. The sequence may either be retained in memory for further processing, or it may be output incrementally.

The translation scheme in Fig. 6.20 generates the same code as the syntax-directed definition in Fig. 6.19. With the incremental approach, the *code* attribute is not used, since there is a single sequence of instructions that is created by successive calls to *gen*. For example, the semantic rule for $E \rightarrow E_1 + E_2$ in Fig. 6.20 simply calls *gen* to generate an add instruction; the instructions to compute E_1 into $E_1.addr$ and E_2 into $E_2.addr$ have already been generated.

The approach of Fig. 6.20 can also be used to build a syntax tree. The new semantic action for $E \rightarrow E_1 + E_2$ creates a node by using a constructor, as in

$$E \rightarrow E_1 + E_2 \quad \{\ E.addr = \textbf{new}\ Node('+', E_1.addr, E_2.addr);\ \}$$

Here, attribute *addr* represents the address of a node rather than a variable or constant.

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

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

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

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

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

Figure 6.20: Generating three-address code for expressions incrementally

6.4.3 Addressing Array Elements

Array elements can be accessed quickly if they are stored in a block of consecutive locations. In C and Java, array elements are numbered $0, 1, \ldots, n-1$, for an array with n elements. If the width of each array element is w, then the ith element of array A begins in location

$$base + i \times w \tag{6.2}$$

where *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of $A[0]$.

The formula (6.2) generalizes to two or more dimensions. In two dimensions, we write $A[i_1][i_2]$ in C and Java for element i_2 in row i_1. Let w_1 be the width of a row and let w_2 be the width of an element in a row. The relative address of $A[i_1][i_2]$ can then be calculated by the formula

$$base + i_1 \times w_1 + i_2 \times w_2 \tag{6.3}$$

In k dimensions, the formula is

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k \tag{6.4}$$

where w_j, for $1 \leq j \leq k$, is the generalization of w_1 and w_2 in (6.3).

Alternatively, the relative address of an array reference can be calculated in terms of the numbers of elements n_j along dimension j of the array and the width $w = w_k$ of a single element of the array. In two dimensions (i.e., $k = 2$ and $w = w_2$), the location for $A[i_1][i_2]$ is given by

$$base + (i_1 \times n_2 + i_2) \times w \tag{6.5}$$

In k dimensions, the following formula calculates the same address as (6.4):

$$base + ((\cdots(i_1 \times n_2 + i_2) \times n_3 + i_3)\cdots) \times n_k + i_k) \times w \tag{6.6}$$

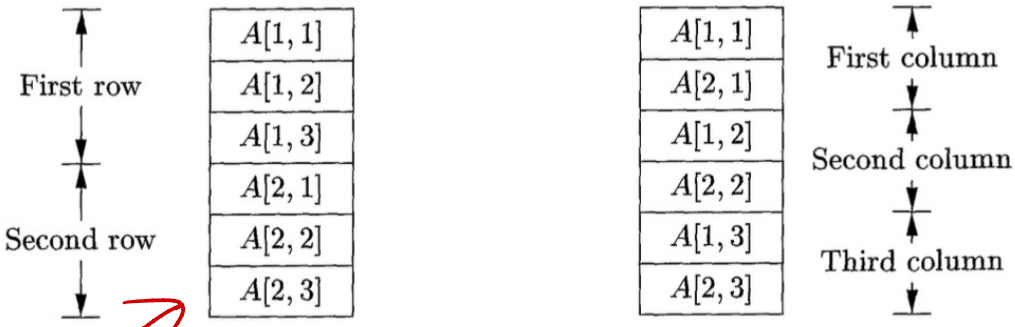
More generally, array elements need not be numbered starting at 0. In a one-dimensional array, the array elements are numbered $low, low + 1, \ldots, high$ and $base$ is the relative address of $A[low]$. Formula (6.2) for the address of $A[i]$ is replaced by:

$$base + (i - low) \times w \tag{6.7}$$

The expressions (6.2) and (6.7) can be both be rewritten as $i \times w + c$, where the subexpression $c = base - low \times w$ can be precalculated at compile time. Note that $c = base$ when low is 0. We assume that c is saved in the symbol table entry for A, so the relative address of $A[i]$ is obtained by simply adding $i \times w$ to c.

