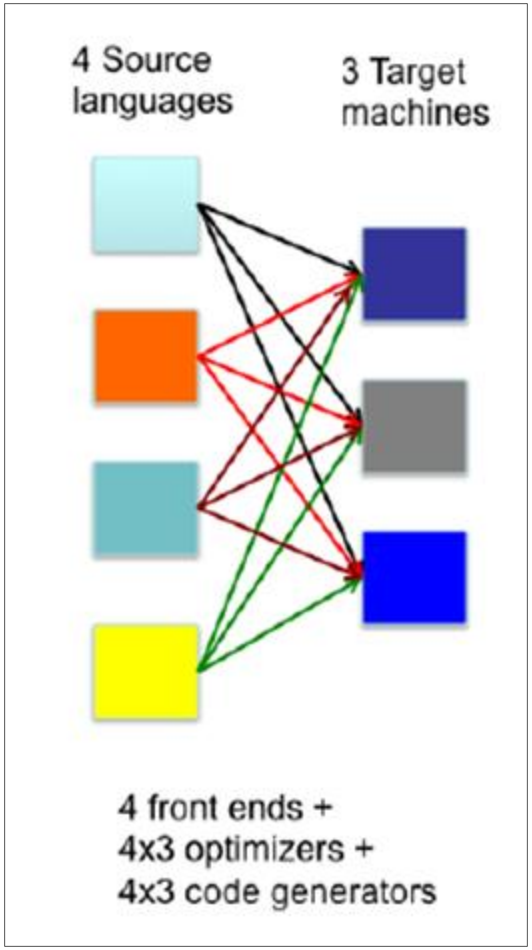
Chapter 6

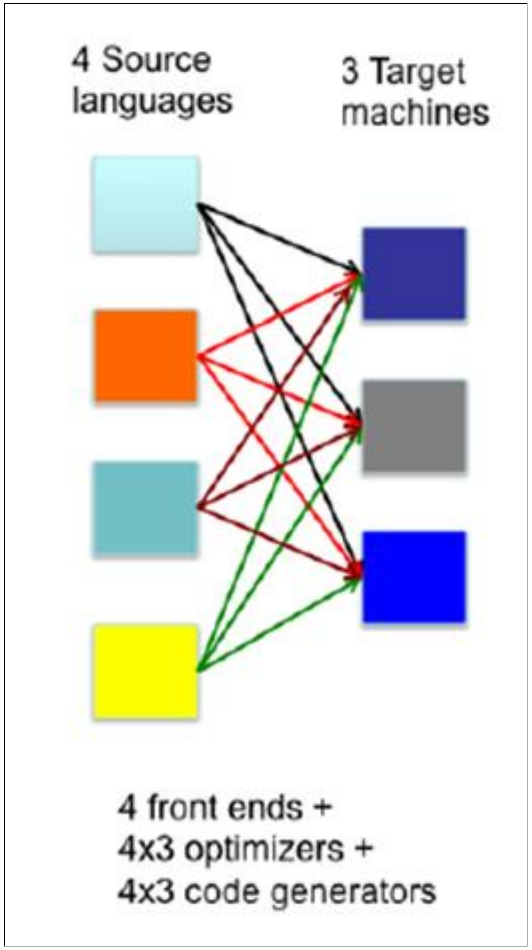
Intermediate-Code Generation

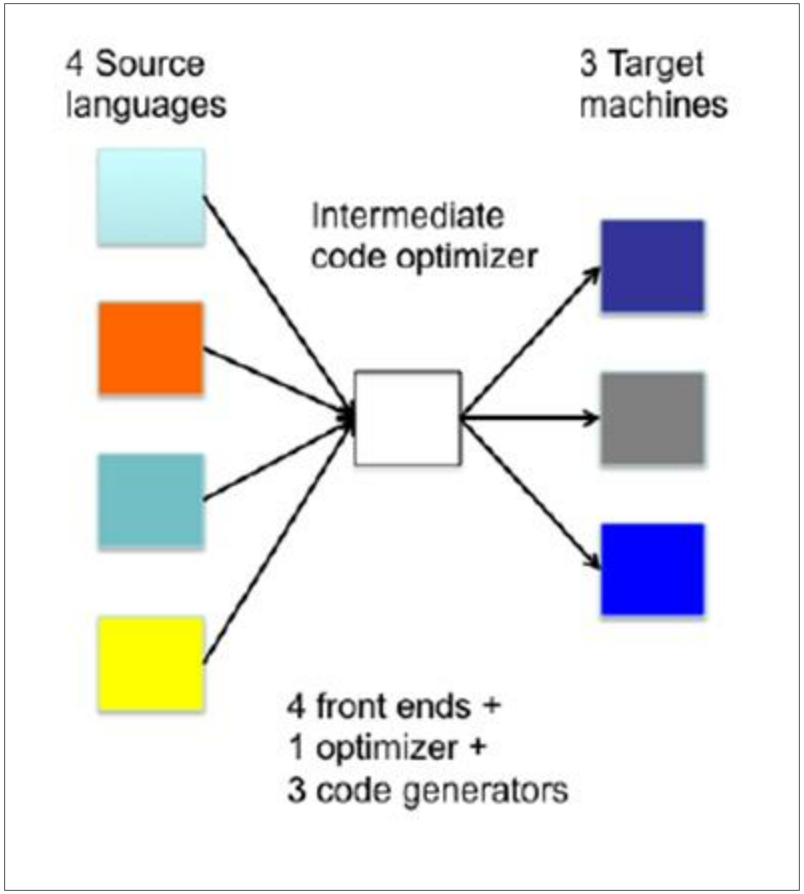
Why Intermediate Code?

Why Intermediate Code?



Why Intermediate Code?





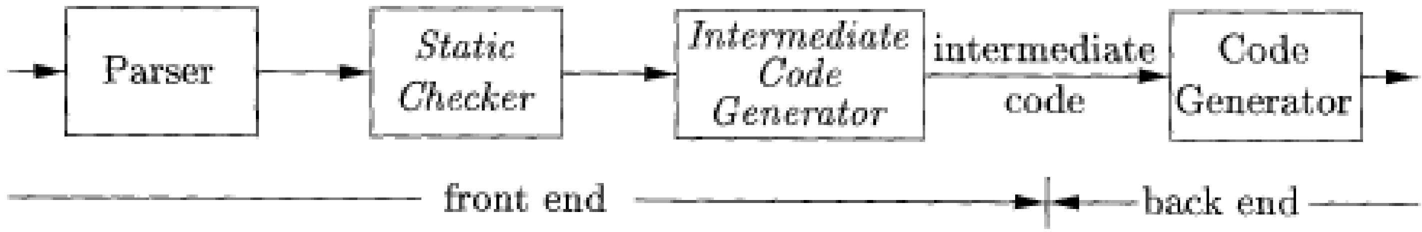


Figure 6.1: Logical structure of a compiler front end

- Instead of $m \times n$ compilers we can have m front ends and $n$backends.
  - Each front end gives the same intermediate representation and back end is going use this.
- C was used as an inter. representation for designing C++.
  - For many machines, C compilers are existing!

Other advantages …

Other advantages …

- Machine independent code optimization.
  - A separate field of study.

Other advantages …

- Machine independent code optimization.
  - A separate field of study.
- We can assume that there is an virtual machine which runs the intermediate code.
  - Interpreters for various machines can be written.
  - Portability increased.
  - Java. JVM.

Different Types of Intermediate Code

- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)
- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations
  - Conditional constant propagation and global value numbering are more effective on SSA
- Program Dependence Graph (PDG) is useful in automatic parallelization, instruction scheduling, and software pipelining

Static Checking -- SDD

Static Checking -- SDD

- Static checking and intermediate code generation can be done with the help of SDTs.

Static Checking -- SDD

- Static checking and intermediate code generation can be done with the help of SDTs.
- Static checking
  - Type mismatch checking of operands of an operator.
  - Ensuring that break; stmt should occur within a loop or switch ---- anywhere else it is an error.
  - continue; should occur within a loop.

Various intermediate codes

Various intermediate codes

Figure 6.2: A compiler might use a sequence of intermediate representations

Various intermediate codes

Figure 6.2: A compiler might use a sequence of intermediate representations

- Higher level inter. Codes are closer to the HLL
- Low level inter. Codes are closer to the m/c.

- Originally, frontend for C++ simply translated into C code (intermediate language!). Then, backend is a C compiler.

Example intermediate representations

- Eg. Intermediate representations
  - Syntax trees
    - Flow is not visible, for example consider loops
  - Three address codes
    - Jump to a previous line is a loop

6.1 Variants of Syntax Trees

6.1 Variants of Syntax Trees

- Node in a syntax tree represent a construct in a source program.
  - It might be a sub-expression, an operator, …
  - Children are the components of that construct.

