CS3501 COMPILER DESIGN

UNIT III SYNTAX DIRECTED TRANSLATION & INTERMEDIATE CODE GENERATION

- Syntax directed Definitions-Construction of Syntax Tree-Bottom-up Evaluation of S-Attribute Definitions- Design of predictive translator - Type Systems-Specification of a simple type Checker- Equivalence of Type Expressions-Type Conversions. Intermediate Languages: Syntax Tree, Three Address Code, Types and Declarations, Translation of Expressions, Type Checking, Back patching.

UNIT IV RUN-TIME ENVIRONMENT AND CODE GENERATION

- Runtime Environments – source language issues – Storage organization – Storage Allocation Strategies: Static, Stack and Heap allocation - Parameter Passing-Symbol Tables - Dynamic Storage Allocation - Issues in the Design of a code generator – Basic Blocks and Flow graphs - Design of a simple Code Generator - Optimal Code Generation for Expressions– Dynamic Programming Code Generation

UNIT V CODE OPTIMIZATION

- Principal Sources of Optimization – Peep-hole optimization - DAG-Optimization of Basic Blocks - Global Data Flow Analysis - Efficient Data Flow Algorithm – Recent trends in Compiler Design.

COURSE OUTCOMES

- **CO1:**Understand the techniques in different phases of a compiler.
- **CO2:**Design a lexical analyser for a sample language and learn to use the LEX tool.
- **CO3:**Apply different parsing algorithms to develop a parser and learn to use YACC tool
- **CO4:**Understand semantics rules (SDT), intermediate code generation and run-time environment.
- **CO5:**Implement code generation and apply code optimization techniques.

BOOKS

TEXT BOOK:

- 1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques and Tools”, Second Edition, Pearson Education, 2009.

REFERENCES

- 1. Randy Allen, Ken Kennedy, Optimizing Compilers for Modern Architectures: A Dependence based Approach, Morgan Kaufmann Publishers, 2002.
- 2. Steven S. Muchnick, Advanced Compiler Design and Implementationǁ, Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint 2003.
- 3. Keith D Cooper and Linda Torczon, Engineering a Compilerǁ, Morgan Kaufmann Publishers Elsevier Science, 2004.
- 4. V. Raghavan, Principles of Compiler Designǁ, Tata McGraw Hill Education Publishers, 2010.
- 5. Allen I. Holub, Compiler Design in Cǁ, Prentice-Hall Software Series, 1993.

UNIT III SYNTAX DIRECTED TRANSLATION & INTERMEDIATE CODE GENERATION

OUTLINE

- ✔Syntax Directed Definitions
- ✔Annotated Parse Tree
- ✔Synthesized attributes
- ✔Inherited Attributes
- ✔Evaluation Orders for SDD's

Syntax-Directed Translation

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Two notations for associating semantic rules with productions,

  Syntax - directed definitions and Translation schemes.

  Syntax - directed definitions: high level specifications for translations

  Translation schemes: Indicate the order in which semantic rules are to be evaluated.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities.

Syntax-Directed Translation

- A syntax-directed definition (SDD) is a context-free grammar together with, attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions
- A syntax-directed definition is a generalization of a context-free grammar in which:
  - Each grammar symbol is associated with a **set of attributes**.
  - **synthesized** and **inherited** attributes of that grammar symbol.
- An attribute can represent any thing.
  - a string, a number, a type, a memory location, a complex record.
- The value of an attribute at a parse – tree node is defined by a semantic rule associated with the production used at that node.
- The value of a **synthesized attribute** at a node is computed from the values of attribute at the children of that node in the parse tree .
- The value of an **inherited attribute** is computed from the values of attributes at the siblings and parent of that node.

Syntax-Directed Definitions

- *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.
- *Dependency graph* determines the evaluation order.
- Evaluation of a semantic rule defines the value of an attribute.
- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** of the parse tree.

Form of a Syntax-Directed Definition

In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$b := f(c_1, c_2, \ldots, c_n)$

where *f* is a function,

and *b* can be one of the followings:

- *b* is a synthesized attribute of A and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

OR

- *b* is an inherited attribute of one of the grammar symbols in α (on the right side of the production), and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

Syntax-Directed Definition - synthesized attributes

A SDD that uses synthesized attributes is said to be an S - Attributed definition

A parse tree for an S - Attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node bottom up

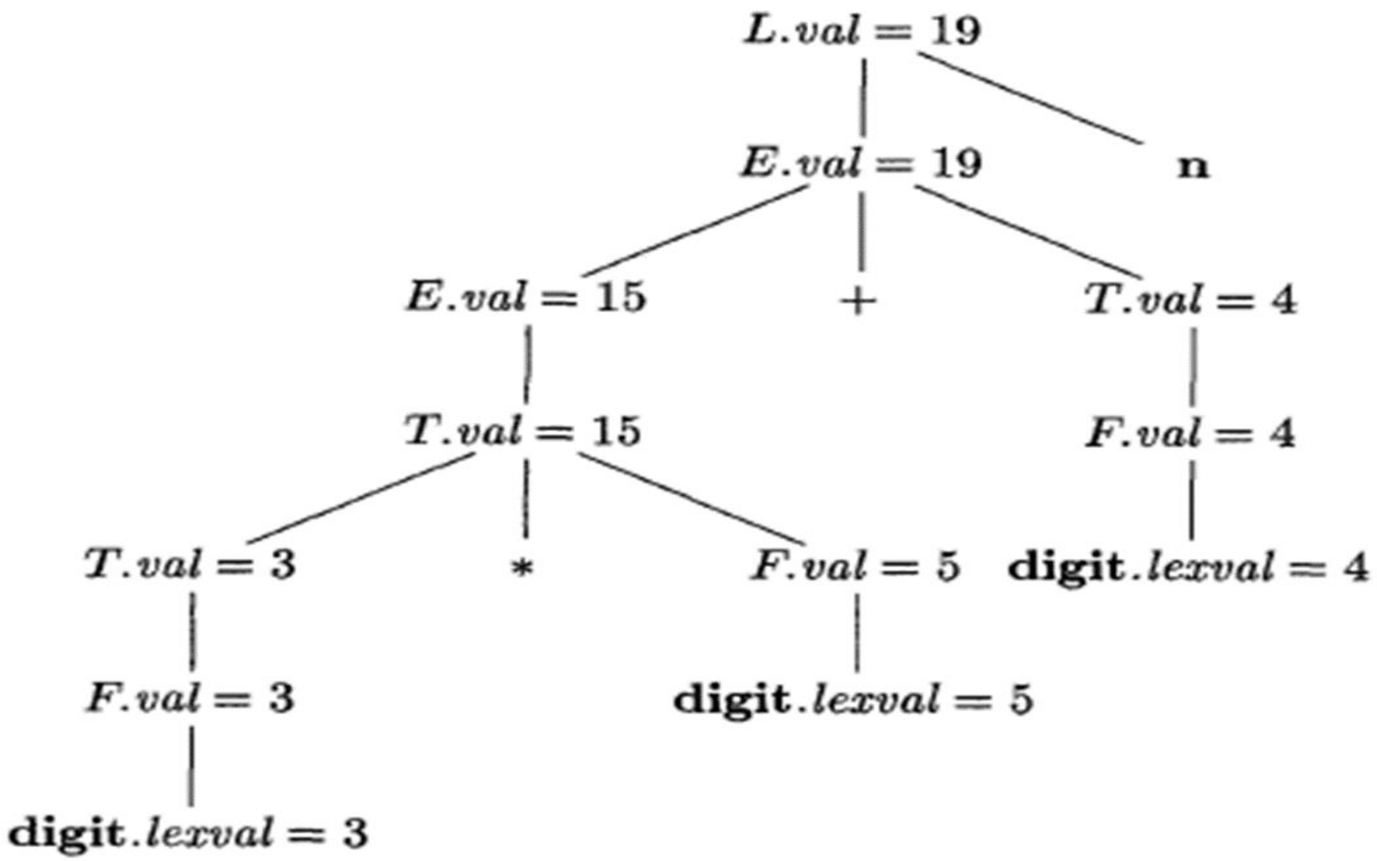
Syntax-Directed Definition of a simple desk calculator

