Intermediate Code Generation

Contd…

6.4 Translation of Expressions

6.5 Type Checking

6.4 Translation of Expressions

- Translation of expressions in to three-address code
- An array reference like A[i][j] is translated in to a sequence of three-address instructions that appropriately calculates the address for the reference.
  - Array reference, in three address code can be A[i]=j; or x=B[y]; In fact these things are seen as *p=q; r=&s; so on.

- We see translation of **expressions** in to three-address code.
- There are other things Like
  - loops
  - Function calls
  - Etc   (these we see later)

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

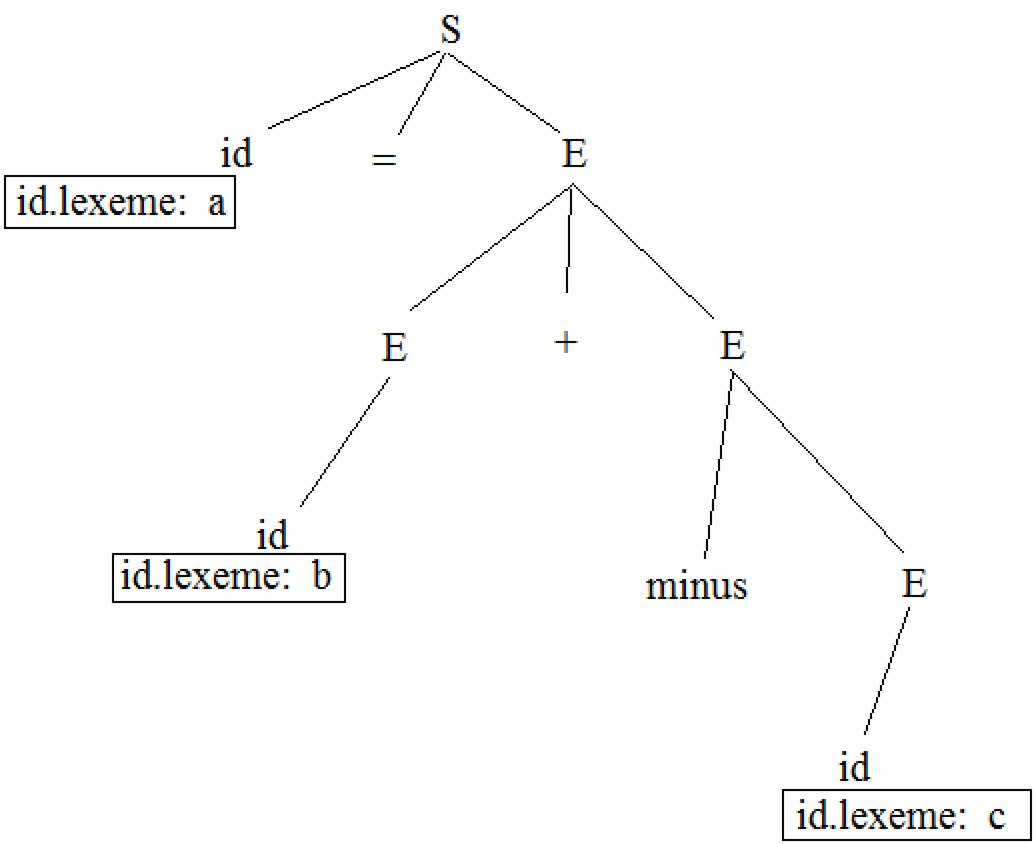
Figure 6.19: Three-address code for expressions

Attributes: *code* for S; *addr* and *code* for an Expression E.

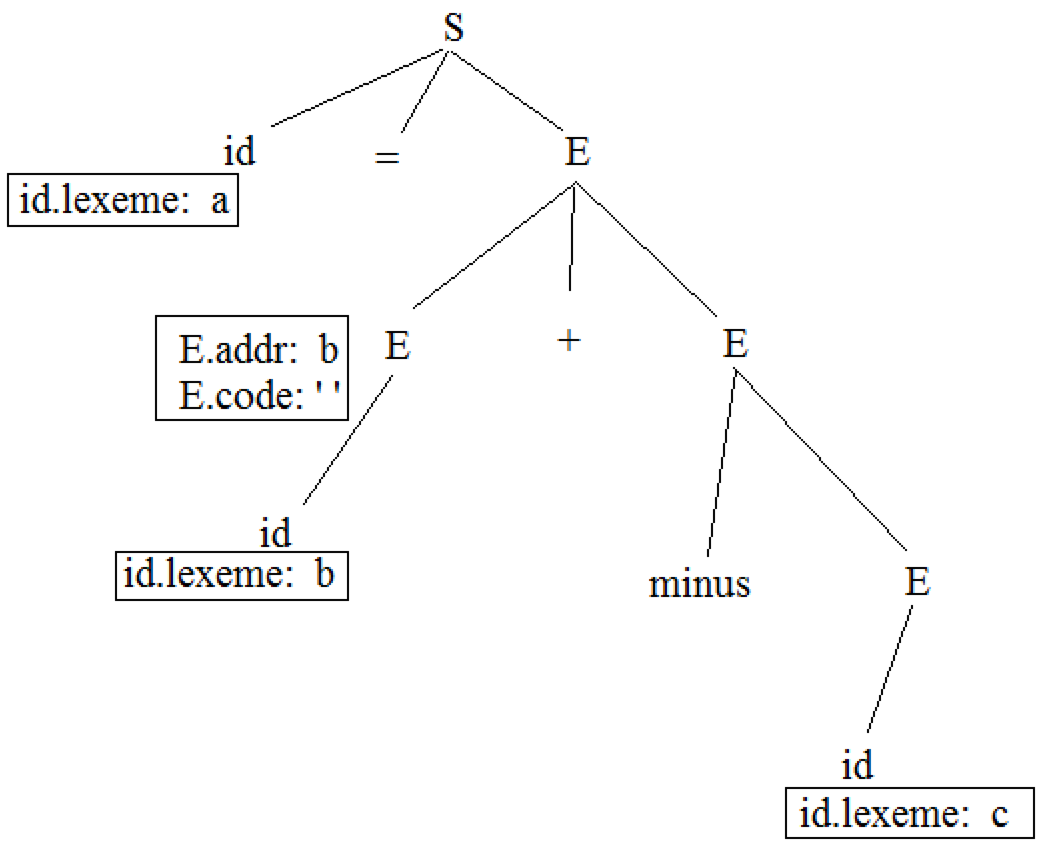
E.addr can be a name, a constant, or a compiler-generated temporary.