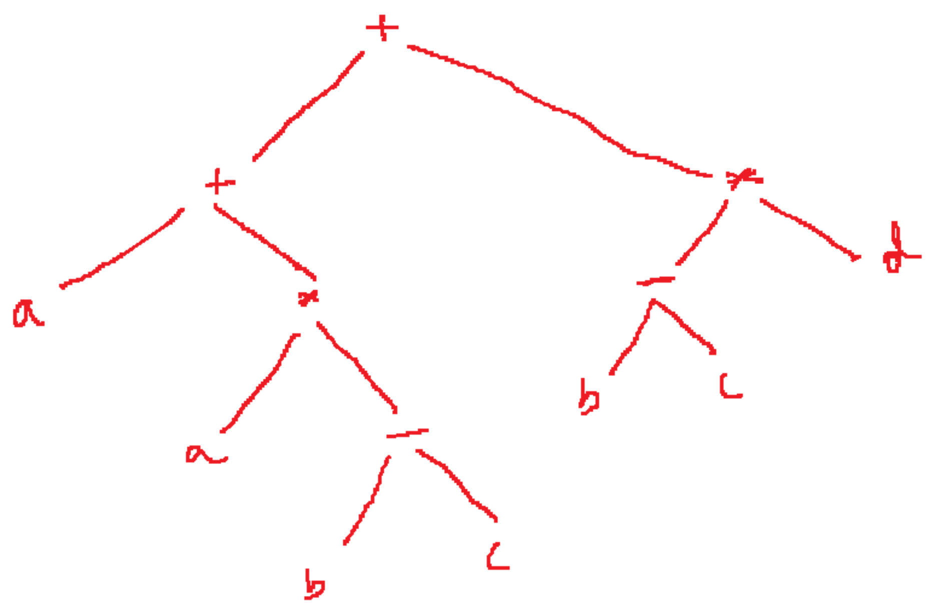
6.1 Variants of Syntax Trees

- Node in a syntax tree represent a construct in a source program.
  - It might be a sub-expression, an operator, …
  - Children are the components of that construct.
- A DAG (in contrast to tree) can factor-out common sub-expressions/ sub-constructs.
  - a child in a DAG can have more than one parent!
  - Cannot have cycles, as in the case of a tree.
- Efficient code can be generated.

Syntax tree for

$$a + a * (b - c) + (b - c) * d$$

	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \mathbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \mathbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \mathbf{id}$	$T.node = \mathbf{new}\ Leaf(\mathbf{id}, \mathbf{id}.entry)$
6)	$T \rightarrow \mathbf{num}$	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$

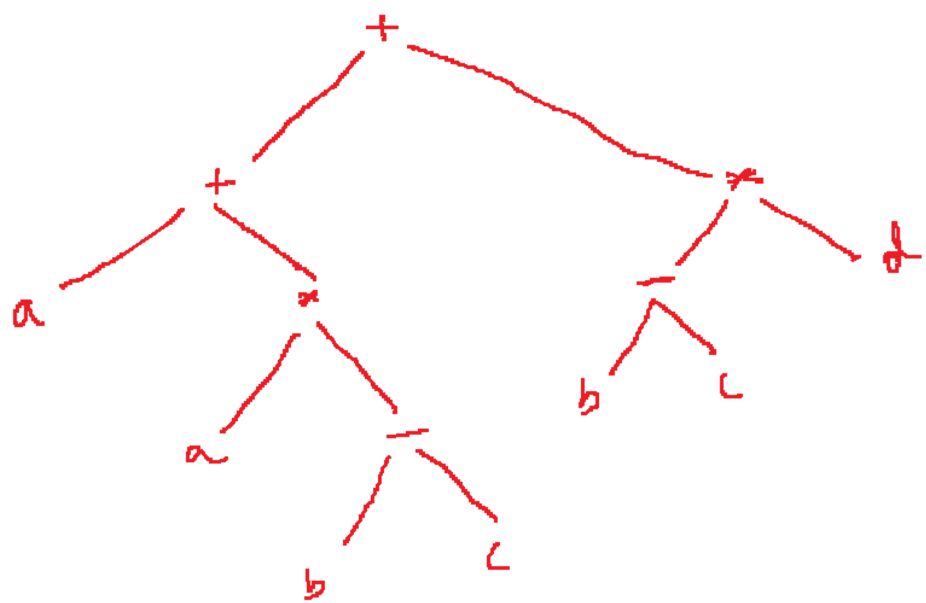


Syntax tree for

$$a + a * (b - c) + (b - c) * d$$

	Production	Semantic Rules
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

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



- But while creating a node we could have checked whether same node already exists… (simply reusing that pointer will do!!)

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) $p_{13} = Node('+', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

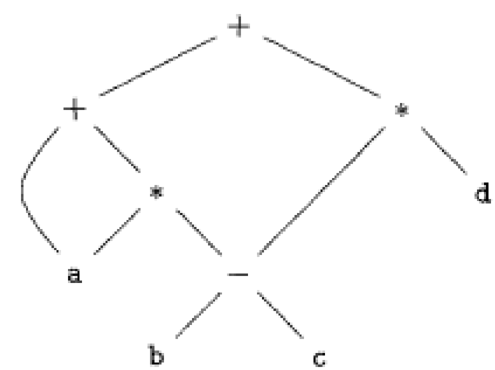


Figure 6.3: Dag for the expression $a + a * (b - c) + (b - c) * d$

6.1.1 Directed Acyclic Graphs for Expressions

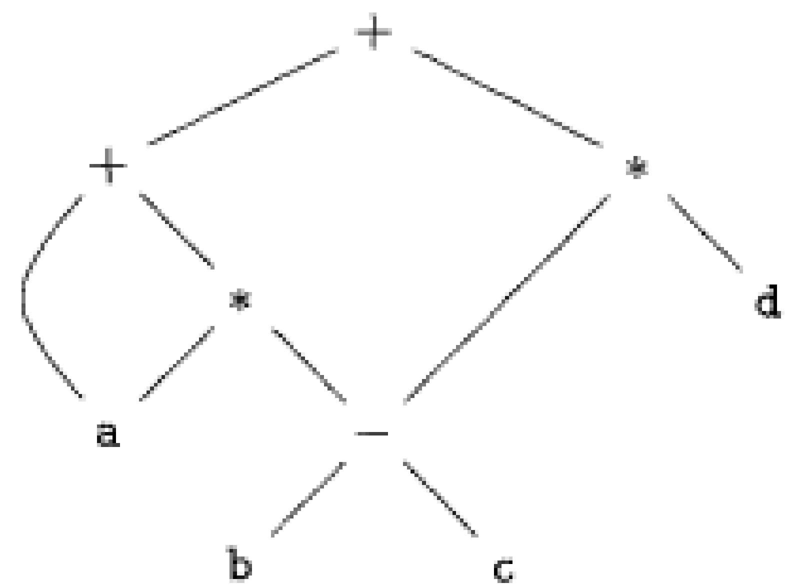


Figure 6.3: Dag for the expression a + a * (b - c) + (b - c) * d

- SDD can be used
- Modification we need is
  - Whenever you want to create a new node verify whether a node with identical information already exists … if so use that.

6.2 Three-Address Code

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.

Thus a source-language expression like `x+y*z` might be translated into the sequence of three-address instructions

$$\begin{aligned} \mathtt{t}_1 &= \mathtt{y} * \mathtt{z} \\ \mathtt{t}_2 &= \mathtt{x} + \mathtt{t}_1 \end{aligned}$$

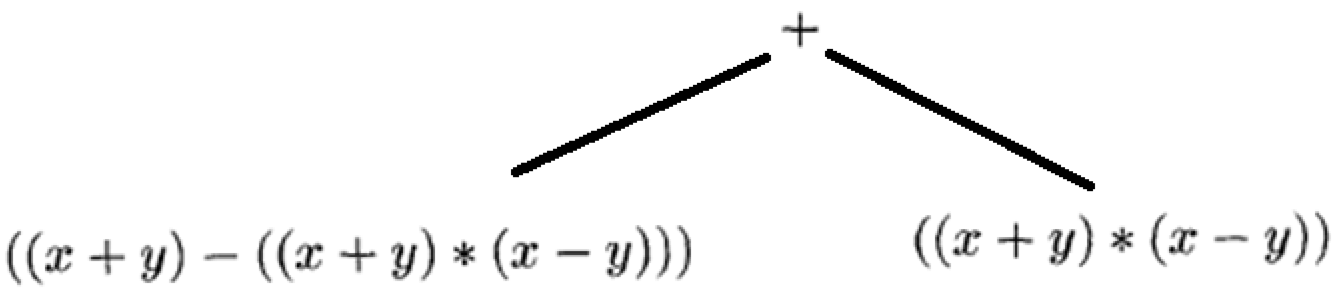
where $\mathtt{t}_1$ and $\mathtt{t}_2$ are compiler-generated temporary names.