Production	**Semantic Rules**
$L \rightarrow E$ **n**	L.val = E.val
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ **digit**	F.val = **digit**.lexval

Syntax-Directed Definition - synthesized attributes

- It evaluates expressions terminated by an end marker n.
- In the SDD, each of the non terminals has a single synthesized attribute, called val.
- The terminal digit has a synthesized attribute lexval, which is an integer value returned by the lexical analyzer
- The rule for production 1, L → En, sets L.val to E.va1, which we shall see is the numerical value of the entire expression.
- Production 2, E → E1+ T, which computes the val attribute for the head E as the sum of the values at El and T. At any parse- tree node N labeled E, the value of val for E is the sum of the values of val at the children of node N labeled E and T.
- Production 3, E → T, has a single rule that defines the value of val for E to be the same as the value of val at the child for T.
- Production 4 is similar to the second production; its rule multiplies the values at the children instead of adding them.
- The rules for productions 5 and 6 copy values at a child, like that for the third production.
- Production 7 gives F.val the value of a digit, that is, the numerical value of the token digit that the lexical analyzer returned.

Annotated parse tree for 3 * 5 + 4 n



Syntax-Directed Definition – Inherited Attributes

- An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of that node.
- Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears

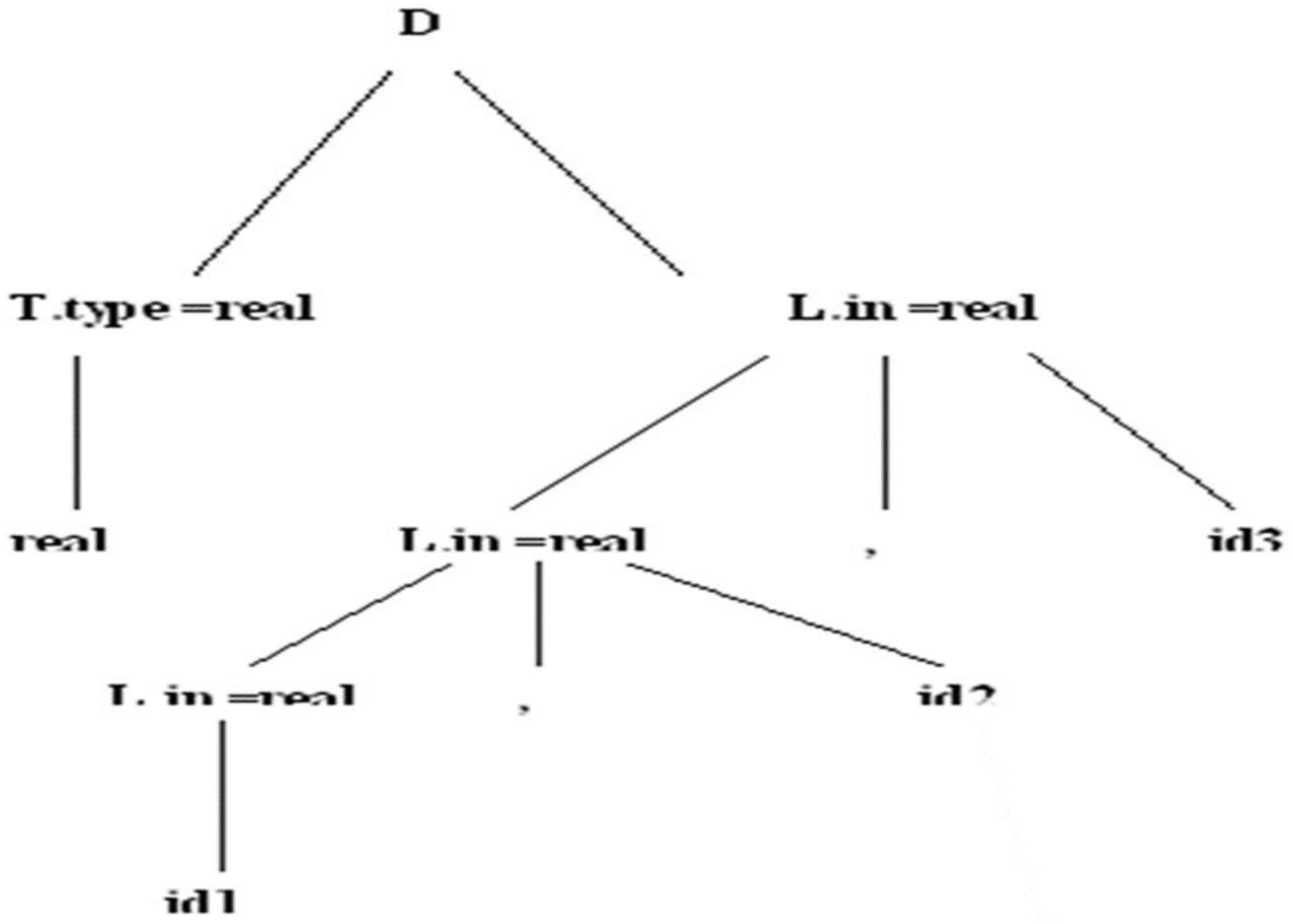
Syntax-directed definition with inherited attribute *L.in.*

Production	**Semantic Rules**
$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	T.type = real
$L \rightarrow L_1$, **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
$L \rightarrow$ **id**	addtype(**id**.entry,L.in)

Syntax-Directed Definition – Inherited Attributes

- A declaration generated by the non terminal D in the SDD consists of the keyword int or real, followed by a list of identifiers.
- The nonterminal T has a synthesized attribute type, whose value is determined by the keyword in the declaration.
- The semantic rule L.in := T. type , associated with production D → T L, sets inherited attribute L.in to the type in the declaration.
- The rules then pass this type down the parse tree using the inherited attribute L.in.
- Rules associated with the productions for L call procedure addtype to add the type of each identifier to its entry in the symbol table.

Annotated parse tree for the sentence real id1 , id2 , id3



- **A Different Alphabet**

Which letter comes next?

A C F J O __

- The next letter is **U.** The series moves forward by skipping 1 letter of the alphabet and then 2 letters and then 3 and so on.

What comes next in this series?

A, B, C, D, E ?

Hint: It’s not F.

- K. The letters represent vitamins, and there is currently no Vitamin F, G, H, I or J.

Evaluation Orders for SDD's

- "Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

Dependency graph

- Interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called Dependency graph

- Put each semantic rule in to the form *$b:=f(c_1,c_2,\ldots,c_n)$ by introducing a dummy* synthesized attribute b for each semantic rule that consists of a procedure call.

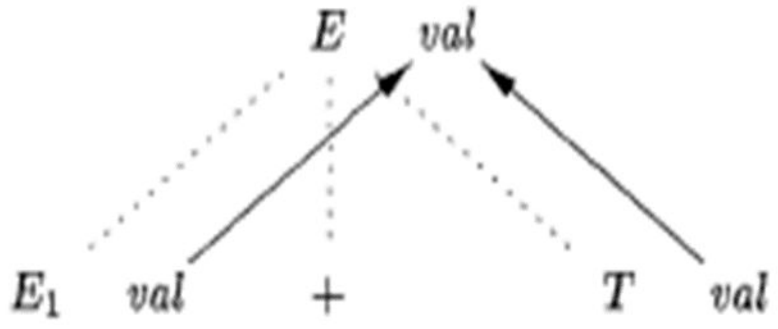
- Graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c

Dependency graph

- Whenever the following production is used in a parse tree, we add the edges to the dependency graph.