This is S-attributed SDD

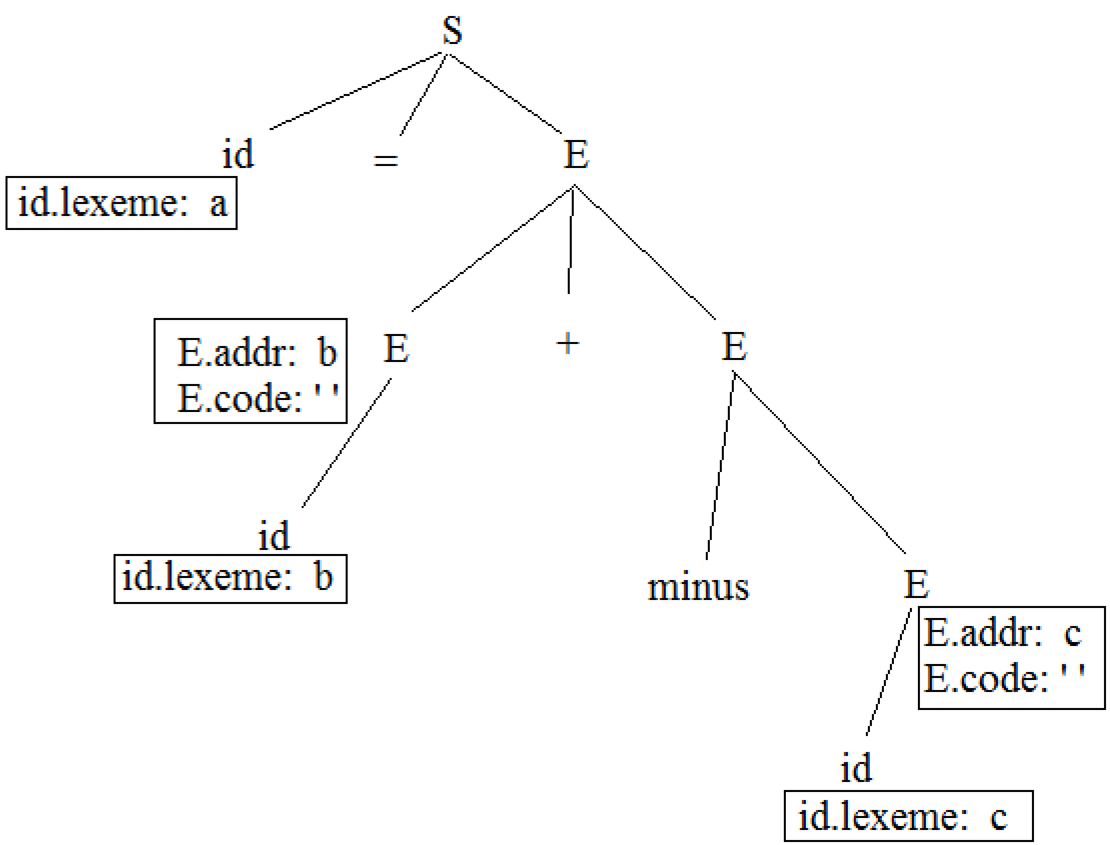
input: a = b + - c



input: a = b + - c

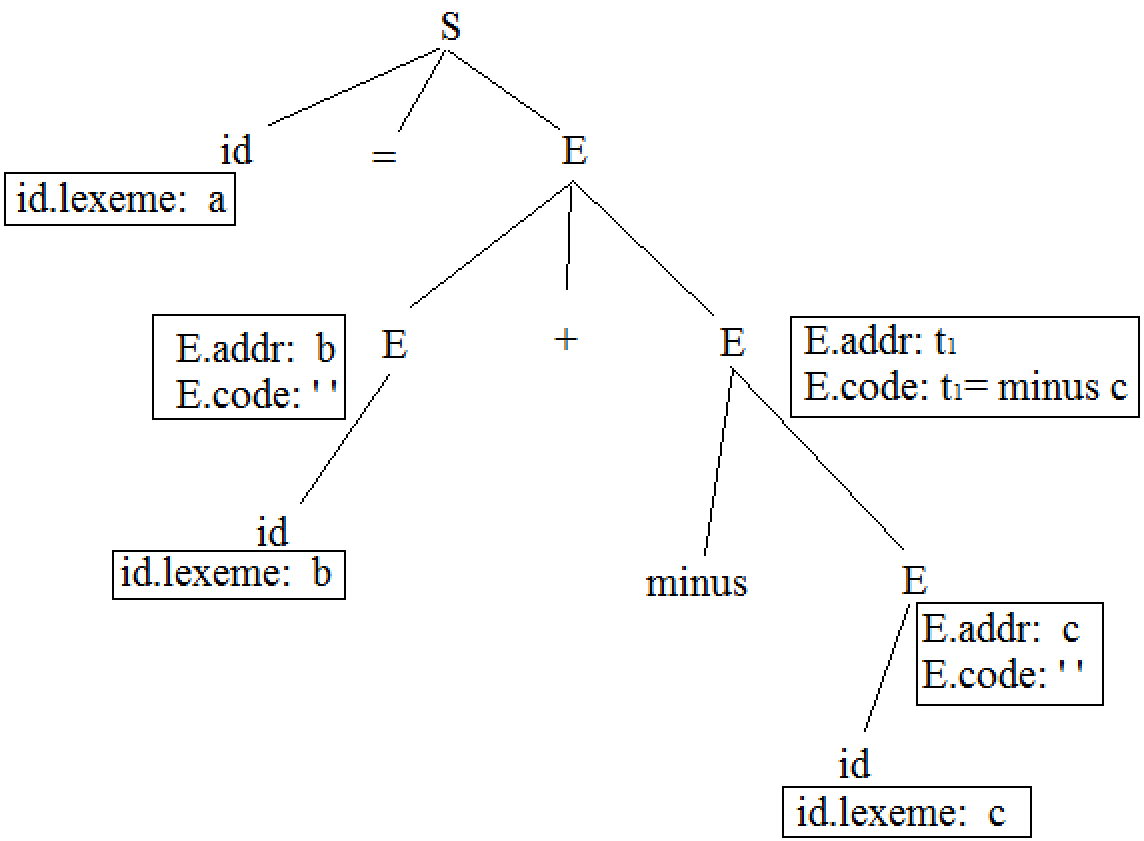

S
id
=
E
id.lexeme: a
E.addr: b
E.code: ' '
E
+
E
id
id.lexeme: b
minus
E
id
id.lexeme: c