Exercise 6.1.1: Construct the DAG for the expression

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

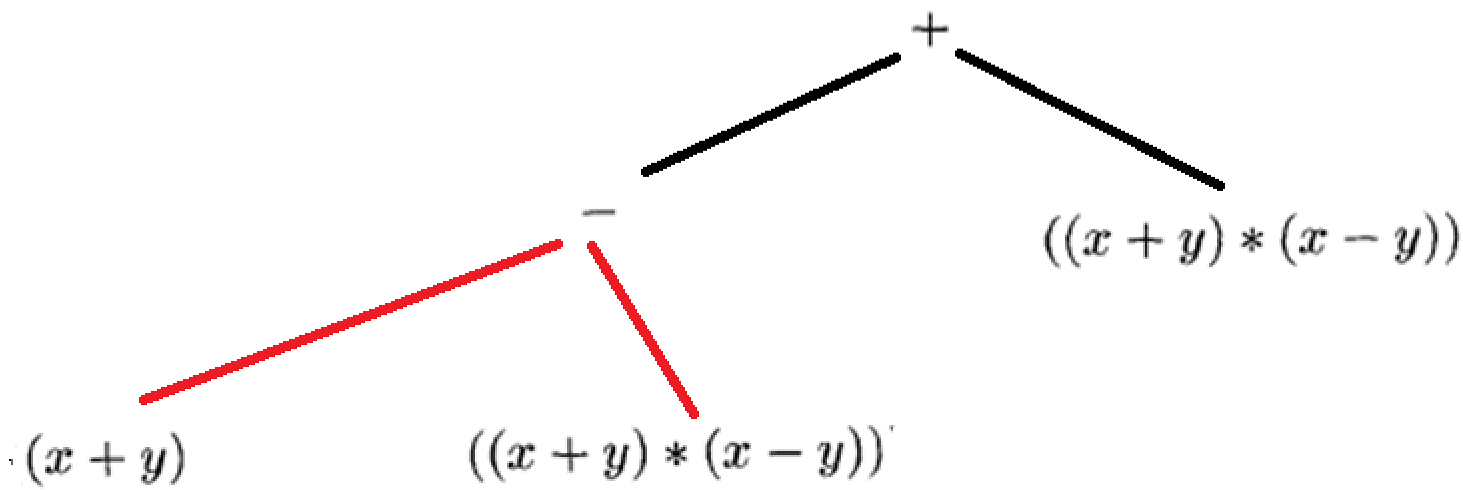
Exercise 6.1.1: Construct the DAG for the expression

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$



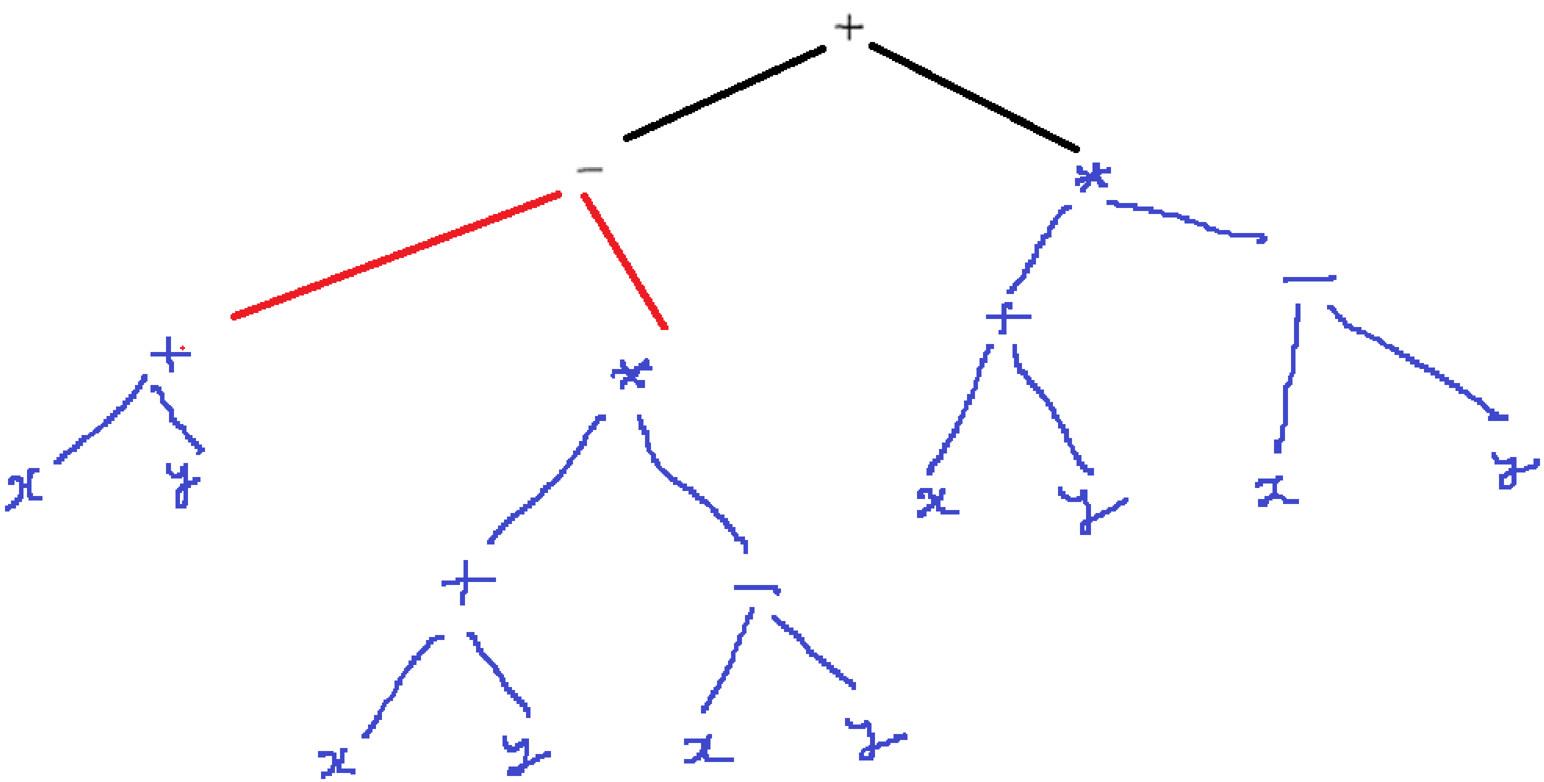
Exercise 6.1.1: Construct the DAG for the expression

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$



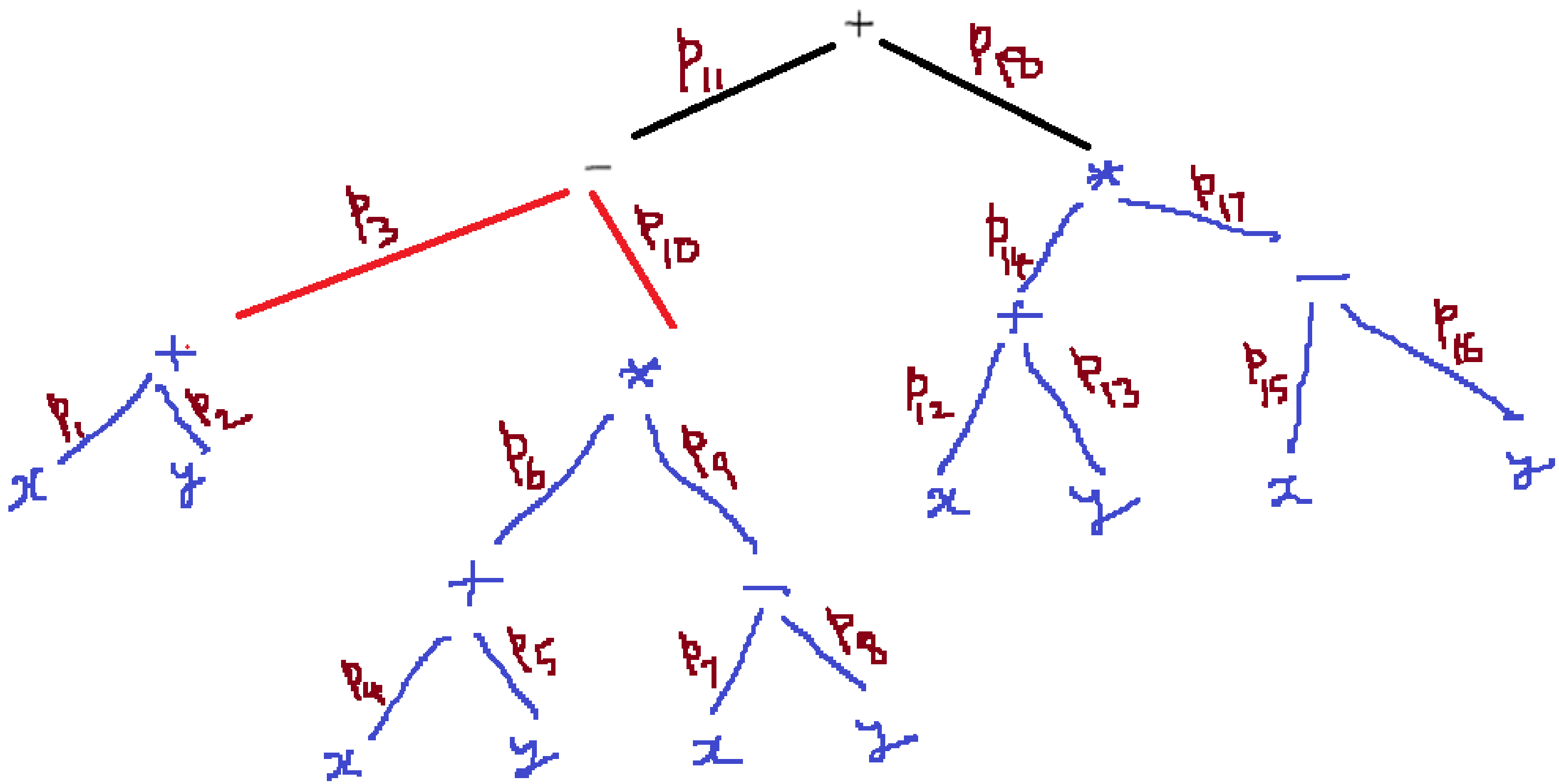
Exercise 6.1.1: Construct the DAG for the expression