PRODUCTION	SEMANTIC RULE
E → E1+T	*E.val := E1.val + T.val*

- The three nodes of the dependency graph marked by . represent the synthesized attributes *E. val, E1. val,* and *E2. val* at the corresponding nodes in the parse tree. The edge to *E. val* from *E1. val* shows that *E. val* depends on *E1. val* and the edge to *E.val* from *E2.val* shows that *E.val* also depends on *E2.val.*



Ordering the Evaluation of Attributes

- ✔ If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N.

- ✔ Only allowable orders of evaluation are those sequences of nodes Nl, N2,. . . . , Nk such that if there is an edge of the dependency graph from Ni to Nj; then i < j. Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.

- ✔ If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort

S-Attributed Definitions

- ✔ Given an SDD, it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles.
- ✔ Translations can be implemented using classes of SDD's that guarantee an evaluation order, since they do not permit dependency graphs with cycles.
- ✔ Two classes of SDD's can be implemented efficiently in connection with top-down or bottom-up parsing.

S-Attributed Definitions

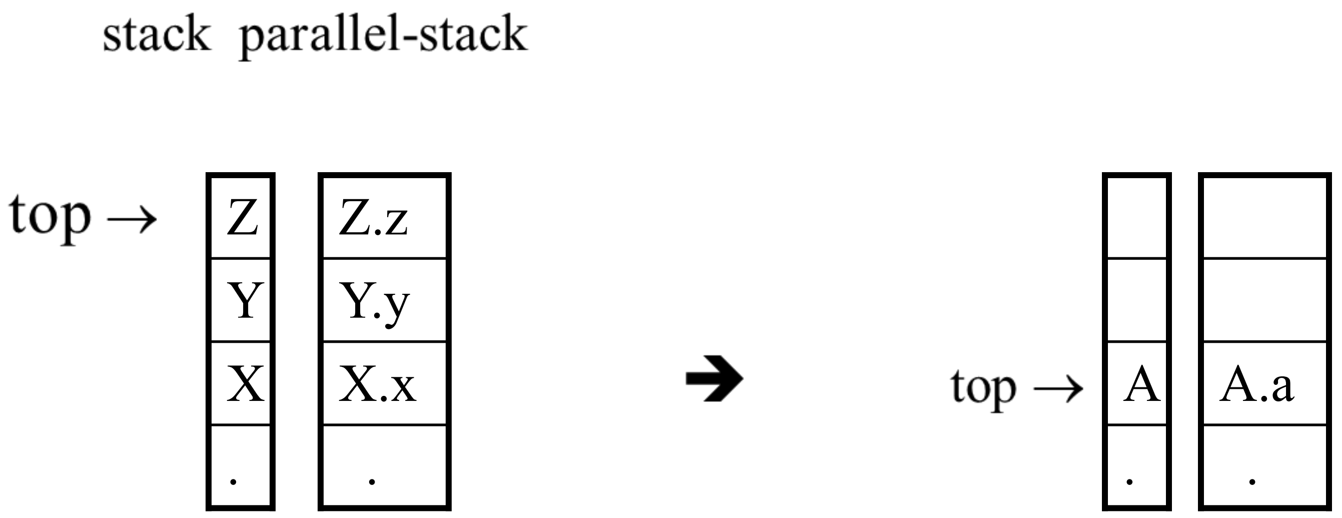
- ✔ We will look at two sub-classes of the syntax-directed definitions:
  - **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
  - **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.

- ✔ To implement S-Attributed Definitions and L-Attributed Definitions are easy (we can evaluate semantic rules in a single pass during the parsing).

- ✔ Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

Bottom-Up Evaluation of S-Attributed Definitions

- Put the values of the synthesized attributes of the grammar symbols into a parallel stack.
  - When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X.



- Which is correct to say, “The yolk of the egg is white” or “The yolk of the egg are white?”

- Neither. Egg yolks are yellow.

- By using only one straight line, can you make the equation correct. 5+5+5=550?

- To make the Equation True you can add a single line on the plus sign to make it 545. So, that when you do the addition of 545 + 5 it will be equal to 550

Construction of syntax trees

Syntax Trees

- Syntax tree is a condensed form of parse tree useful for representing language constructs
- In a syntax tree, operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves in the parse tree

Constructing Syntax Trees for Expressions

- Each node in a syntax tree can be implemented as a record with several fields.
- Use the following functions to create the nodes of syntax trees for **expressions with binary operators.** Each function returns a pointer to a newly created node.

Construction of syntax trees

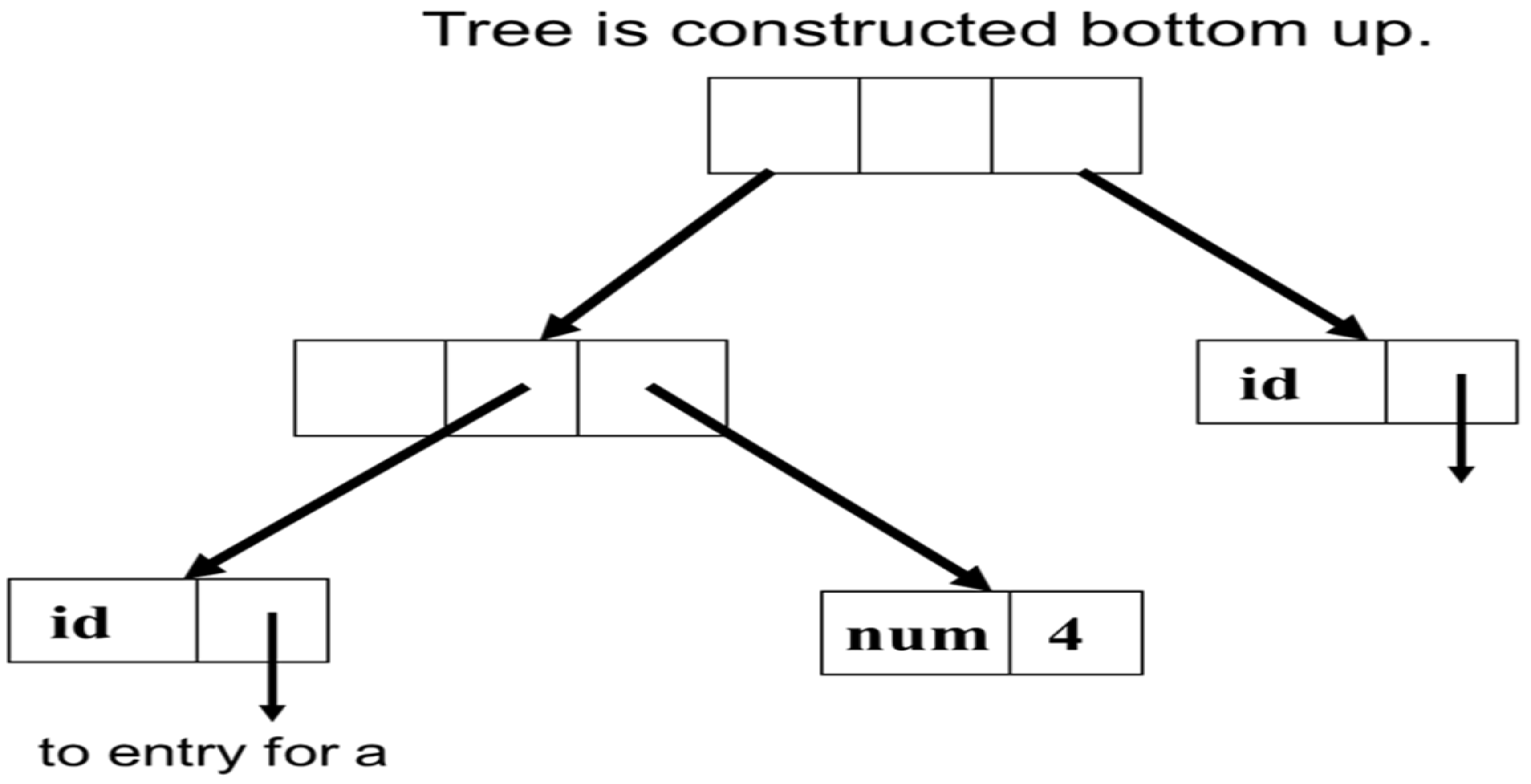
- *mknode(op, left, right)* creates an operator node with label ***op*** and two fields containing pointers to ***left*** and ***right.***
- ***mkleaf* (id, *entry)*** creates an identifier node with label **id** and a field containing *entry,* a pointer to the symbol-table entry for the identifier.
- ***mkleaf* (num, *val)*** creates a number node with label **num** and a field containing *val,* the value of the number.