input: a = b + - c

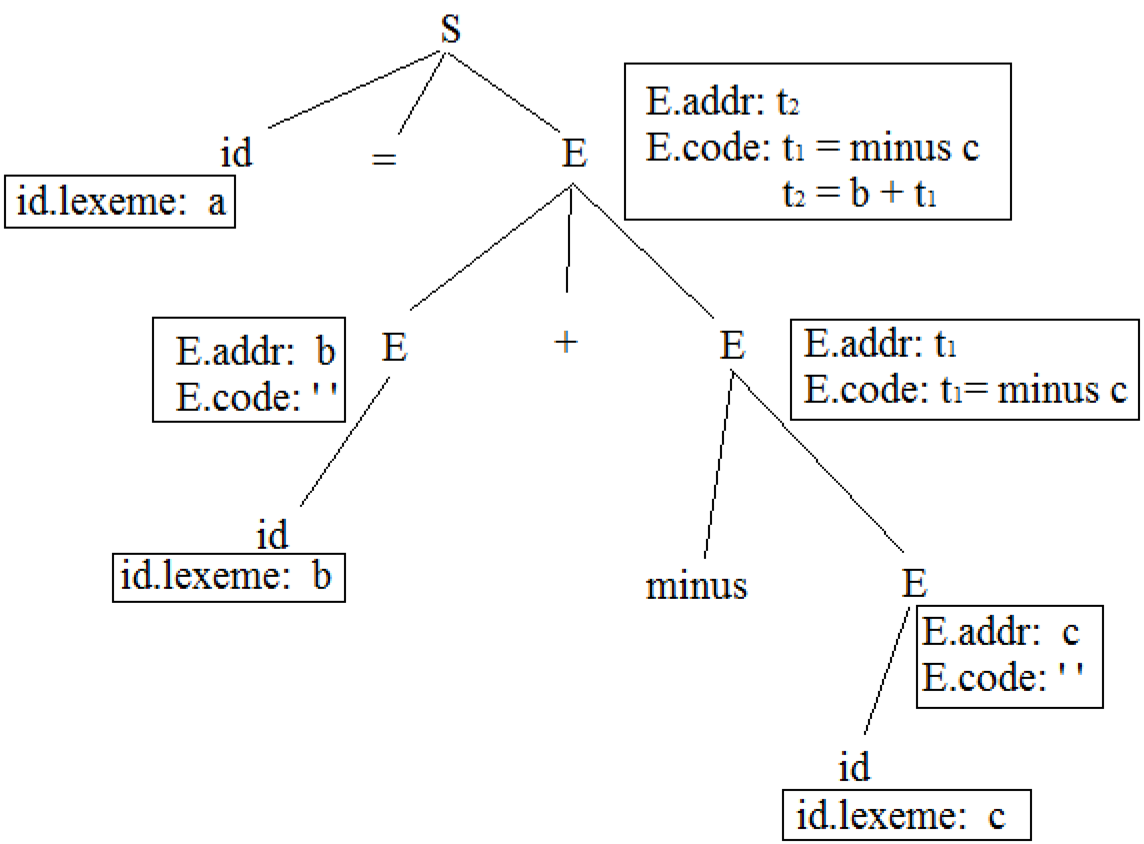


S
id
id.lexeme: a
=
E
E.addr: b
E.code: ' '
E
+
E
id
id.lexeme: b
minus
E
E.addr: c
E.code: ' '
id
id.lexeme: c

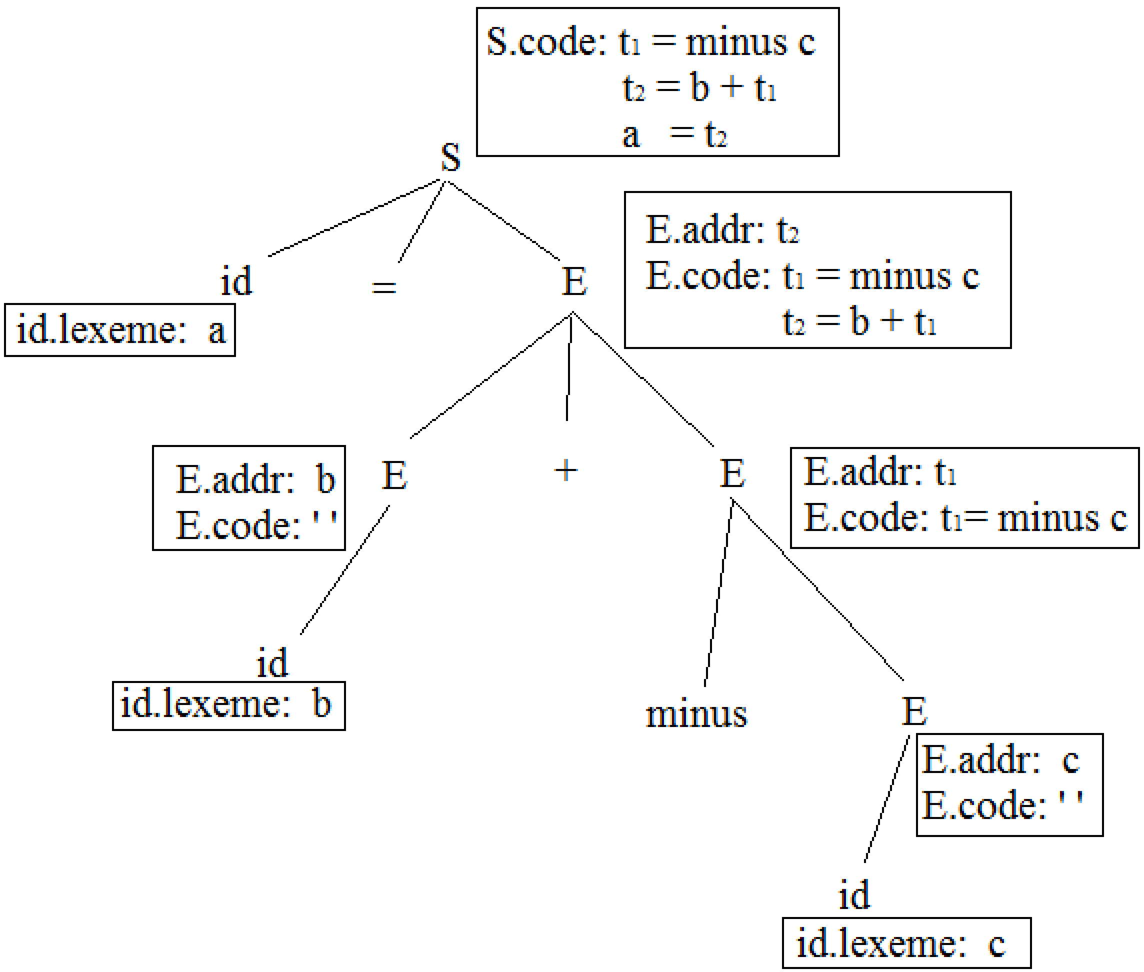
input: a = b + - c


S
id
=
E
id.lexeme: a
E.addr: b
E.code: ' '
E
+
E
E.addr: t1
E.code: t1= minus c
id
id.lexeme: b
minus
E
E.addr: c
E.code: ' '
id
id.lexeme: c

input: a = b + - c


S
id
id.lexeme: a
=
E
E.addr: t2
E.code: t1 = minus c
t2 = b + t1
E
E.addr: b
E.code: ' '
+
E
E.addr: t1
E.code: t1= minus c
id
id.lexeme: b
minus
E
E.addr: c
E.code: ' '
id
id.lexeme: c

input: a = b + - c



S.code: t1 = minus c
t2 = b + t1
a = t2
S
id
id.lexeme: a
=
E
E.addr: t2
E.code: t1 = minus c
t2 = b + t1
E.addr: b
E.code: ' '
E
+
E
E.addr: t1
E.code: t1= minus c
id
id.lexeme: b
minus
E
E.addr: c
E.code: ' '
id
id.lexeme: c

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

Figure 6.19: Three-address code for expressions

Attributes: *code* for S; *addr* and *code* for an Expression E.

E.addr can be a name, a constant, or a compiler-generated temporary.