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$



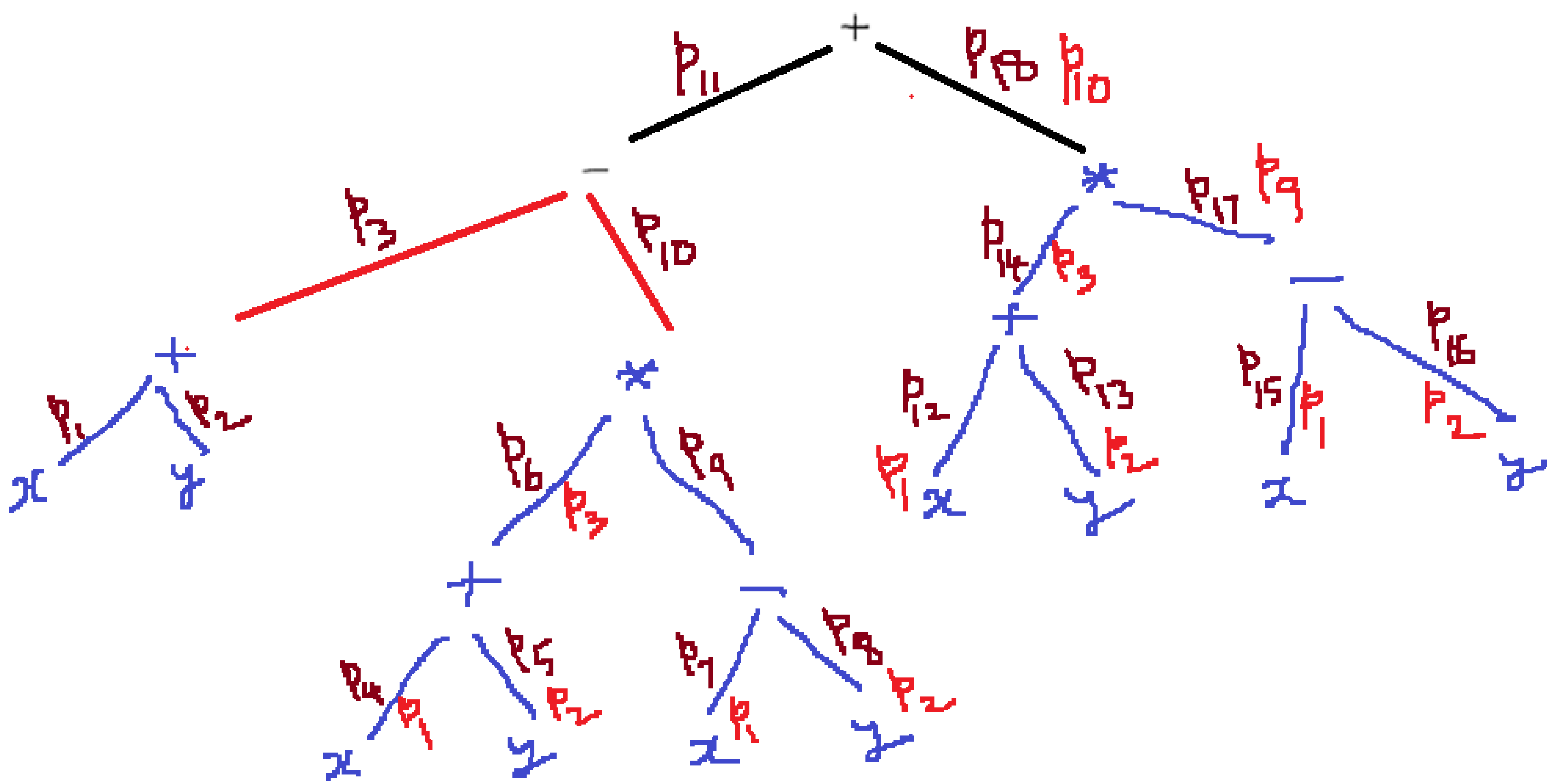
Exercise 6.1.1: Construct the DAG for the expression

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$



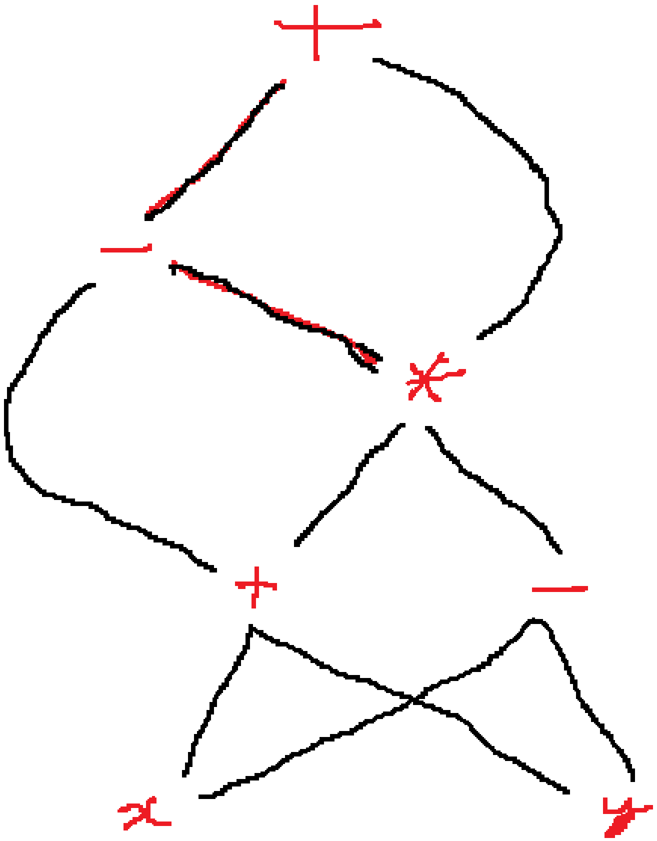
Exercise 6.1.1: Construct the DAG for the expression

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$



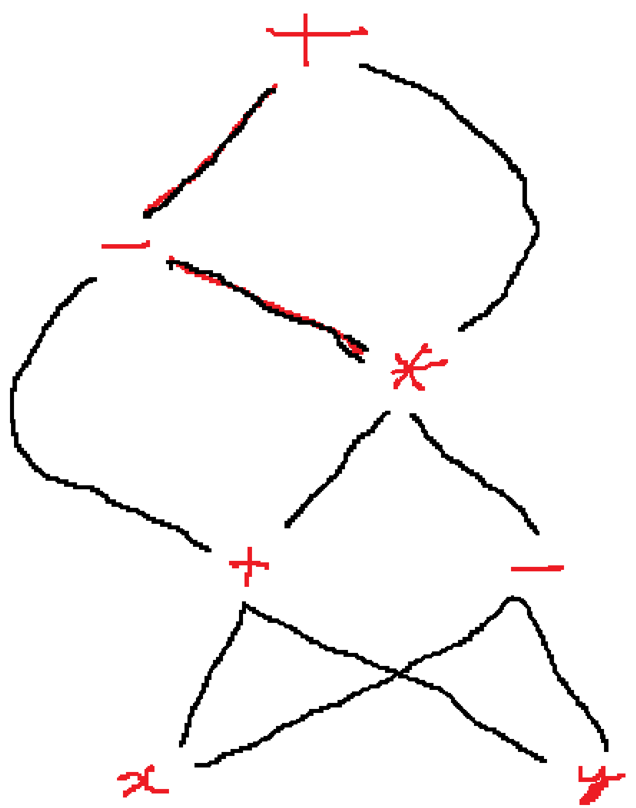
Exercise 6.1.1: Construct the DAG for the expression

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$



Exercise 6.1.1: Construct the DAG for the expression

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$



$$t1 = x + y$$
$$t2 = x - y$$
$$t3 = t1 * t2$$
$$t4 = t1 - t3$$
$$t5 = t4 + t3$$

Three address code

Can you find three address codes, after DAGs?