- Example: Create the syntax tree for the expression: a - 4 + c

  (1) P1:= *mkleaf(id, entrya);* (2) *P2;= mkleaf(num,* 4);
  (3) *P3;=* mknode('-',p1,P2); (4) *P4:= mkleaf(id, entryc);*
  (5) *P5:= mknode('+',p3,P4);*

- P1, *P2, . . . , P 5* are pointers to nodes,
- *entrya* and *entryc* are pointers to the symbol-table entries for identifiers a and c, respectively.

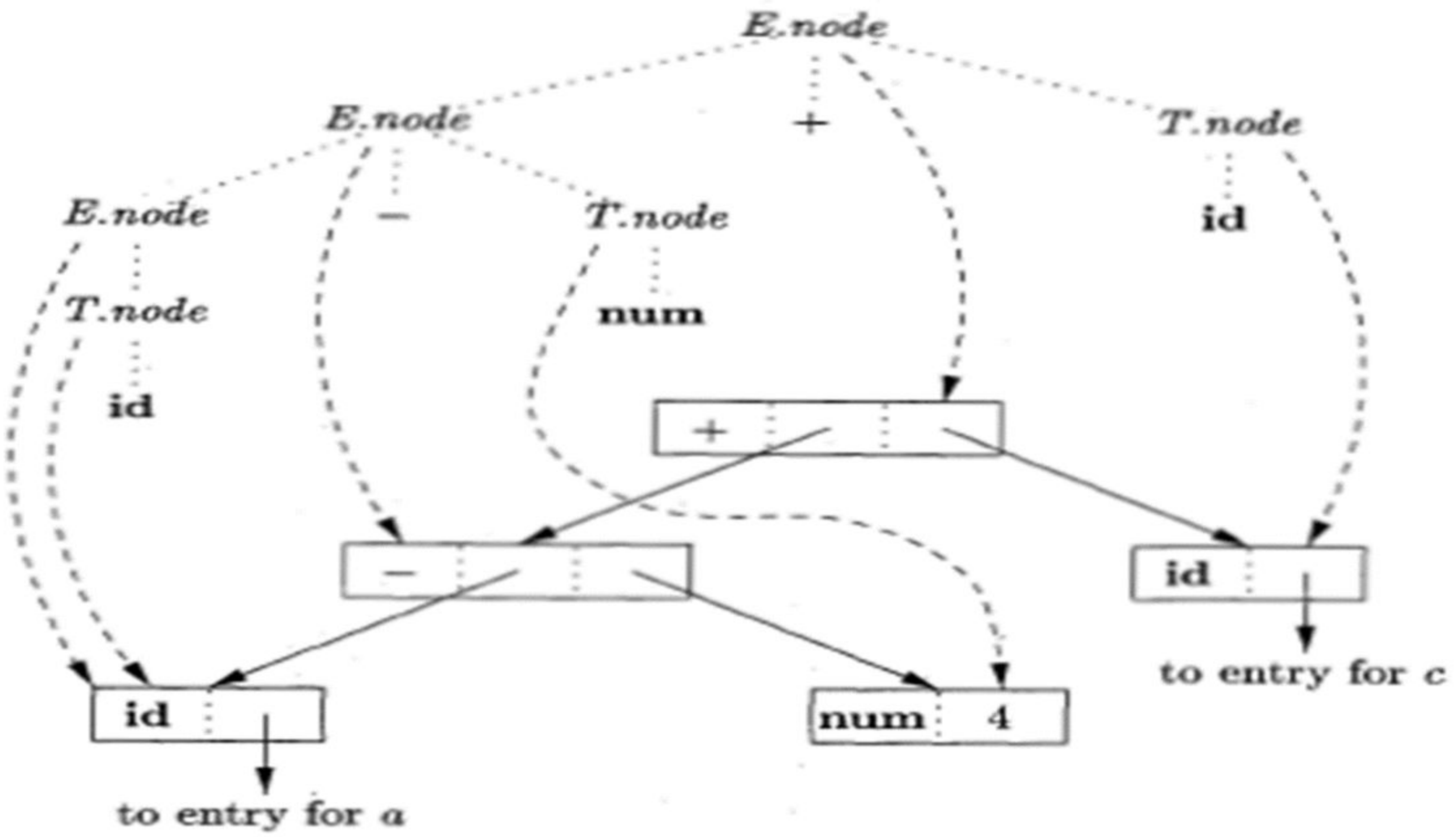
Syntax tree for a-4+c (Tree is constructed bottom up)



A Syntax-Directed Definition for Constructing Syntax Trees for an Expression

PRODUCTION	SEMANTIC RULES
$E \rightarrow E1+T$	*E.nptr* := *mknode{' +', E 1.nptr, T.nptr)*
$E \rightarrow E1-T$	*E.nptr* := *mknode{‘ - ', E 1.nptr, T.nptr)*
$E \rightarrow T$	*E.nptr* := *T.nptr*
$T \rightarrow (E)$	*T.nptr* := E.*nptr*
$T \rightarrow$ id	*T.nptr* := *mkleaf* (id, *id.entry)*
$T \rightarrow$ num	*T.nptr* := *mkleaf* (num, num. *val)*

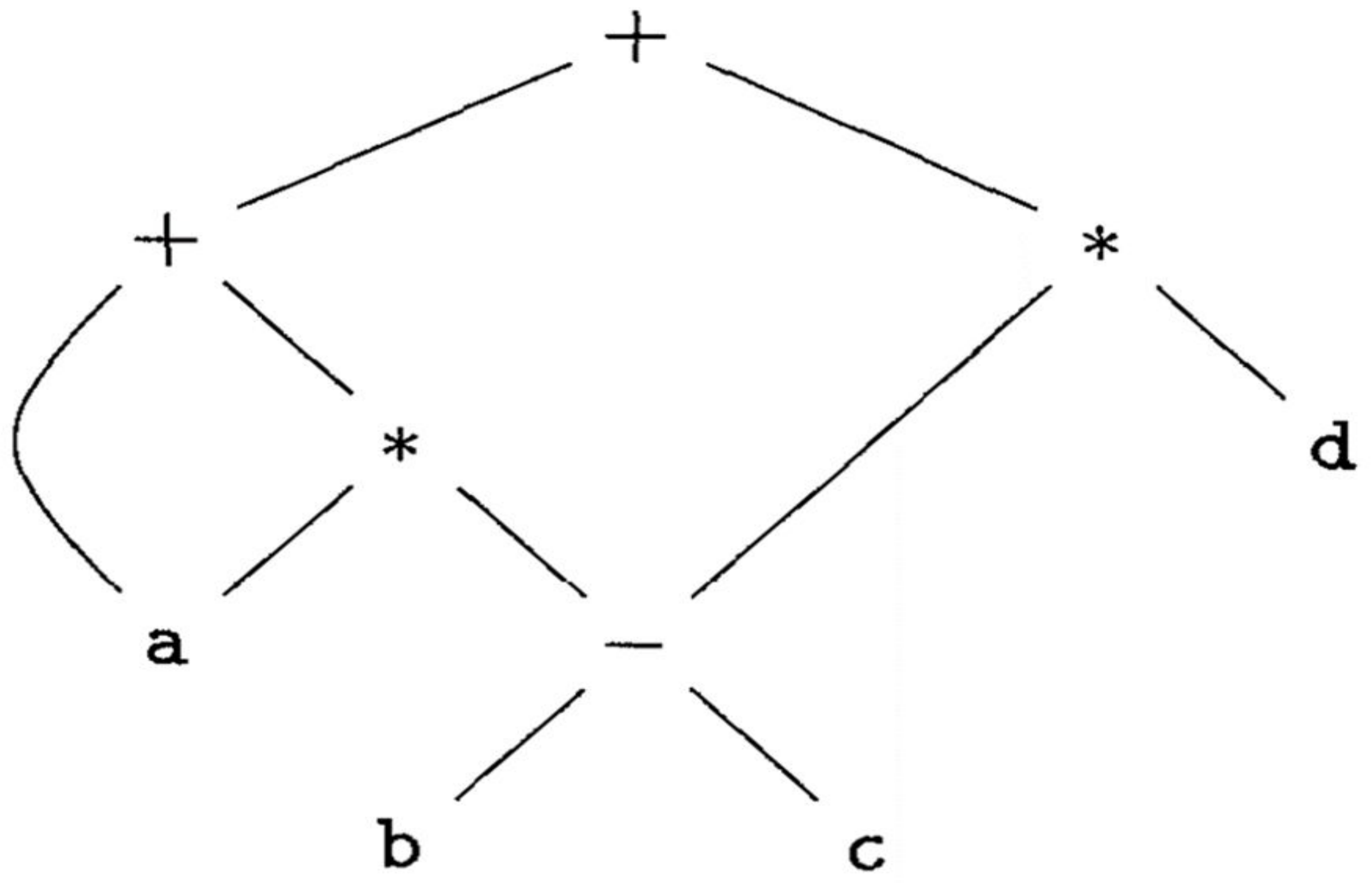
Annotated parse tree depicting the construction of a syntax tree for the expression a-4+c