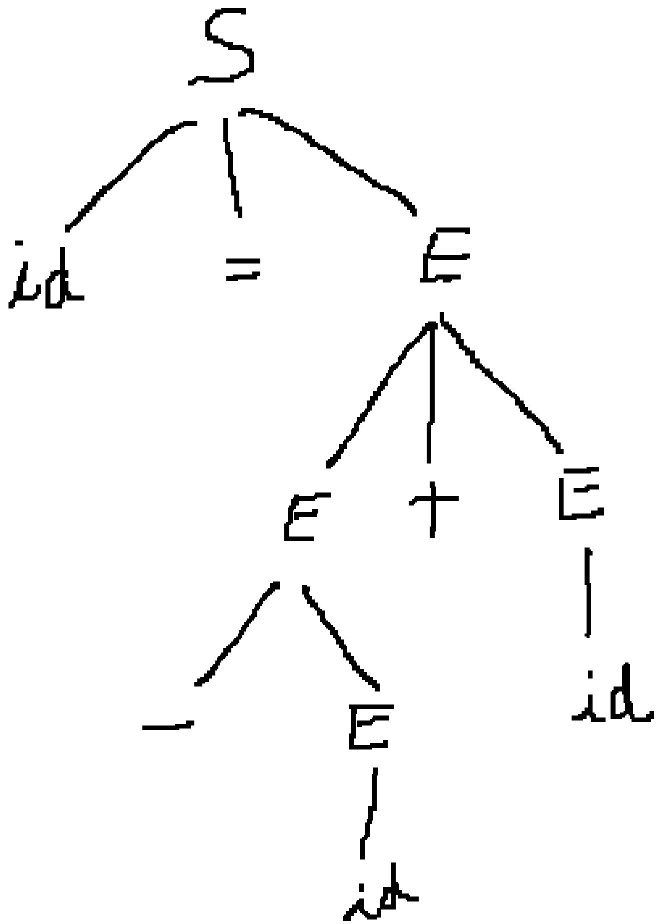
Example 6.11: The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c`; into the three-address code sequence

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

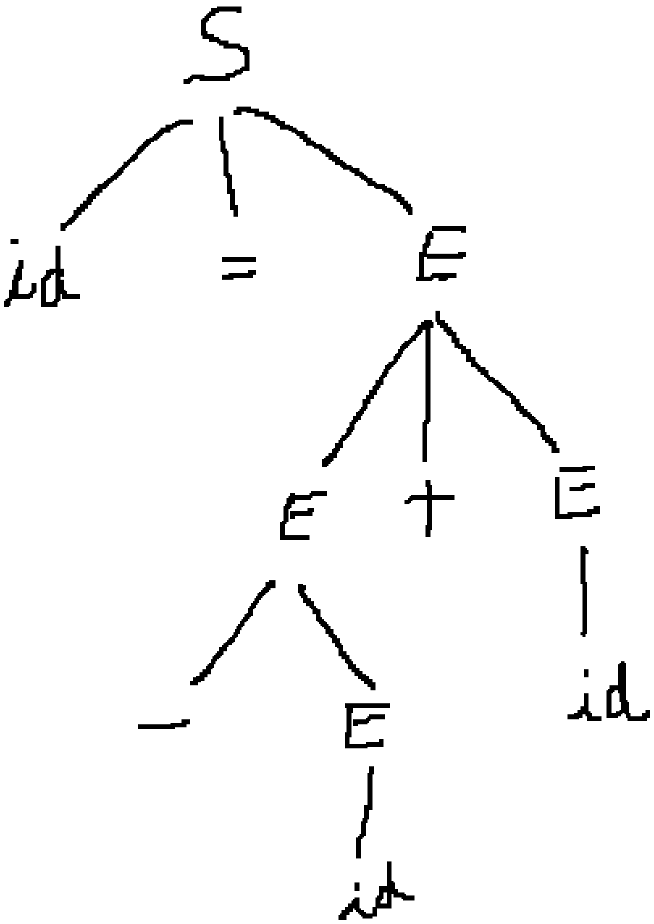
Can you do these?

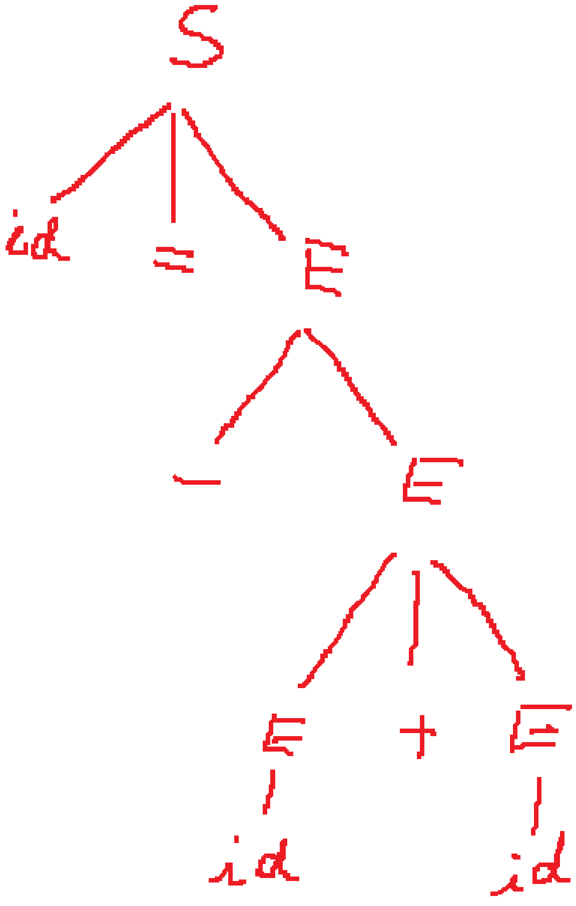
- Can you follow this to find the translation for a=-b+c;
- Is there anything odd you noticed?
- Can you add multiplication to this SDD?
- Work with a = b+c*d+e and produce three-address code.

