Implementation of Three-Address Statements

- A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands.
- Two such representations are quadruples and triples.

Quadruples

- A quadruple is a record structure with four fields, which we call 'op', 'arg1', 'arg2', and 'result'.
- The 'op' field contains an internal code for the operator.
- The three-address instruction $x = y + z$ is represented by placing + in 'op', y in 'arg1', z in 'arg2', and x in 'result'.
- Instructions with unary operators like 'x = uminus y' or '$x = y$' do not use arg2. Note that for a copy statement '$x = y$', op is =, while for most other operations, the assignment operator is implied.
- Operators like 'param' use neither 'arg2' nor 'result'.
- Conditional and unconditional jumps put the target label in 'result'.
- Example: Three-address code for the assignment $a = b * -c + b * -c$ is given below:

	op	arg1	arg2	result
0	uminus	c		t1
1	*	b	t1	t2
2	uminus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

- For readability, we use actual identifiers like a, b, and c in the fields arg1, arg2, and result instead of pointers to their symbol table entries. Here, temporary names are entered into the symbol table.

Triples

- A 'triple' has only three fields, which we call 'op', 'arg1', and 'arg2'. Note that the 'result' field in a quadruple is used primarily for temporary names. Using triples, we refer to the result of an operation 'x op y' by its position, rather than by an explicit temporary name. Thus, instead of the temporary t1 in the quadruple, a triple representation would refer to its position, like (0). Parenthesized numbers represent pointers into the triple structure itself.
- Example: The triple representation for the assignment $a = b * -c + b * -c$ is given below.

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

- In the triple representation, the copy statement $a = t5$ is encoded by placing 'a' in arg1 field and (4) in arg2 field.
- The fields 'arg1' and 'arg2' for the arguments of 'op' are either pointers to the symbol table (for programmer defined names or constants) or pointers into the triple structure (for temporary values).

- An operation like $x[i] = y$ requires two entries in the triple as given below:

	op	arg1	arg2
(0)	[]=	x	i
(1)	=	(0)	y

(a) $x[i] = y$

	op	arg1	arg2
(0)	=[]	y	i
(1)	=	x	(0)

(b) $x = y[i]$

[Page 328 included]

Declarations : Processing Addresses. /* Read section 5.4.3, ALSO also. */

- Languages such as C allow all the declarations in a single procedure to be processed as a group. Therefore, we can use a variable, say 'offset', to keep track of the next available relative address.
- Consider the following grammar for declarations:

$P \rightarrow D S$

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

$T \rightarrow int \mid float$

- The translation scheme given below deals with a sequence of declarations of the form "T id".

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

$D \rightarrow T \; id \; ; \; \{ addentry(id.lexeme, T.type, offset);$
$offset = offset + T.width; \}$

D_1

$D \rightarrow \epsilon$

$T \rightarrow int \quad \{ T.type = integer; \; T.width = 4; \}$

$T \rightarrow float \quad \{ T.type = float; \; T.width = 8; \}$.

- Before the first declaration is considered, 'offset' is set to 0. As each new name x is seen, x is entered into the symbol table with its relative address set to the current value of 'offset', which is then incremented by the width of the type of x.
- Nonterminals generating ϵ, called marker nonterminals, can be used to rewrite productions so that all actions appear at the ends of right sides. The above example is rewritten using a marker nonterminal below:

$P \rightarrow M_1 D \; S$

$M_1 \rightarrow \epsilon \quad \{ offset = 0 \}$

$D \rightarrow T \; id \; ; \; M_2 \; D_1$

$M_2 \rightarrow \epsilon \quad \{ addentry(id.lexeme, T.type, offset;$
$offset = offset + T.width; \}$

Page 328, ALSU

Step 1. Produce parse tree by ignoring the actions
2. Add "actions" to parse tree
3. Perform a preorder traversal, and do action accordingly.

- Example.

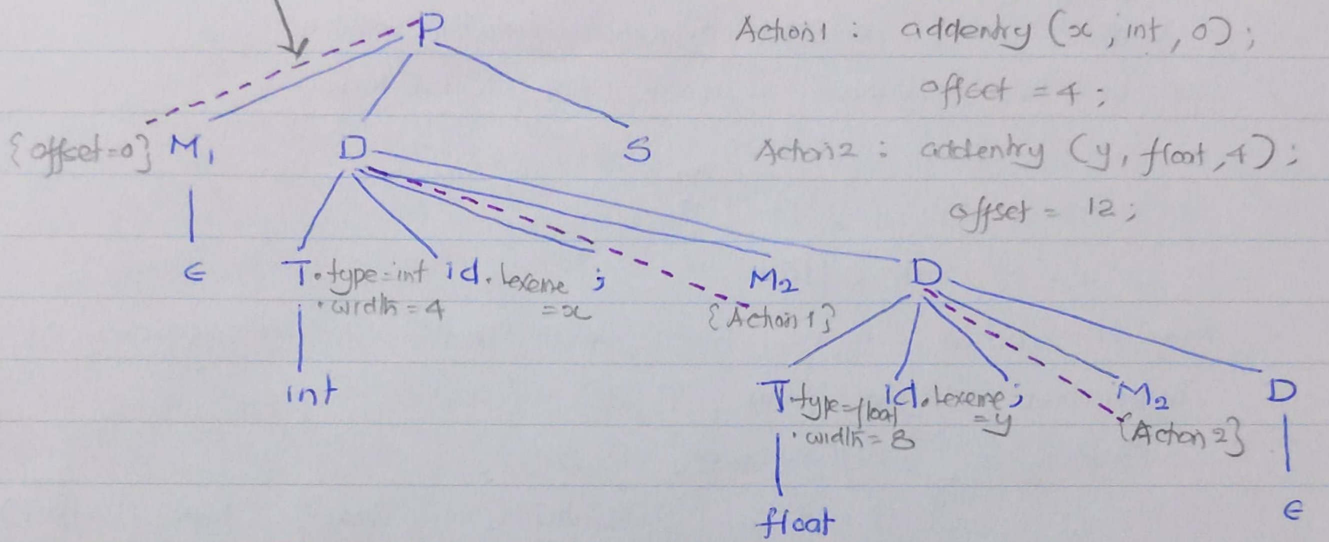
  int x ;

  float y ;

/* Processing is done by 'preorder' traversal */

Action 1 : addentry (x, int, 0);
offset = 4;

Action 2 : addentry (y, float, 4);
offset = 12;



- Arrays

  T → array [num] of T_1 { T.type = array (num.val, T_1.type);
  T.width = num.val * T_1.width; }

- Pointers

  T → pointer T_1 { T.type = pointer (T_1.type);
  T.width = 4; }.

- Records

  T → record '{' D '}' { T.type = record (t); T.width = offset; }

  - Record type may be handled by having a separate symbol table for it.
  - A record type has the form 'record (t)', where 'record' is the type constructor, and 't' is a symbol table ~~object~~ that holds information about the fields of this record type.
  - T.width is the value of 'offset' after processing D.