Directed Acyclic Graphs for Expressions

- A directed acyclic graph (DAG) for an expression identifies the common sub expressions in the expression.
- Like a syntax tree, a dag has a node for every sub expression of the expression; an interior node represents an operator and its children represent its operands.
- The difference is that a node in a dag representing a common sub expression has more than one "parent;" in a syntax tree, the common sub expression would be represented as a duplicated sub-tree.

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



Directed Acyclic Graphs for Expressions

(1)P1:= *mkleaf(id, a);*
(2) P2:= *mkleaf(id, a);*
(3) *P3;= mkleaf(id,* b);
(4) *P4;= mkleaf(id, c);*
(5) *P5:=* mknode('-',p3,P4);
(6) *P6:= mknode(‘*',p2,P5);*
(7) *P7:= mknode(‘+',p1,P6);*
(8) *P8:= mkleaf(id, b);*
(9) *P9:= mkleaf(id, c);*
(10) *P10:= mknode(‘-',p8,P9);*
(11) *P11:= mkleaf(id, d);*
(12)P12:= mknode(‘',p10,P11);*
(13) *P13:= mknode('+',p7,P12);*

- ✔ Intermediate Code Generation
- ✔ Types of Intermediate Languages
- ✔ Three Address Code
  - – Types of Three-Address Statements
  - – Implementations of 3-address statements

Intermediate Code Generation

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code.

Benefits of using machine – independent intermediate form are

1. Retargeting is facilitated – a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end
2. Machine independent Code Optimization can be applied to the intermediate representation

Intermediate Code Generation

- *Intermediate codes* are machine independent codes, but they are close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- **Intermediate language** can be many different languages, and the designer of the compiler decides this intermediate language.
    - syntax trees can be used as an intermediate language.
    - postfix notation can be used as an intermediate language.
    - three-address code can be used as an intermediate language

Types of Intermediate Languages

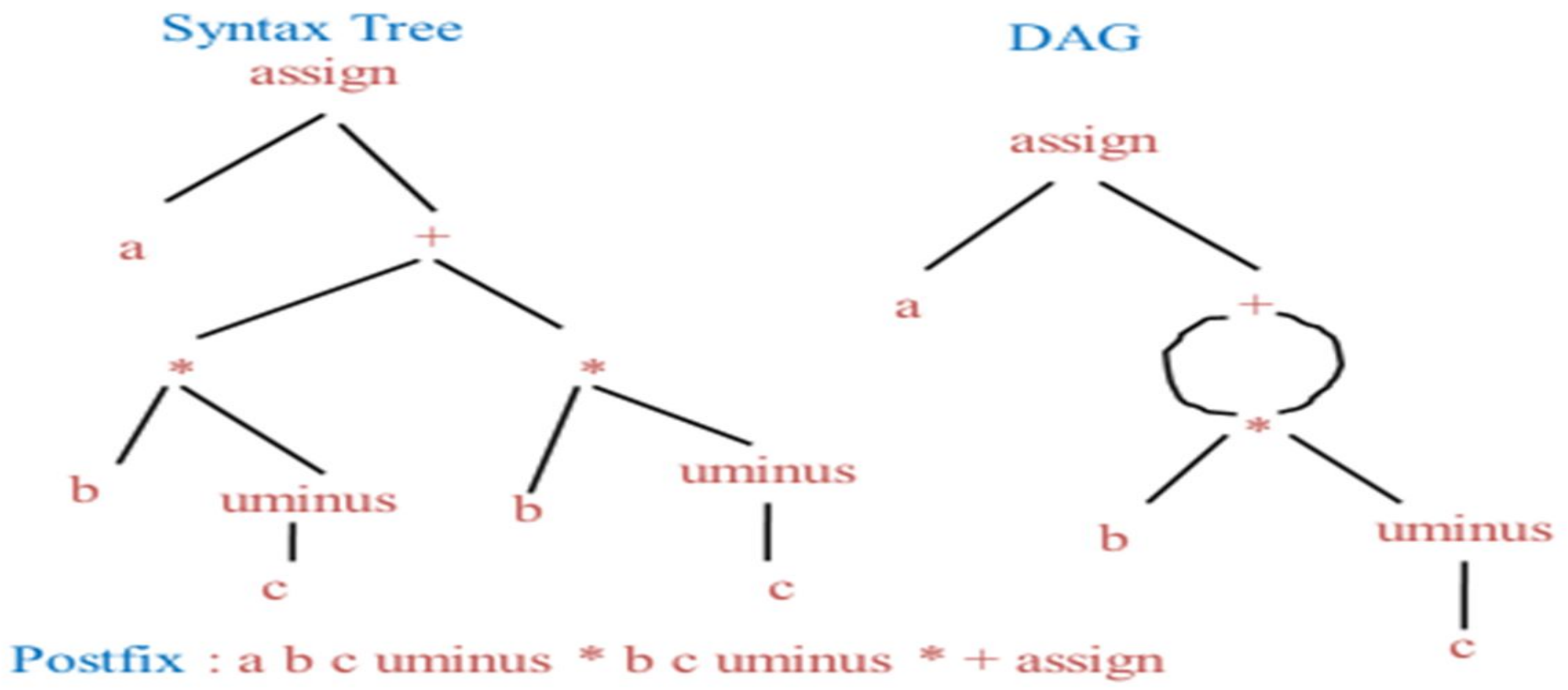
(i)(a) Syntax Tree: depicts the natural hierarchical structure of a source program (Syntax tree is a condensed form of parse tree useful for representing language constructs. In a syntax tree, operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves in the parse tree)

(b) DAG: gives the same information but in a more compact way because common sub-expressions are identified.

(ii) Postfix notation : is a linearized representation of a syntax tree;

it is a list of the nodes of the tree in which a node appear immediately after its children.