$$a = -b + c$$

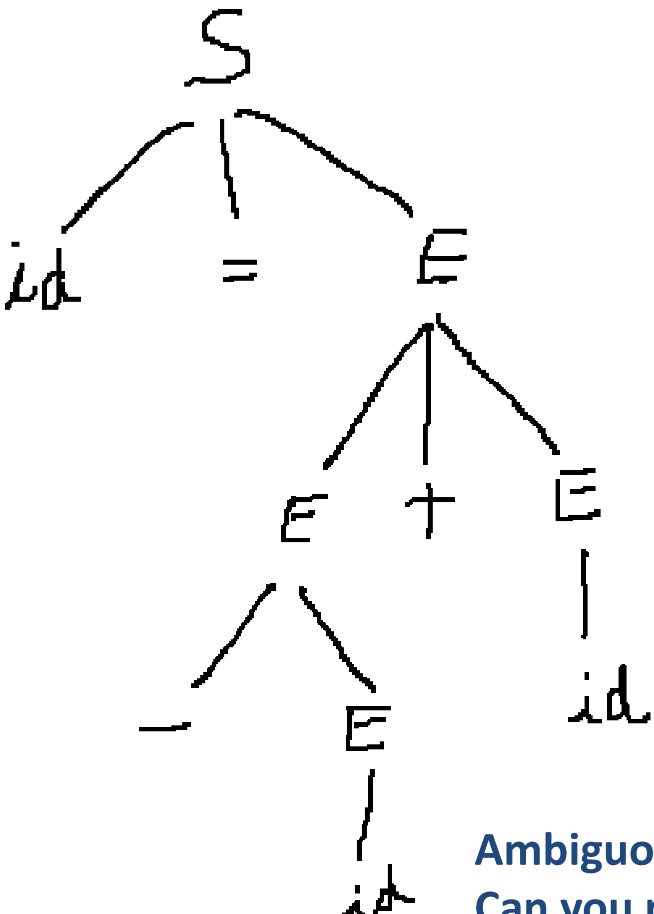


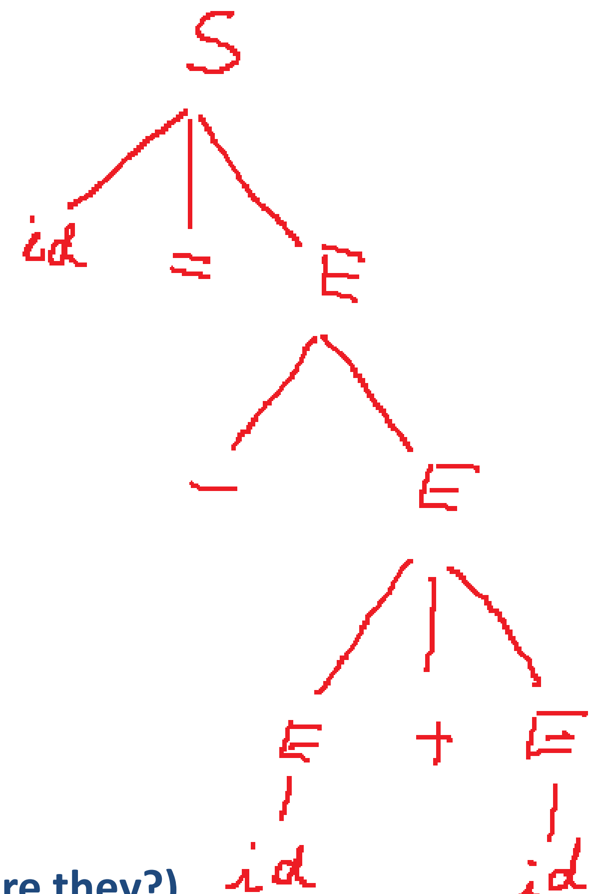
$$a = -b + c$$





$$a = -b + c$$





Ambiguous; two answers (what are they?)
Can you repair this to always give the correct answer

Incremental Translation

- Instead of having the entire code to be accumulated as an attribute of the root node
  - One can generate piece by piece of code incrementally.

- SDT doing this looks rather simple.

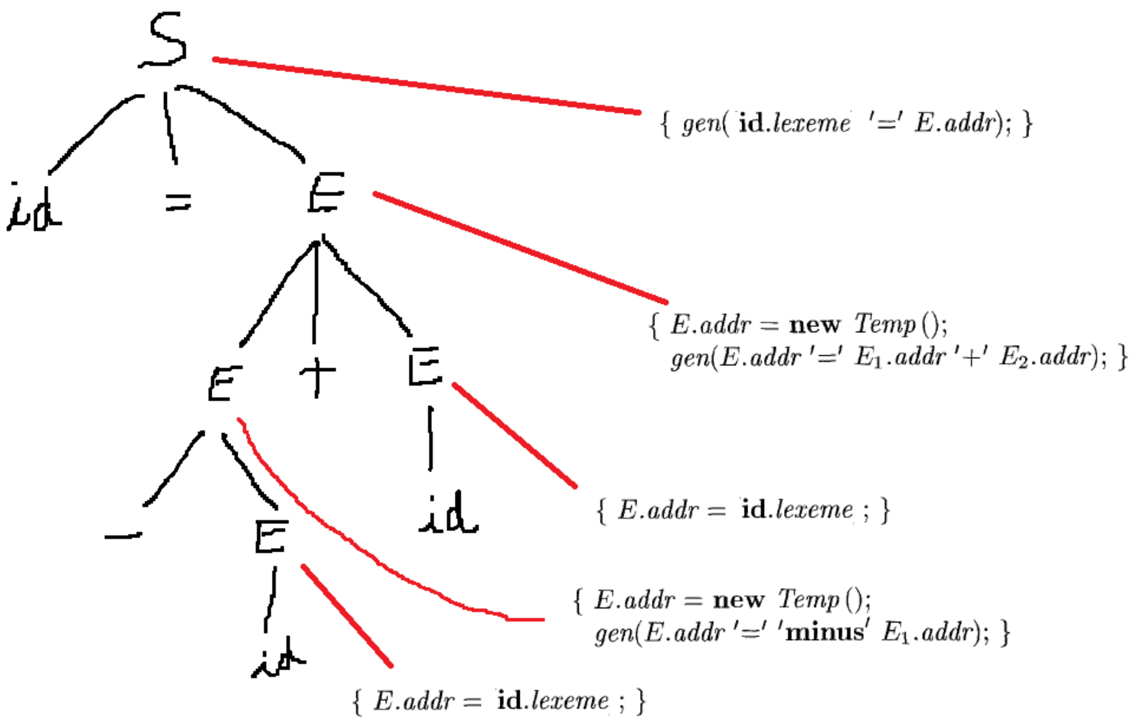
$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E \; ; & \{ \; gen(\, \textbf{id}.lexeme \;\; '=' \; E.addr); \; \} \\[2ex]
E & \rightarrow & E_1 + E_2 & \{ \; E.addr = \textbf{new} \;\; Temp\,(); \\
 & & & \quad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \} \\[2ex]
 & | & - \; E_1 & \{ \; E.addr = \textbf{new} \;\; Temp\,(); \\
 & & & \quad gen(E.addr \; '=' \; '\textbf{minus}' \; E_1.addr); \; \} \\[2ex]
 & | & (\; E_1 \;) & \{ \; E.addr = E_1.addr; \; \} \\[2ex]
 & | & \textbf{id} & \{ \; E.addr = \; \textbf{id}.lexeme \; ; \; \}
\end{array}
$$

Figure 6.20: Generating three-address code for expressions incrementally

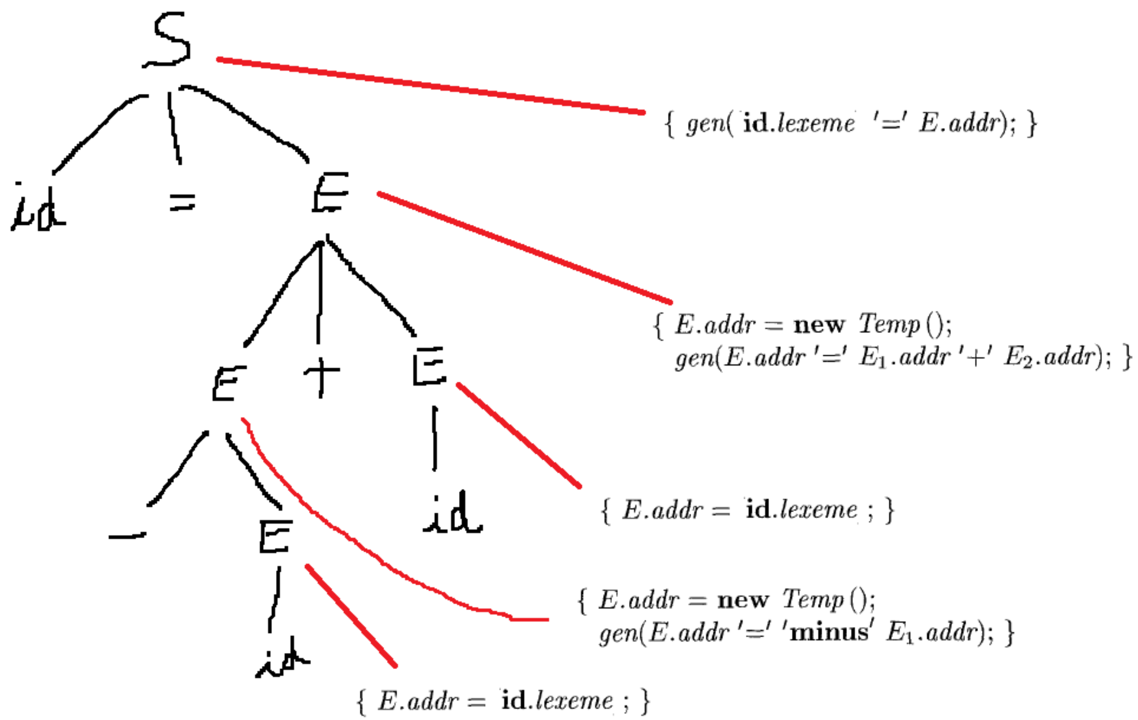
Note: The attribute code is not used. Because there is no need to accumulate the code.