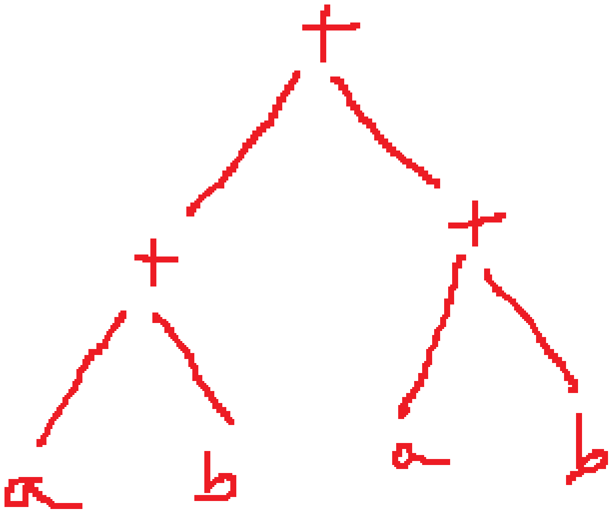
Exercise 6.1.2: Construct the DAG and identify the value numbers for the subexpressions of the following expressions, assuming + associates from the left.

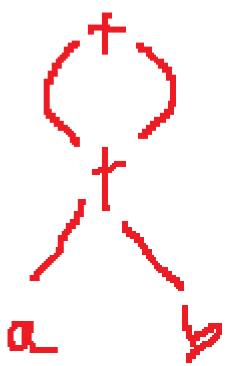
a) $a + b + (a + b)$.

b) $a + b + a + b$.

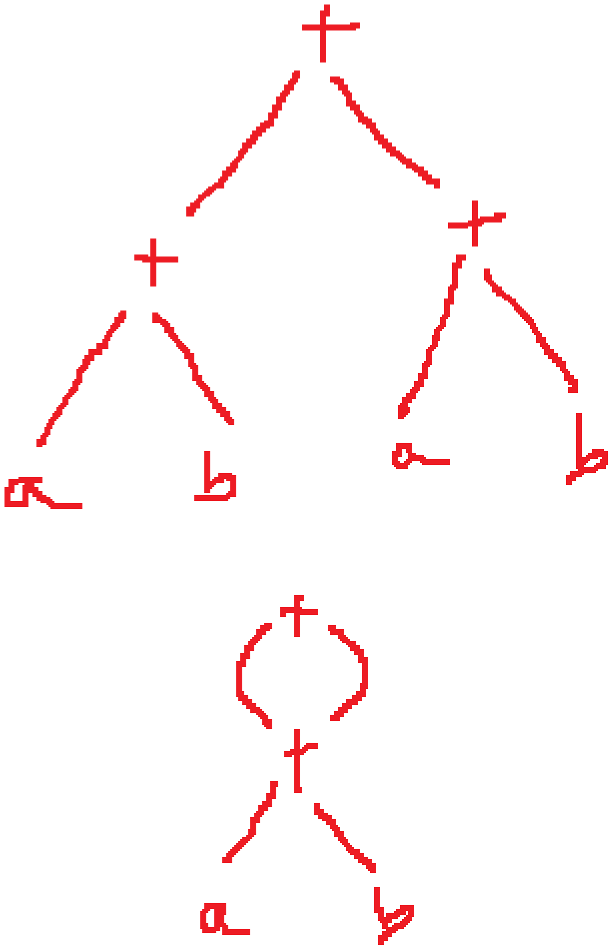
c) $a + a + \big((a + a + a + (a + a + a + a)\big)$.

a) $a + b + (a + b)$.

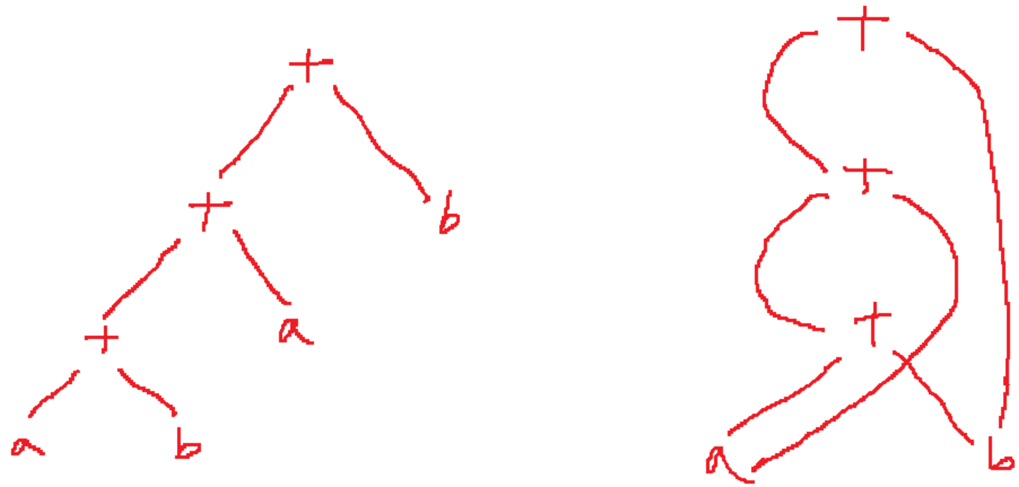




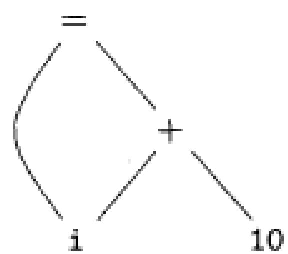
a) $a + b + (a + b)$.



b) $a + b + a + b$.



6.1.2 The Value-Number Method for Constructing DAG's



(a) DAG

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

(b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

Value-number of a node. The array index is seen as a pointer and this is traditionally called value-number of the node. An array element contains a node.

- Hashing can be used to search this array efficiently.

Algorithm for node insertion in value-number method.

Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

Algorithm for node insertion in value-number method.

Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

INPUT: Label op, node l, and node r.

OUTPUT: The value number of a node in the array with signature $\langle op, l, r \rangle$.

Algorithm for node insertion in value-number method.

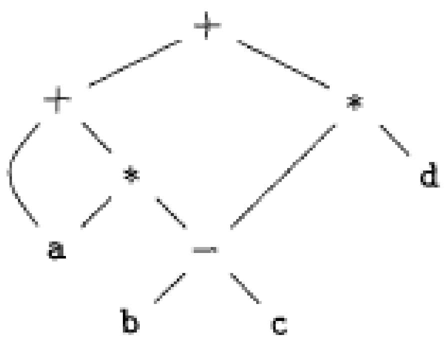
Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

INPUT: Label op, node l, and node r.

OUTPUT: The value number of a node in the array with signature $\langle op, l, r \rangle$.

METHOD: Search the array for a node M with label op, left child l, and right child r. If there is such a node, return the value number of M. If not, create in the array a new node N with label op, left child l, and right child r, and return its value number. □

Example 6.4: Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.



(a) DAG

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

(b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

6.2.1 Addresses and Instructions

An address can be one of the following:

- *A name.* For convenience, we allow source-program names to appear as addresses in three-address code.

  In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

- *A constant.*

- *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed.

Common Three address instructions

1. Assignment instructions of the form $x = y \ op \ z$, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.

Common Three address instructions

1. Assignment instructions of the form $x = y \; op \; z$, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \; y$, where op is a unary operation.

Common Three address instructions

1. Assignment instructions of the form $x = y\ op\ z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op\ y$, where *op* is a unary operation.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

Common Three address instructions

1. Assignment instructions of the form $x = y \ op \ z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \ y$, where *op* is a unary operation.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

Common Three address instructions

1. Assignment instructions of the form $x = y \; op \; z$, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \; y$, where op is a unary operation.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (`<`, `==`, `>=`, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y.

7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y = `call` p, n for procedure and function calls, respectively; and `return` y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$.

param x_1	param x_1	param x_1
param x_2	param x_2	param x_2
...	...	...
param x_n	param x_n	param x_n
call p, n	y = call p, n	call p, n
		return y
Procedure call	Function call	Function call

8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$.

$x = y[i]$

x is assigned with value which is i memory locations away from address y.
The i is a plain number, usually indicating number of bytes (doesnot be having intrinsic dependency on type of y as in C)

9. Address and pointer assignments of the form $x = \& y$, $x = * y$, and $* x = y$.

Example 6.5: Consider the statement

```
do i = i+1; while (a[i] < v);
```

(a) Symbolic labels.		(b) Position numbers.	
L:	t_1 = i + 1	100:	t_1 = i + 1
	i = t_1	101:	i = t_1
	t_2 = i * 8	102:	t_2 = i * 8
	t_3 = a [t_2]	103:	t_3 = a [t_2]
	if t_3 < v goto L	104:	if t_3 < v goto 100

Figure 6.9: Two ways of assigning labels to three-address statements

Note the number 8 above, which is the number of bytes that one position means in i + 1

Representations of Three address code

- Quadruples
- Triples
- Indirect triples

6.2.2 Quadruples

$$
\begin{array}{lcl}
\mathtt{t}_1 & = & \mathtt{minus\ c} \\
\mathtt{t}_2 & = & \mathtt{b} * \mathtt{t}_1 \\
\mathtt{t}_3 & = & \mathtt{minus\ c} \\
\mathtt{t}_4 & = & \mathtt{b} * \mathtt{t}_3 \\
\mathtt{t}_5 & = & \mathtt{t}_2 + \mathtt{t}_4 \\
\mathtt{a} & = & \mathtt{t}_5
\end{array}
$$

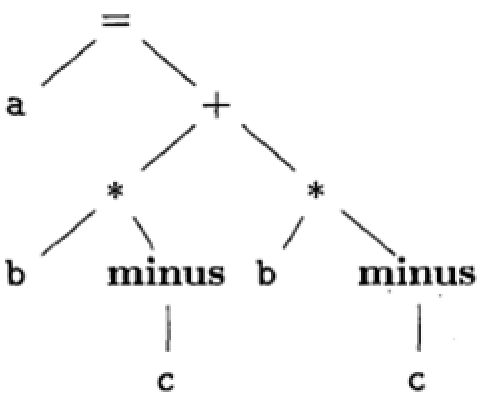
(a) Three-address code

	op	arg_1	arg_2	*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
	...			