Consider the assignment a:=b*-c+b*-c



Syntax Dir. Definition to produce Syntax trees for Assignment Statements

Production	Semantic Rule
$S \rightarrow$ id := E	{ S.*nptr* = *mknode* ('assign', *mkleaf*(id, id.*entry*), E.*nptr*) }
$E \rightarrow E_1 + E_2$	{E.*nptr* = *mknode*('+', E_1.*nptr*,E_2.*nptr*) }
$E \rightarrow E_1 * E_2$	{E.*nptr* = *mknode*('*', E_1.*nptr*,E_2.*nptr*) }
$E \rightarrow - E_1$	{E.*nptr* = *mknode*('uminus',E_1.*nptr*) }
$E \rightarrow (E_1)$	{E.*nptr* = E_1.*nptr* }
$E \rightarrow$ id	{E.*nptr* = *mkleaf*(id, id.*entry*) }

Three Address Code

- Three Address Code is a sequence of Statements of general form x:=y op z
  where x,y,z are names, constants or compiler generated temporaries; op is any operator
- No built-up arithmetic expressions are allowed.
- Given the syntax-tree or the dag of the graphical representation we can easily derive a three address code for assignments.
- Three-address code is a linearization of the syntax tree.
- Three-address code is useful: related to machine-language/ simple/ optimizable.

Example of 3-address code

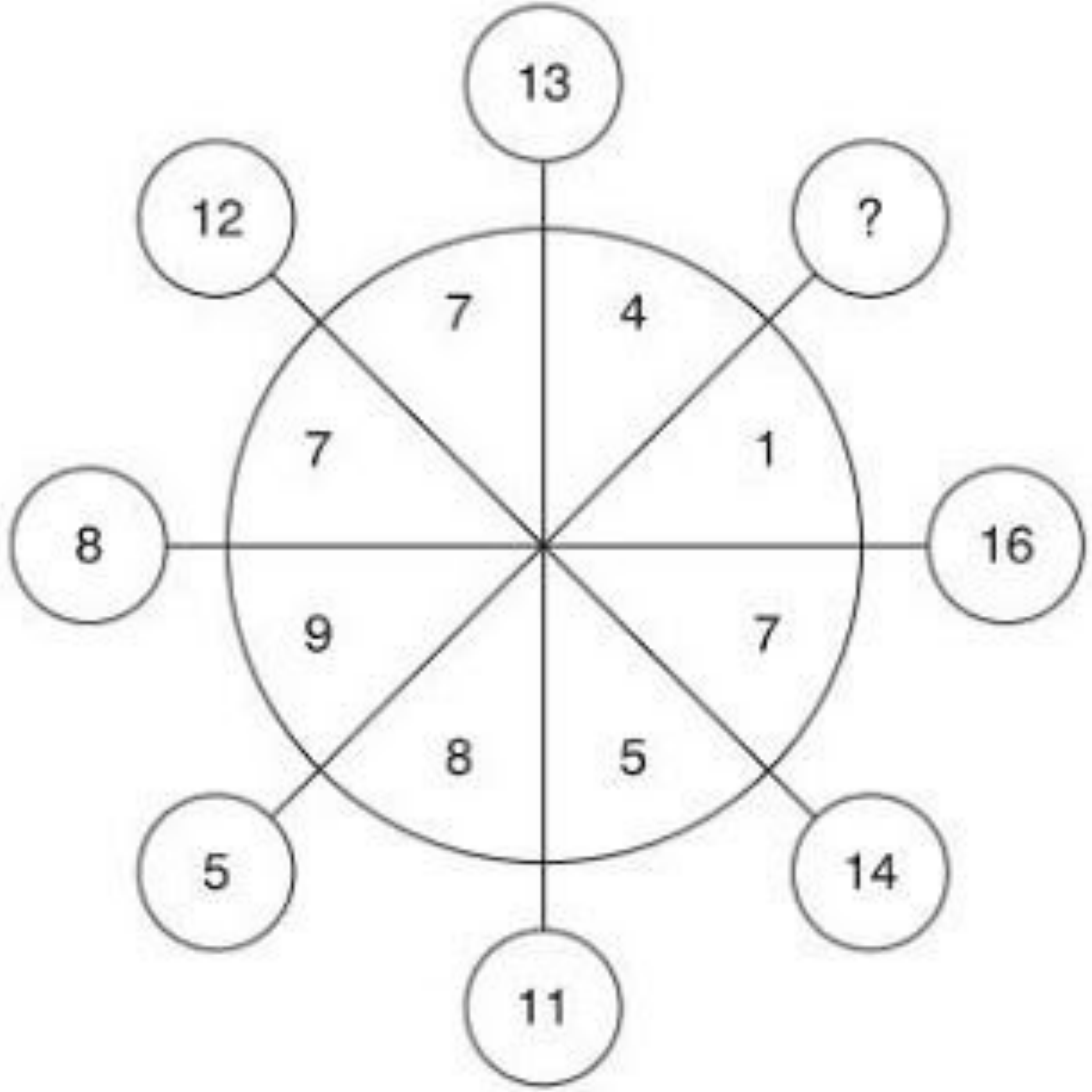
a:= b* -c + b * -c

```
t1:=- c
t2:=b * t1
t3:=- c
t4:=b * t3
t5:=t2 + t4
a:=t5
```

Code for syntax tree

```
t1:=- c
t2:=b * t1
t5:=t2 + t2
a:=t5
```

Code for DAG


13
?
16
14
11
5
8
12
7
4
1
7
5
8
9
7

- It is the sum of the two digits(9 + 8) in opposite quadrant

List of the common three-address instruction form

(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.

store result of binary operation on y and z to x

(2). Assignment instructions of the form **x := op y,** where op is a unary operation. Essential unary operations include **unary minus, logical negation, shift operators, and conversion operators** that, for example, convert an integer to a floating-point number.

store result of unary operation on y to x

(3). Copy instructions of the form **x := y**, where the value of y is assigned to x .

store y to x

(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 if False x goto L**. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

(6). Binary 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. If not, the three-address instruction following if x relop y goto L is executed next, in sequence.

List of the common three-address instruction form

(7). **param x**

store x as a parameter

call p,n,x

call procedure p with n parameters, store result in x

return x

return from procedure, use x as the result

List of the common three-address instruction form

(8) Indexed copy instructions of the form x: = y [i] and x [i] := y.

The instruction x: = y [i] *sets x to the value in the location i memory units beyond* location y .

The instruction x [i] := y sets the contents of the location ***i*** units beyond x to the value of y.

(9). Address and pointer assignments of the form x := & y, x := * y, and * x:= y.

The instruction x := & y   store address of y to x

The instruction x = * y store contents pointed to by y to x

The instruction * x = y store y to location pointed to by x

Syntax-Directed Translation into 3-address code

- Synthesized attribute S.code - represents 3ACode for assignment

- Use attributes
  - E.*place:* the name that will hold the value of E
  - E.*code:* the sequence of three address statements evaluating E.

- Use function ***newtemp*** that returns a new temporary variable that we can use.

- Use function ***gen*** to generate a single three address statement given the necessary information (variable names and operations).

Syntax Dir. Definition to produce 3-address code for Assignment Statements

Production	**Semantic Rule**
S → id := E	{ S.*code* = E.*code*||*gen*(id.*place* ':=' E.*place* ';') }
E → E_1 + E_2	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || E_2.*code* || || *gen*(E.*place*':='E_1.*place*'+'E_2.*place*) }
E → E_1 * E_2	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || E_2.*code* || || *gen*(E.*place*':='E_1.*place*'*'E_2.*place*) }
E → - E_1	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || || *gen*(E.*place* ':=' 'uminus' E_1.*place*) }
E → (E_1)	{E.*place*= E_1.*place* ; E.*code* := E_1.*code*}
E → id	{E.*place* = id.*place*; E.*code* = '' }

Ex: a:= b* -c + b * -c

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

Implementations of 3-address statements

In a compiler, three address statements can be implemented as records with fields for the operator and the operands. Three such representations are called **"quadruples, triples, and indirect triples."**

- **Quadruple** has four fields - op, arg1, arg2,and result.
- op field contains an internal code for the operator.
- ✔ 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 = - y or x = y do not use arg2.
- **The contents of fields arg1, arg2,and result are normally pointers to the symbol table entries for the names represented by these fields.**

  If , so temporary names must be entered into the symbol table as they are created.