top.get(id.lexeme) refers to the current symbol table and gets id.lexeme. Book uses this instead of just **id.lexeme**

$$\boxed{a = -b + c}$$

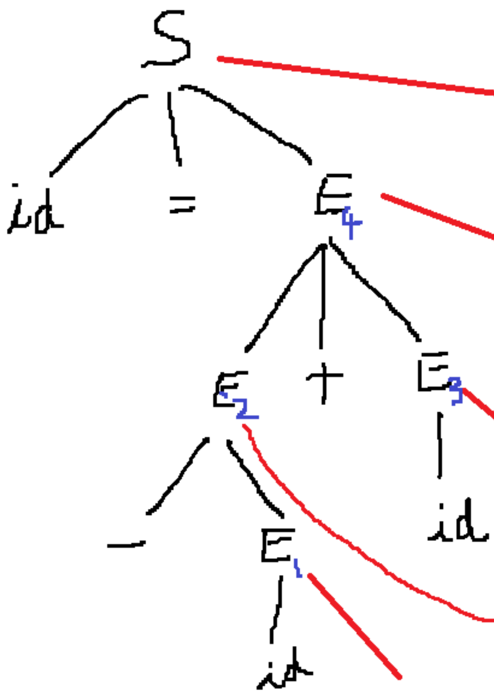

S
id
=
E
E
+
E
−
E
id
id
{ gen(id.lexeme '=' E.addr); }
{ E.addr = new Temp();
gen(E.addr '=' E1.addr '+' E2.addr); }
{ E.addr = id.lexeme ; }
{ E.addr = new Temp();
gen(E.addr '=' 'minus' E1.addr); }
{ E.addr = id.lexeme ; }

$$a = -b + c$$



- For our convenience let us give subscripts, etc.

$$a = -b + c$$



$\{\ gen(\ \mathbf{id}.lexeme\ \ '='\ E_4.addr);\ \}$

a

$\{\ E_4.addr = \mathbf{new}\ \ Temp\,();$
$\ \ gen(E_4.addr\ '='\ E_2.addr\ '+'\ E_3.addr);\ \}$

$\{\ E_3.addr =\ \mathbf{id}.lexeme\ ;\ \}$

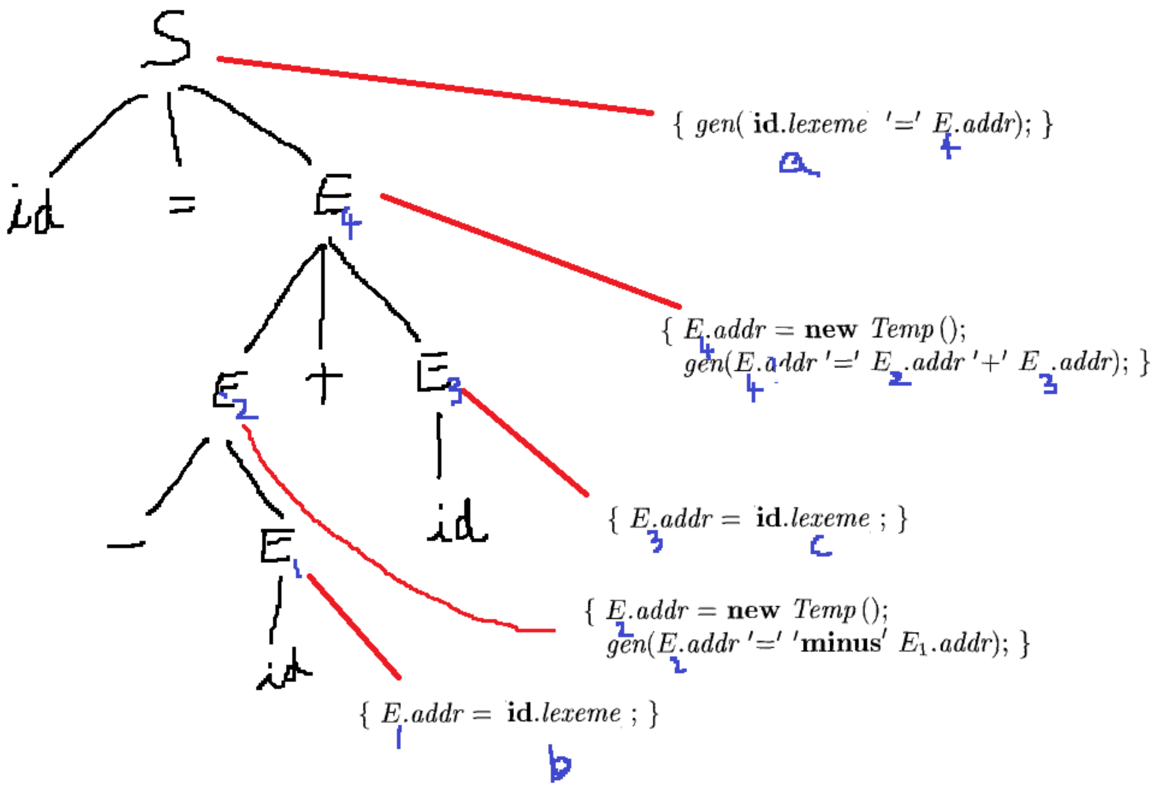
c

$\{\ E_2.addr = \mathbf{new}\ \ Temp\,();$
$\ \ gen(E_2.addr\ '='\ '\mathbf{minus}'\ E_1.addr);\ \}$