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

6.2.3 Triples



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

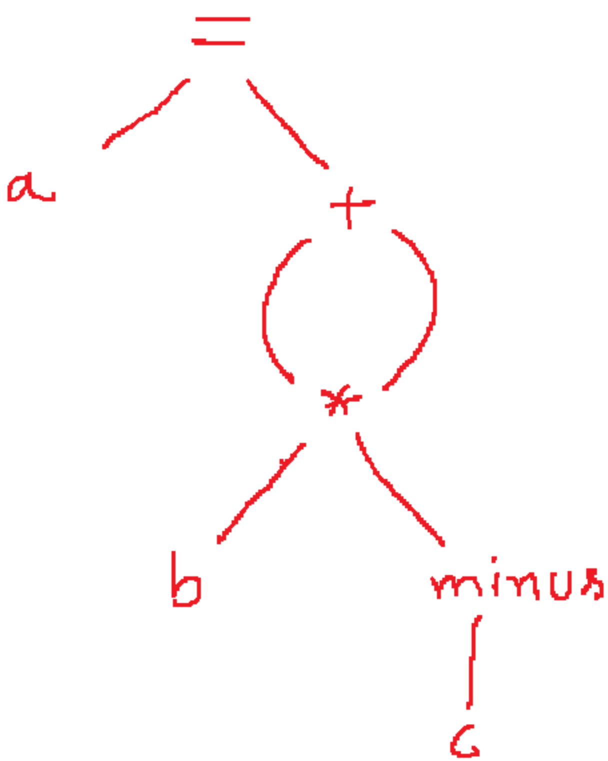
(b) Triples

This is
a = result of (4).

$a = t_5$

Representations of a = b * - c + b * - c

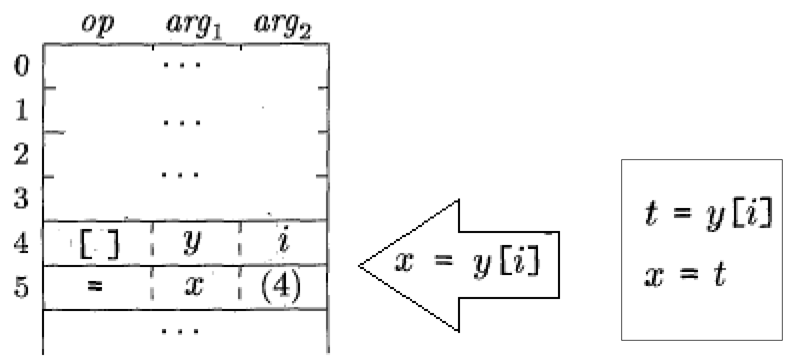
DAG based Triples



	op	arg1	arg 2
0	minus	c	
1	*	b	(0)
2	+	(1)	(1)
3	=	a	(2)

Triples for $x = y[i]$

- This is a ternary operation.



Triples

- Similarly for $x[i] = y$

Indirect triples

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

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

Figure 6.12: Indirect triples representation of three-address code

- Optimizer can reorder the instructions without changing the triples itself.
  - Java does this

Static Single-Assignment Form

- An IR (Intermediate representation) form
- Useful in optimization {covered later}
- Idea: Use various versions of a variable (each with a distinct name)
- When time comes to use the variable, apply the merge operation to get the exact version that should be used.

- ```
  y := 1
  y := 2
  x := y
  ```

- $y_1 := 1$
  $y_2 := 2$
  $x_1 := y_2$

It may be easy to find that $y_1 \coloneqq 1$ is useless, hence can be removed.
  ```

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

$$
\begin{aligned}
p_1 &= a + b \\
q_1 &= p_1 - c \\
p_2 &= q_1 * d \\
p_3 &= e - p_2 \\
q_2 &= p_3 + q_1
\end{aligned}
$$

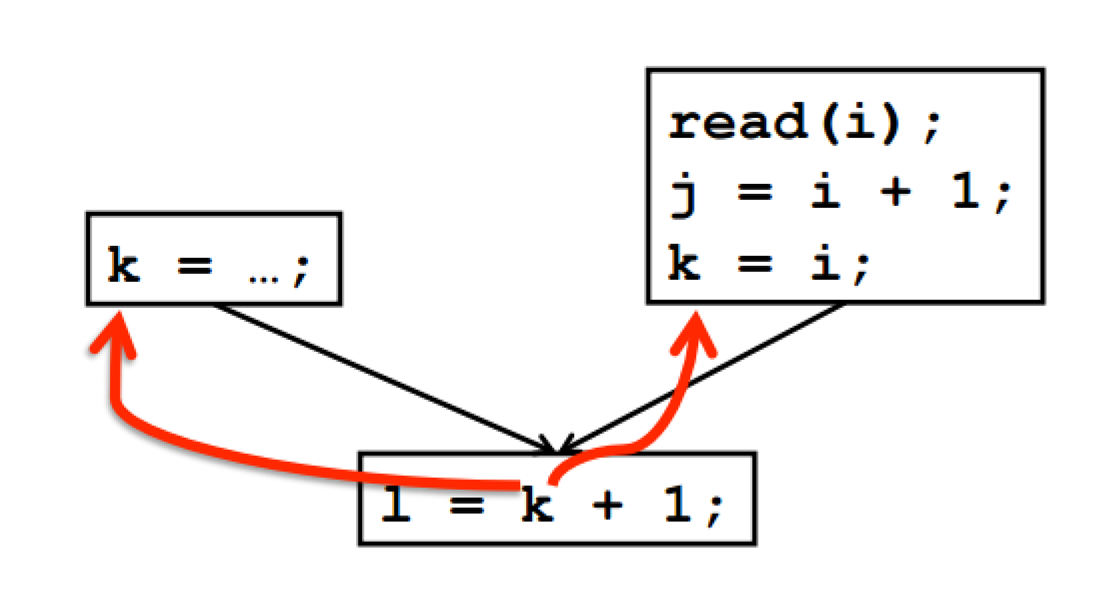
(b) Static single-assignment form.

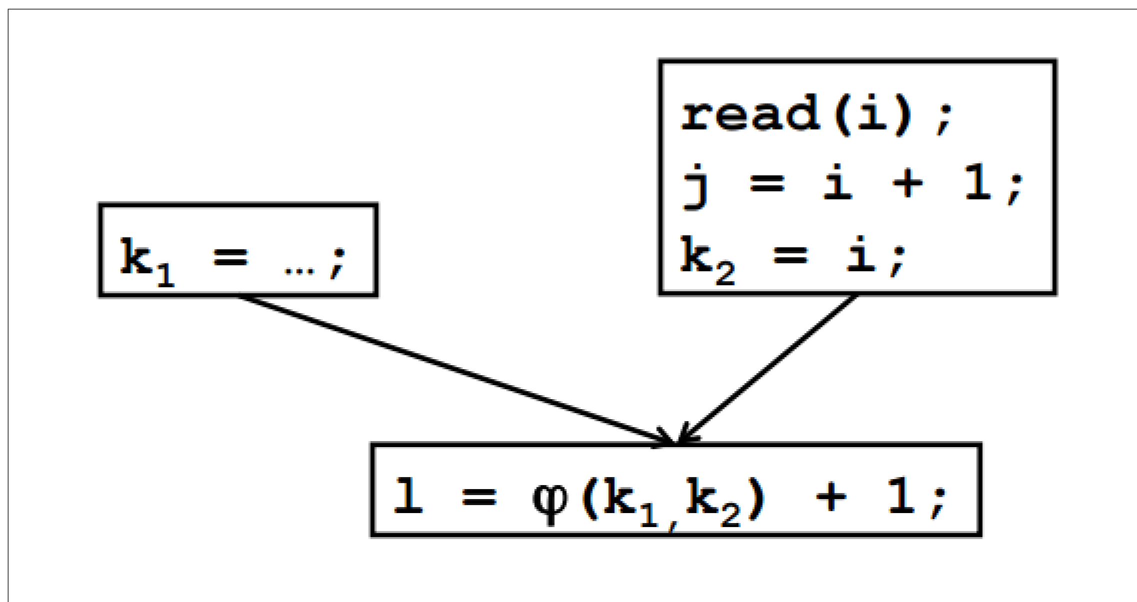
```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

→

$\texttt{if (flag) } x_1 = -1\texttt{; else } x_2 = 1;$
$x_3 = \phi(x_1, x_2);$
$y = x_3 * a;$

Data flow analysis can tell us about the **merge** function, i.e., $\phi(x_1, x_2)$.


k = ...;
read(i);
j = i + 1;
k = i;
l = k + 1;


k_1 = ...;
read(i);
j = i + 1;
k_2 = i;
l = $\varphi(k_1, k_2)$ + 1;

Exercise 6.2.1: Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.

b) Quadruples.

c) Triples.

d) Indirect triples.