Implementations of 3-address statements….

Quadruples

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

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Temporary names must be entered into the symbol table as they are created.

Implementations of 3-address statements

- A **triple** has only three fields, which we call op, arg1, and arg2.
- Note that, the result field in 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 tl in quadruple, a triple representation would refer to position (0). Parenthesized numbers represent pointers into the triple structure itself

Implementations of 3-address statements….

- A **triple** has only three fields, which we call op, arg1, and arg2.

  Triples

  t1:=- c

  t2:=b * t1

  t3:=- c

  t4:=b * t3

  t5:=t2 + t4

  a:=t5

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

Temporary names are not entered into the symbol table. Refer to a temporary value by the position of the statement that computes it.

Implementations of 3-address statements…..

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves. For example, let us use an array instruction to list pointers to triples in the desired order.

	op		*op*	*arg1*	*arg2*
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	assign	a	(18)

Type Expression

- The type of a language construct is denoted by a *type expression*.
- A ***type expression*** can be:

A basic type

- Among the basic types are *boolean, char, integer,* and *real.*
- A special basic type, *type_error,* will signal an error during type checking.
- A basic type *void* denoting "the absence of a value" allows statements to checked.

A type name

- a name can be used to denote a type expression.

A type constructor applies to other type expressions.

- **arrays**: If T is a type expression, then *array(I,T)* is a type expression where I denotes index range. Ex: array(0..99,int)
- **products**: If T_1 and T_2 are type expressions, then their cartesian product $T_1 x\ T_2$ is a type expression. Ex: int x int
- **pointers**: If T is a type expression, then *pointer(T)* is a type expression. Ex: pointer(int)
- **functions**: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression $D \rightarrow R$ where D are R type expressions. Ex: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.

Type Equivalence

- How do we know that two type expressions are equal?
- As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions
- 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

- If type names are treated as standing for themselves, then the first two conditions in the above definition lead to name equivalence of type expressions

Declarations

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

- ✔ Study types and declarations using a simplified grammar that declares just one name at a time
- ✔ Nonterminal D generates a sequence of declarations.
- ✔ Nonterminal T generates basic, array, or record types.
- ✔ Nonterminal B generates one of the basic types int and 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 (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

Storage Layout for Local Names

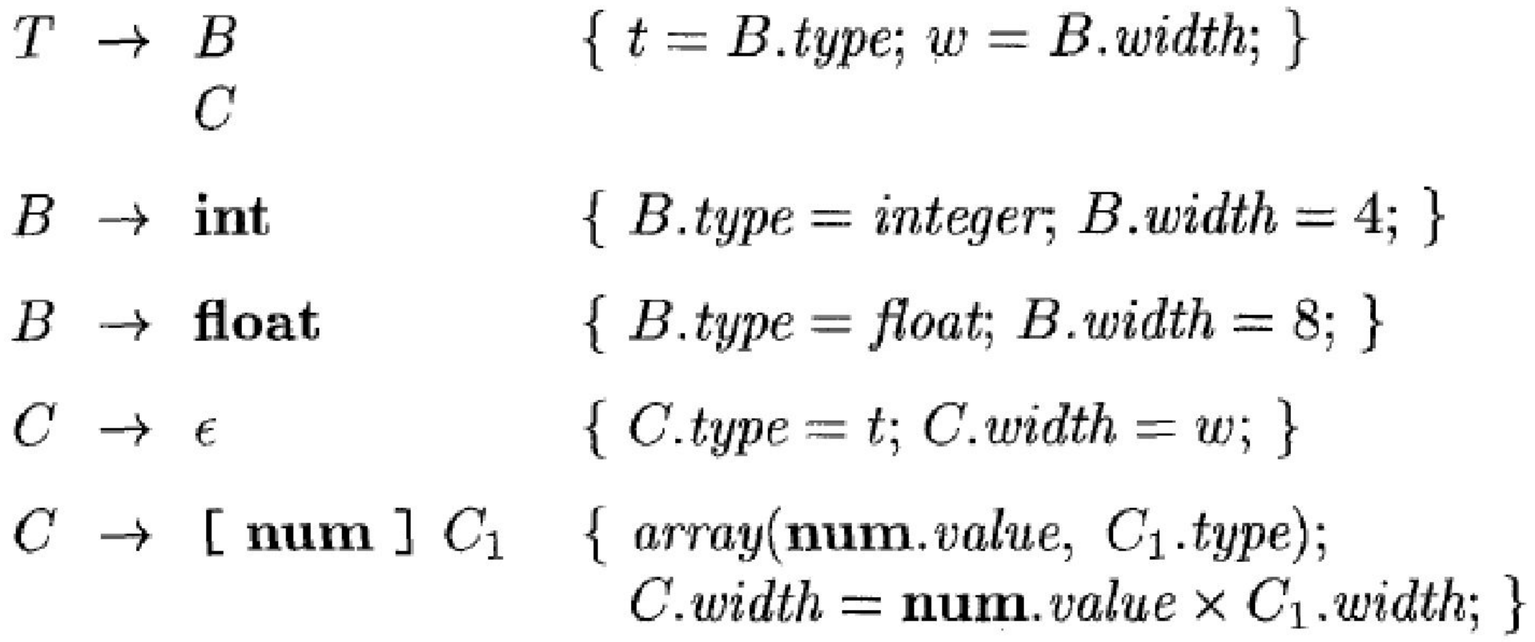
- ✔ From the type of a name, we can determine the amount of storage that will be needed for the name at run time.
- ✔ At compile time, we can use these amounts to assign each name a relative address.
- ✔ The type and relative address are saved in the symbol-table entry for the name.

Storage Layout for Local Names

- ✔ The width of a type is the number of storage units needed for objects of that type.
- ✔ 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

Storage Layout for Local Names

- The translation scheme (SDT) given below computes types and their widths for basic and array types



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

Storage Layout for Local Names

- ✔ SDT uses synthesized attributes type and width for each nonterminal and two variables t and w to pass type and width information from a B node in a parse tree to the node for the production C □ Ɛ. In a syntax-directed definition, t and w would be inherited attributes for C.
- ✔ Body of the T-production consists of nonterminal B, an action, and nonterminal C, which appears on the next line.
- ✔ The action between B and C sets t to B.type and w to B.width.
- ✔ If B □ int then B. type is set to integer and B. width is set to 4, the width of an integer.
- ✔ If B □ float then B. type is float and B. width is 8, the width of a float.

Storage Layout for Local Names

- ✔ The productions for C determine whether T generates a basic type or an array type.
- ✔ If C Ɛ, then t becomes C.type and w becomes C.width. Otherwise, C specifies an array component.
- ✔ The action for C [num] Cl forms C.type by applying the type constructor array to the operands num.value and Cl .type
- ✔ The width of an array is obtained by multiplying the width of an element by the number of elements in the array. If addresses of consecutive integers differ by 4, then address calculations for an array of integers will include multiplications by 4

Sequences of Declarations

- Languages such as C and Java allow all the declarations in a single procedure to be processed as a group.
- The declarations may be distributed within a Java procedure, but they can still be processed when the procedure is analyzed.
- Therefore, we can use a variable, say offset, to keep track of the next available relative address.