$\{\ E_1.addr =\ \mathbf{id}.lexeme\ ;\ \}$

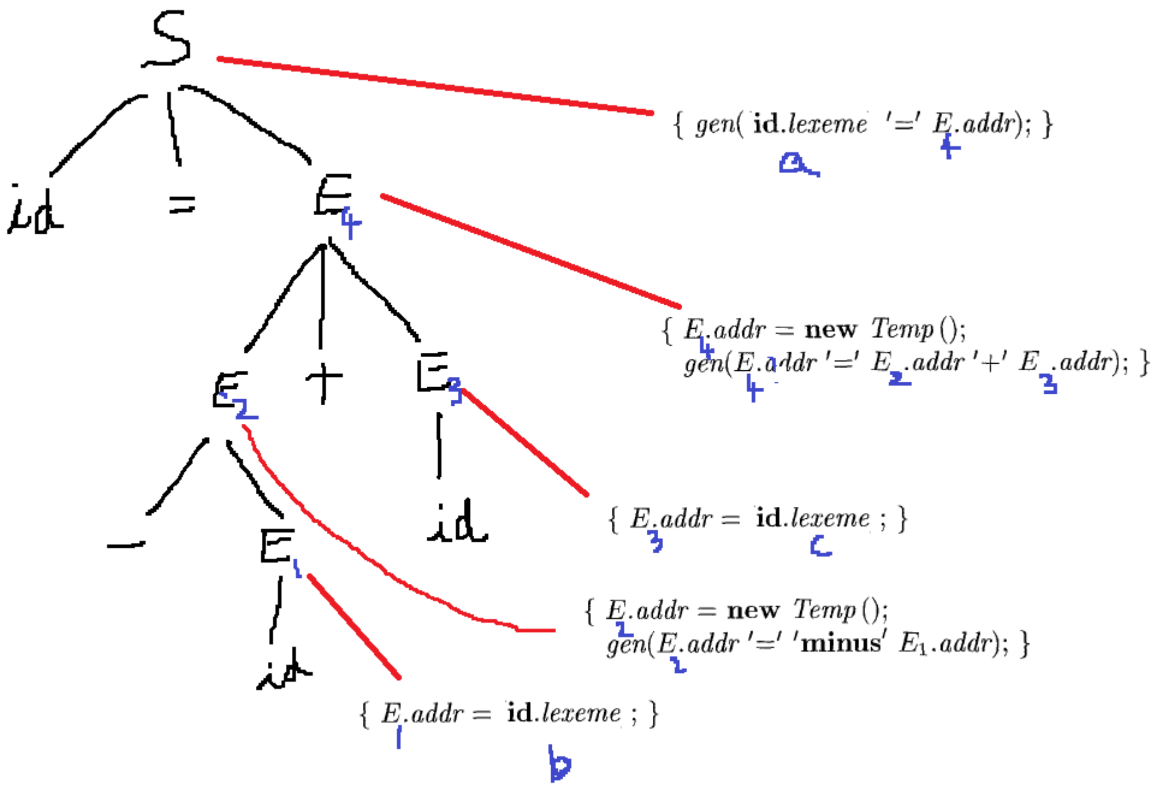
b

$$a = -b + c$$



addr, lexeme are attributes. new operator creates a new temp variable { each time with a new subscript like t1, t2, so on }. gen ( ) is going to generate the out put {prints its argument}.

$$\boxed{a = -b + c}$$



addr, lexeme are attributes. new operator creates a new temp variable { each time with a new subscript like t1, t2, so on }. gen ( ) is going to generate the out put {prints its argument}.

t1 = -b;

t2 = t1 + c;

a = t2;

6.4.3 Addressing Array Elements

Array elements can be accessed quickly if they are stored in a block of consecutive locations.

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.

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

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

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).

To generalize further,

More generally, array elements need not be numbered starting at 0.

To generalize further,

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 $A[low]$.

To generalize further,

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:

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

When is the address of a data area is calculated?

- Compile time pre-calculation of addresses can be done for static arrays (declaration specifies size).
- But for dynamic arrays this cannot be done.

When is the address of a data area is calculated?

- Compile time pre-calculation of addresses can be done for static arrays (declaration specifies size).
- But for dynamic arrays this cannot be done.

- Arrays can be stored in ***row major*** layout
  - This is what we assumed so far and is used in C, Java and many other languages
- Arrays can be stored in ***column major*** layout
  - For example, Matlab can choose between these two, or may represent in both forms (same object in different representations)

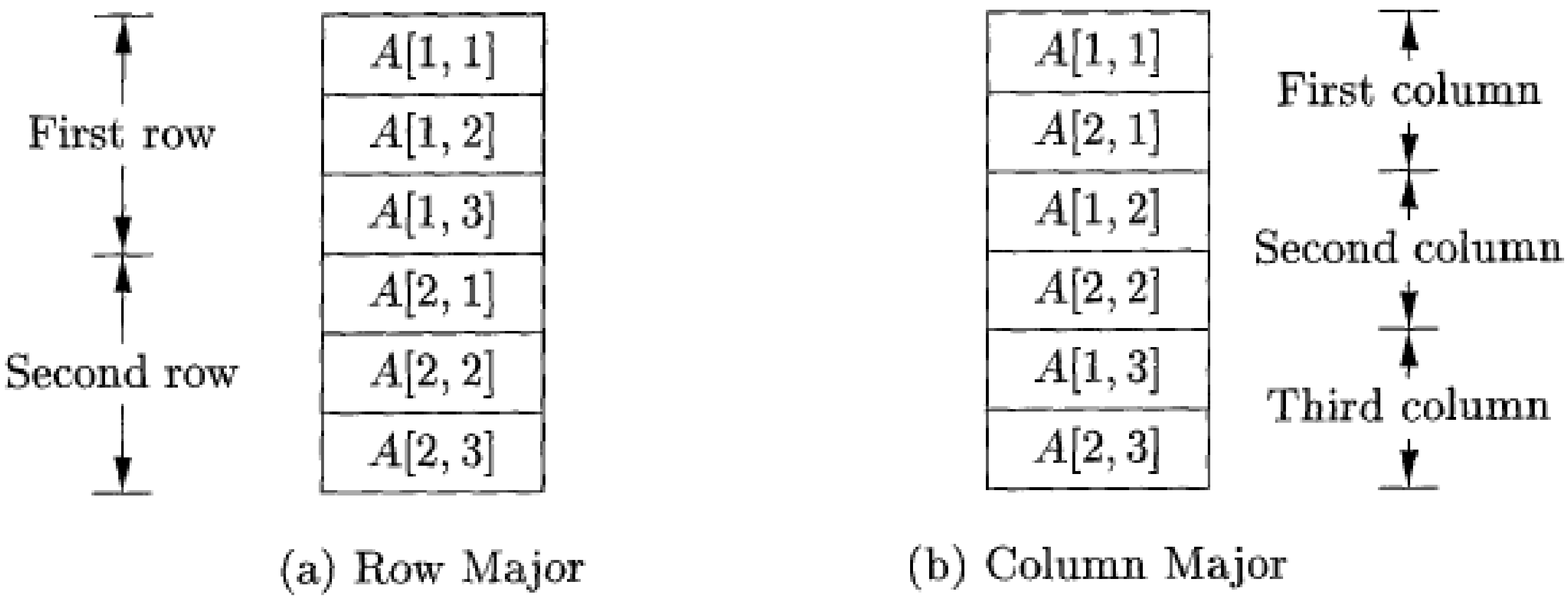


Figure 6.21: Layouts for a two-dimensional array.

Generalization to many dimensions

- Row major layout for A[i][j][k]
  - Assume you stored it in row major layout
  - As you scan the memory, you will encounter various elements of A.
  - Subscript k varies fastest and i varies slowest.
- Column major layout is opposite

- We have seen, how to find size (memory required) for arrays, multi-dimensional arrays.
- SDD/SDT which converts something like A[i][j][k] = b+c; in to three-address code can be created by combining, SDT for declarations and SDT for expressions (with arrays) as shown in the next two slides.

Recall SDT for declaration

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\, T.type = B.type; \quad T.width = B.width; \\
 & & C & \quad t = B.type;\ w = B.width;\, \} \\
B & \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\quad C.type = array(\textbf{num}.value,\ C_1.type); \\
 & & & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

Figure 6.15: Computing types and their widths

SDT for expressions with array arguments

$$
\begin{array}{lll}
S & \rightarrow \ \mathbf{id} = E \ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\[1em]
 & \mid \ L = E \ ; & \{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\[1em]
E & \rightarrow \ E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\[1em]
 & \mid \ \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \} \\[1em]
 & \mid \ L & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & & \quad gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} \\[1em]
L & \rightarrow \ \mathbf{id}\ [\ E\ ] & \{\ L.array = top.get(\mathbf{id}.lexeme); \\
 & & \quad L.type = L.array.type.elem; \\
 & & \quad L.addr = \mathbf{new}\ Temp\,(); \\
 & & \quad gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\[1em]
 & \mid \ L_1\ [\ E\ ] & \{\ L.array = L_1.array; \\
 & & \quad L.type = L_1.type.elem; \\
 & & \quad t = \mathbf{new}\ Temp\,(); \\
 & & \quad L.addr = \mathbf{new}\ Temp\,(); \\
 & & \quad gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
 & & \quad gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

Figure 6.22: Semantic actions for array references

- We will see about **type system** and **type checking**.