6.3 Types and Declarations

- Type checking uses logical rules to reason about the behavior of a program at run time.
  - Types of operands should match.
  - Relational operator requires Boolean operands.
- Type → storage required at run time.
  - Type also says how to do address arithmetic in array indexing.
  - When to do explicit type conversion.

Type Expressions

- Basic types
- Type constructor is used to create complicated types
  - Classes
  - Structures
  - Pointers to arrays, array of pointers, etc

Type

- In C, int [2][3] is a type ➔it is an array of two elements, where each element is an array of 3 integers.
- The corresponding type expression is array(2,array(3,integer)). Second argument is a type, first argument is the number of elements.

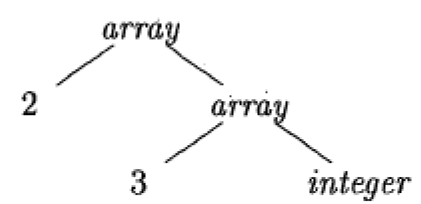


Figure 5.15: Type expression for **int**[2][3]

Type expression, inductively defined

- Basic type is a type expression.
- A type name is a type expression.
- array(number, type expression).
- record(type exp1, type exp2, …).
- type expression →type expression. s→t is type for a function (function from type s to type t).
- if s and t are type expressions, their cartesian product s X t is a type expression. For tuples (records).

- Type expressions may contain variables whose values are type expressions. At run time variables can contain different values!

- **Type expressions can be created using SDTs and can be represented as DAGs. Just like for any other expression.**

RECALL the following…(we saw in SDTs)

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

For input **int[2][3]**

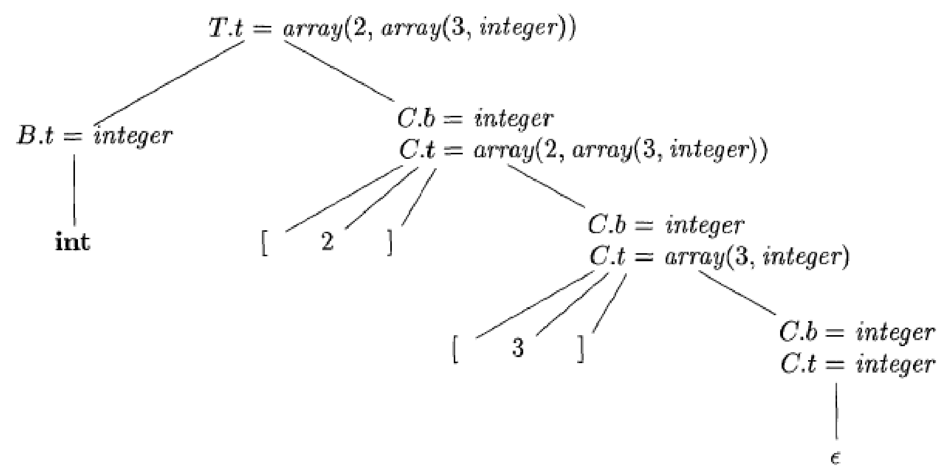


Figure 5.17: Syntax-directed translation of array types

6.3.2 Type Equivalence

- Often, we want to verify whether two types are equivalent or not (error has to be given).
- Operations are defined on same/particular typed variables/values
  - So type checking is a must
- Ambiguity arise when names are given to type expressions, which in turn are used in creating new types.

Two types are structurally equivalent ..

- If and only if one of the following 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.

- First two conditions define *name equivalence* of type expressions.

6.3.3 Declarations

- Grammar for declaring one name at a time.
- Basic type, structure/record type, array type.

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id}\ ;\ D\ \mid\ \epsilon \\
T & \rightarrow & B\ C\ \mid\ \textbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \textbf{int}\ \mid\ \textbf{float} \\
C & \rightarrow & \epsilon\ \mid\ [\ \textbf{num}\ ]\ C
\end{array}
$$

- $D \Rightarrow T\ id; D \Rightarrow T\ id; \Rightarrow B\ C\ id; \Rightarrow int\ C\ id \Rightarrow int\ id;$

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D\ \mid\ \epsilon \\
T & \rightarrow & B\ C\ \mid\ \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int}\ \mid\ \mathbf{float} \\
C & \rightarrow & \epsilon\ \mid\ [\ \mathbf{num}\ ]\ C
\end{array}
$$

- $D \Rightarrow T\ id; D \Rightarrow T\ id; \Rightarrow B\ C\ id; \Rightarrow int\ C\ id \Rightarrow int\ id;$

- $D \Rightarrow T\ id; D \Rightarrow T\ id; \Rightarrow record\ '\{'D'\}'\ id;$

$$\Rightarrow record\ '\{'\ T\ id;\ D\ '\}'\ id;$$

$$\ldots$$

$$\Rightarrow record\ '\{'int\ \ id; float\ id;\ '\}'id;$$

6.3.4 Storage Layout for Local Names

- From the type of a variable we can determine the storage needed at compile time,
  - except for few types, like strings, dynamic arrays,..
- Static & dynamic memory are needed
- Relative addresses of variables whose storage requirements are known, can be calculated at compile time
  - This can be stored in the symbol table.

Address Alignment

- Padding is often used and will waste memory to align addresses.
  - 10 character string is stored in 12 bytes (since each word is of 4 bytes)

SDT for type and width

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

Figure 6.15: Computing types and their widths

SDT for type and width

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

Figure 6.15: Computing types and their widths

t and *w* are local variables used by the SDT. This simplifies.

For int[2][3]

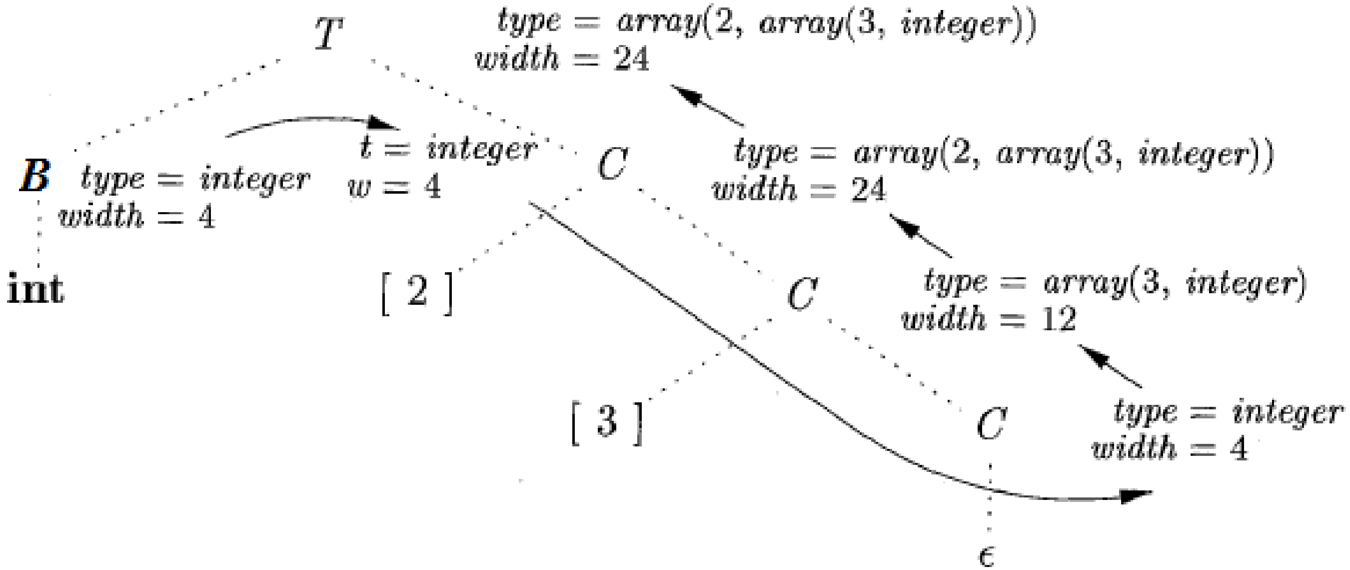


Figure 6.16: Syntax-directed translation of array types

6.3.5 Sequences of Declarations

$$P \rightarrow D$$

$$D \rightarrow T \ \mathbf{id} \ ; \ D_1$$

$$D \rightarrow \epsilon$$

6.3.5 Sequences of Declarations

- Relative addresses are kept track with a variable *offset* in the following SDD.

A local variable to keep track of relative address for the next variable.

$$
\begin{array}{lll}
P & \rightarrow & \{\ \mathit{offset} = 0;\ \}\ D \\[1em]
D & \rightarrow & T\ \mathbf{id}\ ; \quad \{\ \mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset}); \\
 & & \qquad\qquad\quad \mathit{offset} = \mathit{offset} + T.\mathit{width};\ \} \\
 & & D_1 \\
D & \rightarrow & \epsilon
\end{array}
$$

Figure 6.17: Computing the relative addresses of declared names

$\mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset})$ creates a symboltable entry
Here *top* denotes the current symbol table.