Compile-time precalculation can also be applied to address calculations for elements of multidimensional arrays; see Exercise 6.4.5. However, there is one situation where we cannot use compile-time precalculation: when the array's size is dynamic. If we do not know the values of low and $high$ (or their generalizations in many dimensions) at compile time, then we cannot compute constants such as c. Then, formulas like (6.7) must be evaluated as they are written, when the program executes.

The above address calculations are based on row-major layout for arrays, which is used in C and Java. A two-dimensional array is normally stored in one of two forms, either *row-major* (row-by-row) or *column-major* (column-by-column). Figure 6.21 shows the layout of a 2×3 array A in (a) row-major form and (b) column-major form. Column-major form is used in the Fortran family of languages.



(a) Row Major (b) Column Major

Figure 6.21: Layouts for a two-dimensional array.

We will be focusing on row major

Array may not always start from 0, it can start from another value.

We can calculate the address using the formula above.

We can get two dimensional array by generating another parse tree within its parse tree.

we can use the formula to calculate relative address.

$base + i \times w$ [for 1 Dimensional array]

$base + i_1 \times w_1 + i_2 \times w_2$ [for 2 Dimensional array]

If the index does not start from zero:

$base + (i - low) \times w$ (low: starting index.)

Lecture 16-17 | Control Flow

Reading: 6.6 of textbook

Youtube video: Sudhakar

Suppose production rule: $S \rightarrow \text{if}(B)\ \text{then}\ S_1$

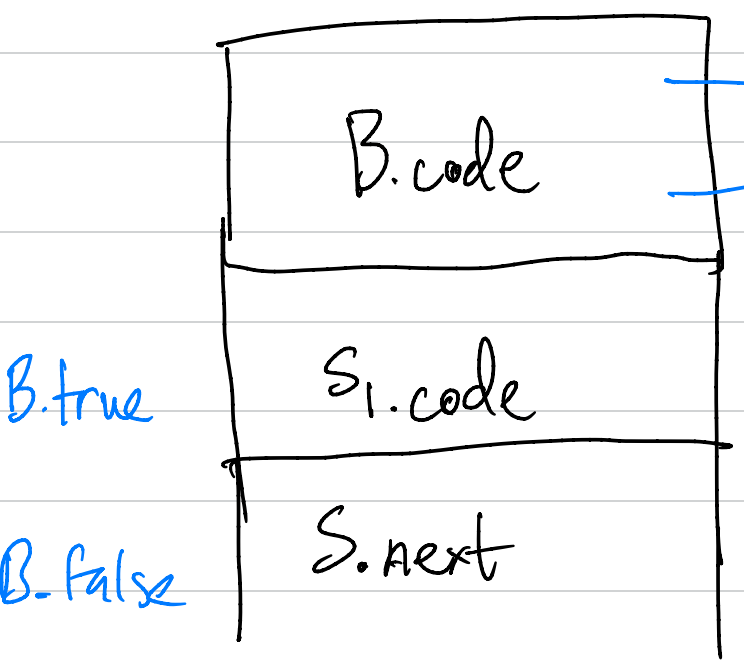
and, suppose the semantic rules for it:

$B.\text{true} = \text{newlabel}()$

$S_1.\text{next} = S.\text{next}$

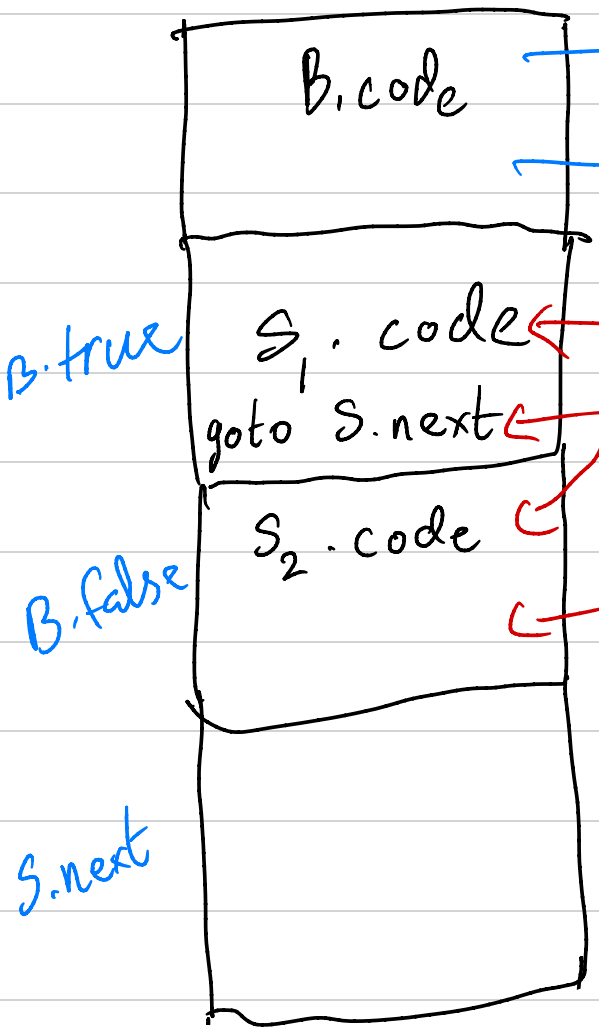
$B.\text{false} = S.\text{next}$

$S.\text{code} = B.\text{code} \,||\, \text{label}(B.\text{true}) \,||\, S_1.\text{code}$



Suppose another production rule:

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



newlabel()

Written to skip B.false

Not written as next block is the desired block.

Now, we can write the semantic rules:

B.true = newlabel()

S_1.next = S.next

B.false = newlabel()

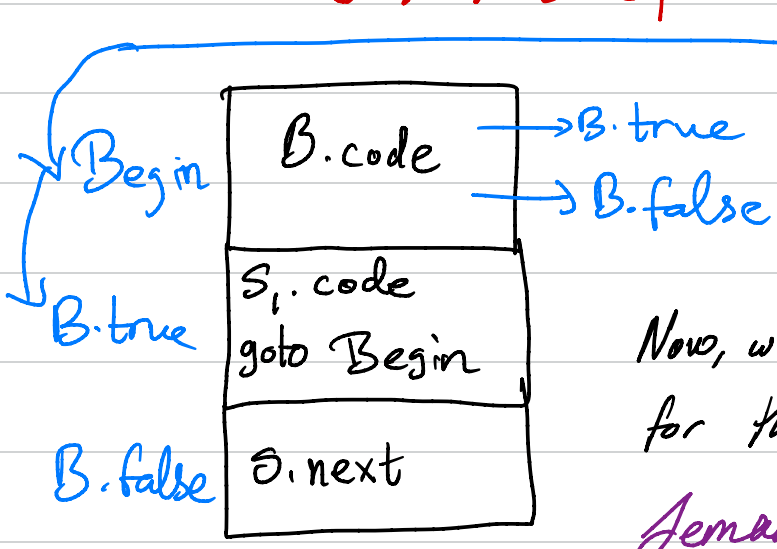
S_2.next = S.next

S.code = B.code || label(B.true) ||
S_1.code || gen('goto' S.next) ||
label(B.false) || S_2.code

For while loops:

Suppose, the production rule:

$S \rightarrow$ while (B) then S_1



Now, we can write the semantic rules for this block of code:

Semantic Rules:

Begin = newlabel()

B.true = newlabel()

S_1.next = Begin

B.false = S.next

S.code = label(Begin) || B.code ||
label(B.true) || S_1.code ||
gen('goto' Begin)

Backpatching 6.7 in book (Lec-18)

Sudhakar

logical or ↙ ↘ logical and

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

First pass

100: if x<100 then 106

101: goto 102

102: if y>200 then 104

103: goto 107

104: if x!=y then 106

105: goto 107

106: true

107: false

Second pass (Backpatching)

Short circuit evaluation

Translation rules for the productions above

① marker non-terminal

$B \rightarrow B_1 \,||\, M\, B_2$

{ Backpatch(B_1.falselist, M.instr);

B.truelist = merge(B_1.truelist, B_2.truelist);

B.falselist = B_2.falselist;

②

$B \rightarrow B_1$ && $M B_2$

{ Backpatch(B_1.truelist, M.instr);

B.truelist = B_2.truelist;

B.falselist = merge(B_1.falselist, B_2.falselist)

③ $B \rightarrow !B_1$

{ B.truelist = B.falselist;

B.falselist = B.truelist; }

④ $B \rightarrow (B_1)$

{ B.truelist = B.truelist;
B.falselist = B.falselist; }

⑤ $M \rightarrow \epsilon$ { m.instr = nextinstr; }

Annotated Parse Tree