6.5 Type Checking

- A type system is a set of rules that assigns a type to various constructs of the language, such as variables, expressions, functions, etc.
- The main purpose of a type system is to reduce possibilities for bugs in computer programs.
- checking can happen statically (at compile time), dynamically (at run time), or as a combination of static and dynamic checking.

6.5 Type Checking

- A strongly typed HLL guarantees that the programs it accepts will run without type errors.
  - Bugs are reduced.
- Security is increased.
  - Java byte code comes with variables and their types also. It can not do whatever it wants.. JVM can check for its behavior.

6.5.1 Rules for Type Checking

- Type checking can take two forms
  - Synthesis
  - inference

Rules for Type Checking

- **Type synthesis**: Find type of an expression from the types of its subexpressions.
  - Basic elements like ids must be declared before they are used. {so that we know their type}.
  - Type of E1 + E2 is determined from types of E1 and E2.
- A typical rule for type synthesis is:

$$\begin{array}{l}\textbf{if } f \text{ has type } s \rightarrow t \textbf{ and } x \text{ has type } s,\\ \textbf{then} \text{ expression } f(x) \text{ has type } t\end{array} \qquad (6.8)$$

- E1+E2 has type add(E1, E2).
- **Type inference** determines the type of a language construct from the way it is used.
- Eg: Let *null(x)* be a function that tests whether a list is empty.
  - Then from *null(x)*, we can tell that *x* must be a list.
  - The type of elements of the list is unknown (at present); even then we can say it is a list.

- Type Inference:
  - If(E) S; /* type of E must be boolean*/
- Variables representing type expressions allow us to talk about unknown types.
- Dragon book uses Greek letters $\alpha, \beta, \ldots$ for type variables in type expressions.
- For the expression, $f(x)$, one can assume that there is a type $\alpha \rightarrow \beta$ for f and α is the type of x

- Type inference allows polymorphism, i.e., based on the context, the type is found.
- f might have two types(overloaded) $int \rightarrow float$ and $char \rightarrow int$.
- Now $f(5)$ says the type of f is $int \rightarrow float$
  – Accordingly the correct function is called.

6.5.2 Type Conversions

- How 2*3.14 is translated.
  - For int type their element representation and multiplication can be different from that of float elements.

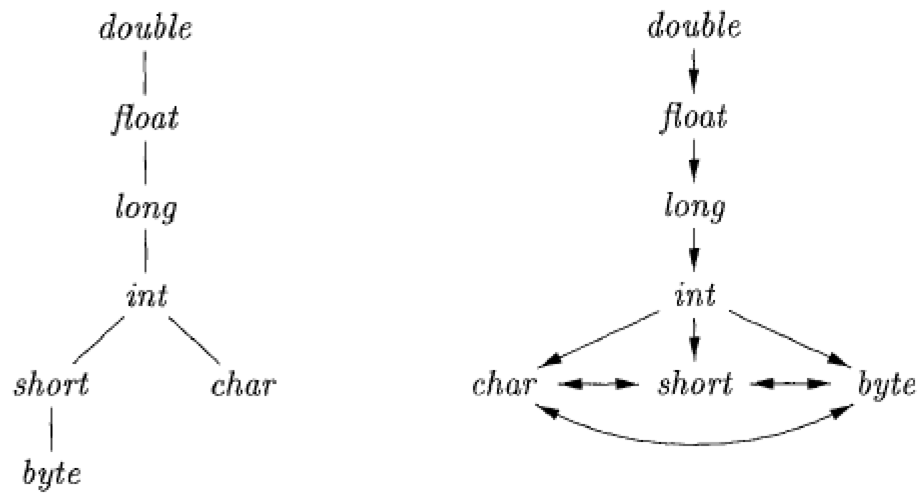
- Unary operators to convert type can be used by the programmer (explicit type coversion).
  - Type casting.
- Compiler can automatically do such conversions (coercions). Three address code for 2*3.14:

$$t_1 = (\texttt{float})\ 2$$
$$t_2 = t_1 * 3.14$$

Type conversion rules

- Can vary from language to language.
- **Widening** conversion preserves the information.
- Whereas, **narrowing** conversions can lose.

Conversions in Java



(a) Widening conversions

(b) Narrowing conversions

Figure 6.25: Conversions between primitive types in Java

- Coercions are widening conversions mostly (except for assignment).
- In assignment narrowing is used mostly.
- SDD/SDT can automatically put code to do these conversions.

The semantic action for checking $E \to E_1 + E_2$ uses two functions:

The semantic action for checking $E \to E_1 + E_2$ uses two functions:

1. $max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.

The semantic action for checking $E \rightarrow E_1 + E_2$ uses two functions:

1. $max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.

2. $widen(a, t, w)$ generates type conversions if needed to widen an address a of type t into a value of type w. It returns a itself if t and w are the same type. Otherwise, it generates an instruction to do the conversion and place the result in a temporary t, which is returned as the result. Pseudocode for *widen*, assuming that the only types are *integer* and *float*, appears in Fig. 6.26.

A sample code for widen (this should be extended to cover all possibilities)

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

Figure 6.26: Pseudocode for function *widen*

SDT

$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{ & E.type \; = \; max(E_1.type, E_2.type); \\
 & & a_1 \; = \; widen(E_1.addr, E_1.type, E.type); \\
 & & a_2 \; = \; widen(E_2.addr, E_2.type, E.type); \\
 & & E.addr = \mathbf{new} \; Temp\,(); \\
 & & gen(E.addr \; '=' \; a_1 \; '+' \; a_2); \; \}
\end{array}
$$

Figure 6.27: Introducing type conversions into expression evaluation

6.5.3 Overloading of Functions and Operators

- An overloaded symbol has different meanings based on its context.
- Overloading is said to be resolved when a unique meaning is determined for each occurrence of a name.

Example 6.13: The + operator in Java denotes either string concatenation or addition, depending on the types of its operands. User-defined functions can be overloaded as well, as in

```
void err() { ··· }
void err(String s) { ··· }
```

Note that we can choose between these two versions of a function `err` by looking at their arguments. □

When overloading is allowed?

The following is a type-synthesis rule for overloaded functions:

$$\begin{array}{ll} \textbf{if } f \text{ can have type } s_i \to t_i, \text{ for } 1 \le i \le n, \text{ where } s_i \neq s_j \text{ for } i \neq j & \\ \textbf{and } x \text{ has type } s_k, \text{ for some } 1 \le k \le n & (6.10) \\ \textbf{then } \text{expression } f(x) \text{ has type } t_k & \end{array}$$

- The signature for a function consists of the function name and the types of its arguments.
- Overloading can be resolved based on argument types is equivalent to saying that overloading can be resolved based on signatures.

- The rest of intermediate code generation are left as a reading assignment.