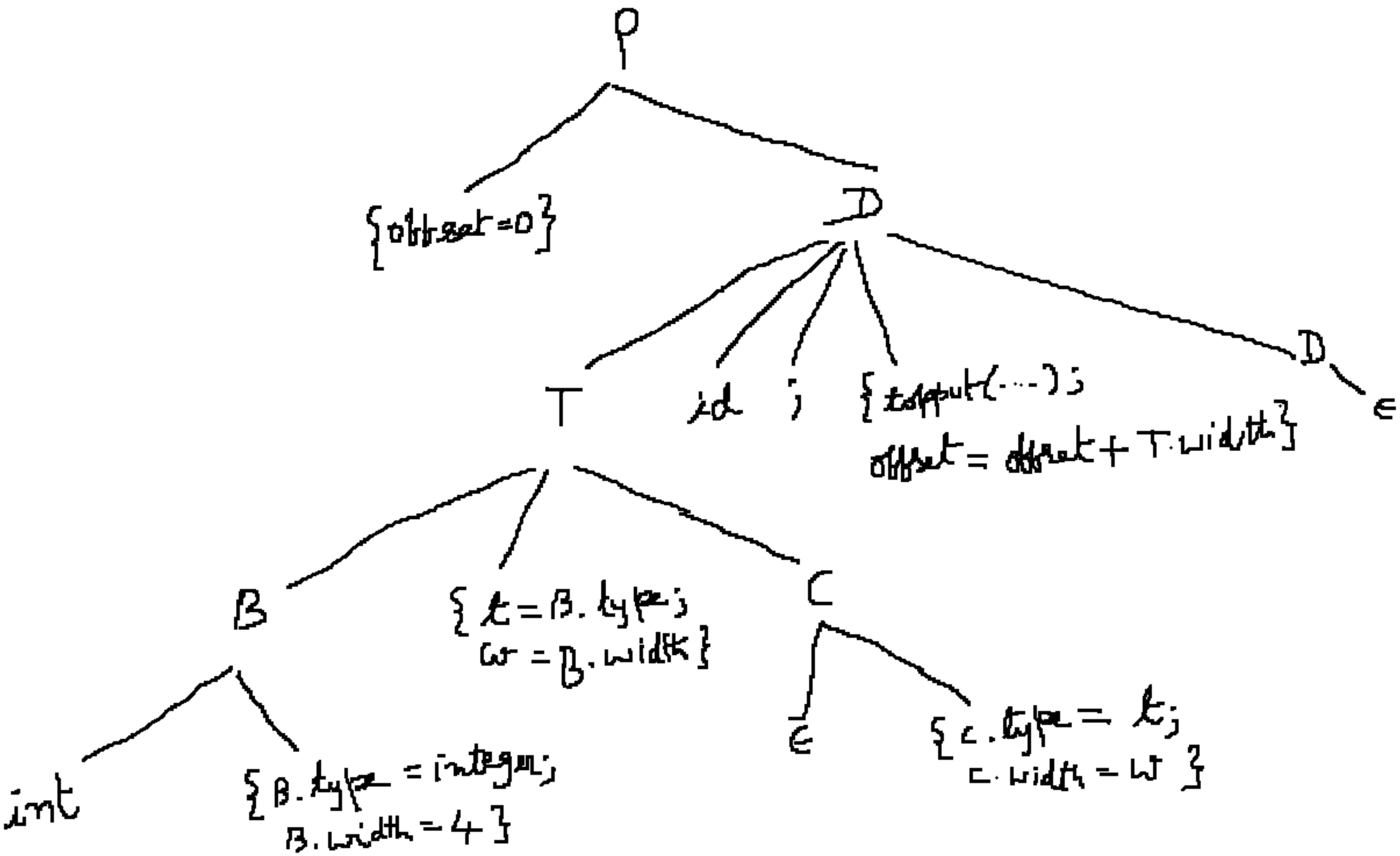
Fields in Records and Classes

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

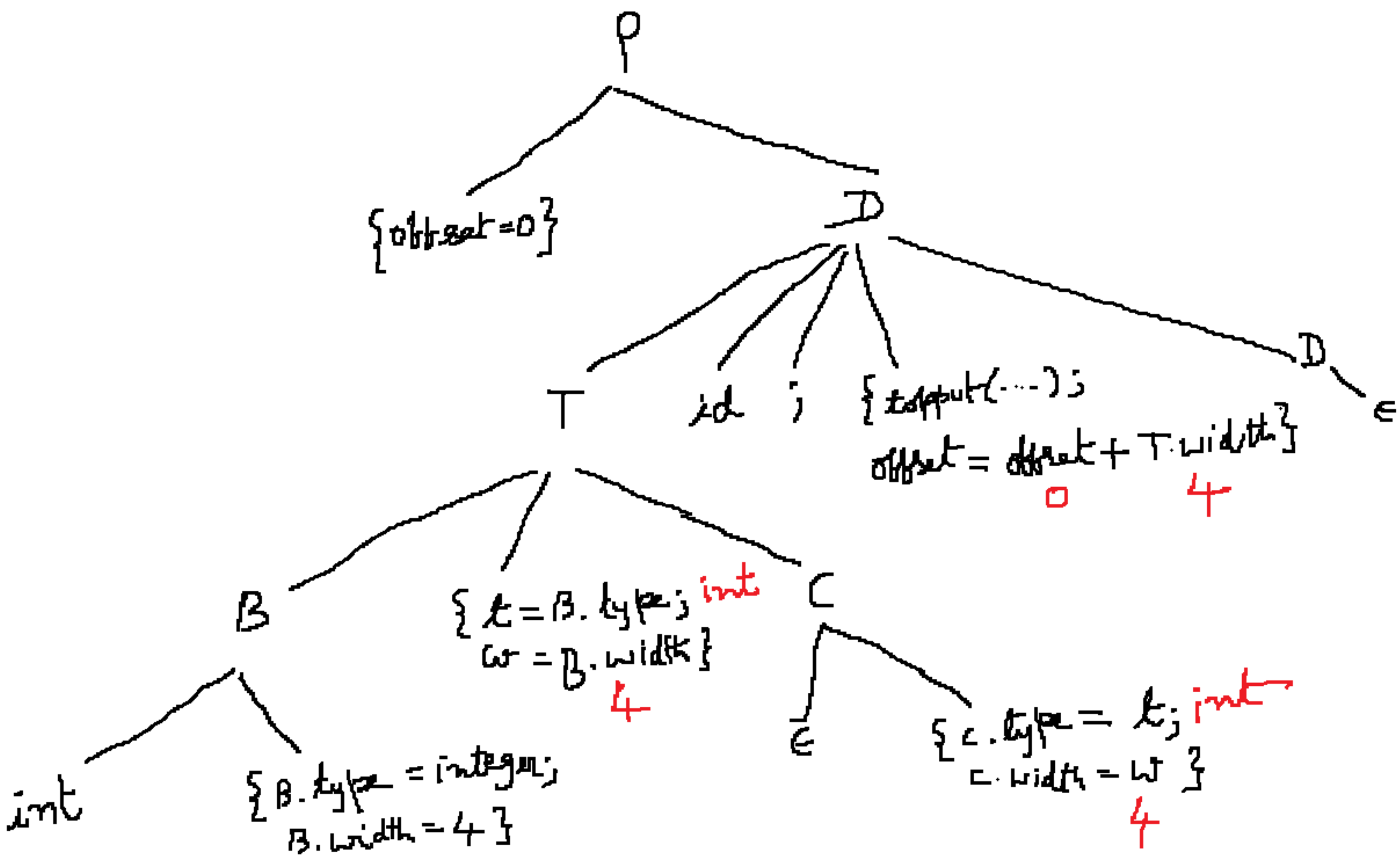
$$T \rightarrow \textbf{record} \; '\{' \; D \; '\}'$$

This is added

int id;



P
{offset=0}
D
T
id
;
{top.put(...);
offset = offset + T.width}
D
ϵ
B
{t = B.type;
w = B.width}
C
int
{B.type = integer;
B.width = 4}
ϵ
{c.type = t;
c.width = w}


P
{offset=0}
D
T
id
;
{top.put(....);
offset = offset + T.width}
0
4
D
∈
B
{t = B.type; int
w = B.width}
4
C
int
{B.type = integer;
B.width = 4}
∈
{c.type = t; int
c.width = w}
4

- The field names within a record must be distinct; that is, a name may appear at most once in the declarations generated by D. Static checking.

- The offset or relative address for a field name is relative to the data area for that record.

Example 6.10: The use of a name x for a field within a record does not conflict with other uses of the name outside the record. Thus, the three uses of x in the following declarations are distinct and do not conflict with each other:

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

A subsequent assignment `x = p.x + q.x;` sets variable x to the sum of the fields named x in the records p and q. Note that the relative address of x in p differs from the relative address of x in q. □