Sequences of Declarations

- ✔ Below translation scheme deals with a sequence of declarations of the form T id, where T generates a type as in previous SDT. 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

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

Sequences of Declarations

- ✔ The semantic action within the production D T id ; D1 creates a symbol table entry by executing top.put(id.lexeme, T. type, offset).
- ✔ Here top denotes the current symbol table.
- ✔ The method top.put creates a symbol-table entry for id.lexerne, with type T.type and relative address offset in its data area.
- ✔ The initialization of offset is more evident if the first production appears on one line as:

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

Sequences of Declarations

- ✔ Marker Nonterminals generating Ɛ, can be used to rewrite productions
- ✔ Using a marker nonterminal M, previous equation can be restated as:

$$\begin{array}{lcll} P & \rightarrow & M\ D & \\ M & \rightarrow & \epsilon & \{\ \mathit{offset} = 0;\ \} \end{array}$$

- ✔ *Operations Within Expressions*
- ✔ *Names in the Symbol Table*
- ✔ *Addressing Array Elements*
- ✔ *Translation of Array References*

Operations Within Expressions

Syntax Dir. Definition to produce 3-address code for Assignment Statements:

Production	**Semantic Rule**
$S \rightarrow$ id := E	{ S.*code* = E.*code*||*gen*(id.*place* ':=' E.*place* ';') }
$E \rightarrow E_1 + E_2$	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || E_2.*code* || || *gen*(E.*place*':='E_1.*place*'+'E_2.*place*) }
$E \rightarrow E_1 * E_2$	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || E_2.*code* || || *gen*(E.*place*':='E_1.*place*'*'E_2.*place*) }
$E \rightarrow - E_1$	{E.*place*= *newtemp* ; E.*code* = E_1.*code* || || *gen*(E.*place* ':=' 'uminus' E_1.*place*) }
$E \rightarrow (E_1)$	{E.*place*= E_1.*place* ; E.*code* := E_1.*code*}
$E \rightarrow$ id	{E.*place* = id.*place*; E.*code* = '' }

Syntax Dir. Definition to produce 3-address code for Assignment Statements

- Synthesized attribute S.Code - represents 3ACode for assignment

- Use attributes
  - E.*place:* the name that will hold the value of E
  - E.*code:* the sequence of three address statements evaluating E.

- Use function ***newtemp*** that returns a new temporary variable that we can use.

- Use function ***gen*** to generate a single three address statement given the necessary information (variable names and operations).

Example:

The syntax-directed definition translates the assignment statement a = b + - c ; into the three-address code sequence

tl = minus c

t2 = b + tl

a = t2

Names in the Symbol Table

- ✔ The translation scheme to Produce Three-Address Code for assignments, shows how symbol-table entries can be found.
- ✔ Code can be sent to an output file instead of being assigned to the *code attribute.*
- ✔ Lexeme for the name represented by id is given by attribute *id.name.*
- ✔ Operation *lookup (id.name)* checks if there is an entry for this occurrence of the name in the symbol table. If so, a pointer to the entry is returned; otherwise, *lookup* returns *nil* to indicate that no entry was found.
- ✔ Semantic actions use procedure *emit* to emit three-address statements to an Output file, rather than building up *code* attributes for non-terminals.

Translation Scheme to Produce Three-Address Code

Production **Semantic Rule**

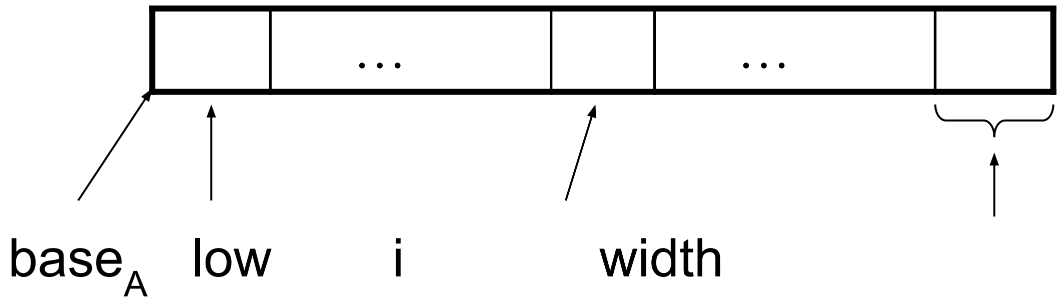
```
S → id := E { p:= lookup(id.name);
              if (p is not nil) then  emit(p ':=' E.place)
              else error}
E → E₁ + E₂{ E.place := newtemp;
              emit(E.place ':=' E₁.place '+' E₂.place) }
E → E₁ * E₂ { E.place: = newtemp;
              emit(E.place ':=' E₁.place '*' E₂.place)  }
E → - E₁     { E.place := newtemp;
              emit(E.place ':=' 'uminus' E₁.place)  }
E → ( E₁ )   { E.place := E₁.place; }
E → id       { p= lookup(id.name);
              if (p is not nil) then E.place = id.place
              else error}
```

Addressing array Elements

Arrays

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations.

A one-dimensional array **A**:



$base_A$ is the address of the first location of the array A,

width is the width of each array element.

low is the index of the first array element

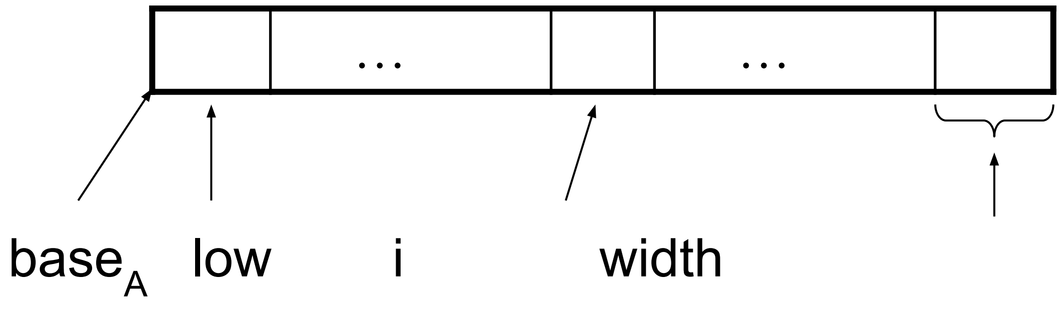
location of A[i] □ $base_A$+(i-low)*width …………………(1)

Addressing array Elements

Arrays

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations.

A one-dimensional array **A**:



$base_A$ is the address of the first location of the array A,

width is the width of each array element.

low is the index of the first array element

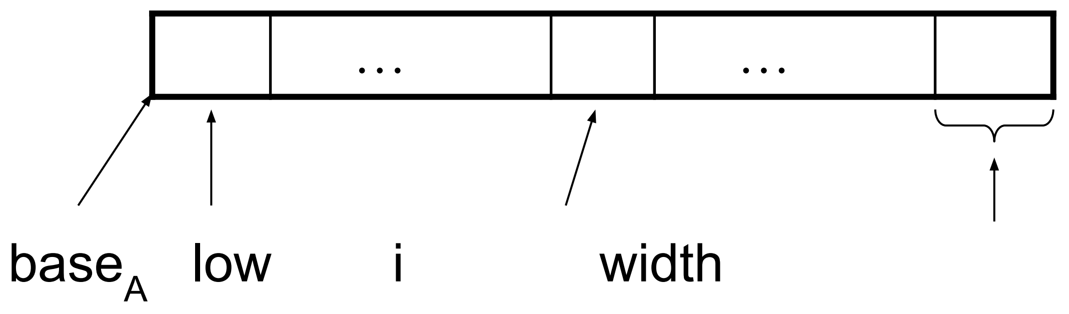
location of A[i] □ $base_A$+(i-low)*width …………………(1)

Addressing array Elements

Arrays

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations.

A one-dimensional array **A**:



$base_A$ is the address of the first location of the array A,

width is the width of each array element.

low is the index of the first array element

location of A[i] □ $base_A$+(i-low)*width …………………(1)

Translation needed

- If we use the following grammar to calculate addresses of array elements, we need inherited attributes.

  L → **id** | **id** [Elist]
  Elist → Elist , E | E

- Instead of this grammar, we will use the following grammar to calculate addresses of array elements so that we do not need inherited attributes (we will use only synthesized attributes).

  L → **id** | Elist]
  Elist → Elist , E | **id** [E

Translation needed

- These Productions allow a pointer to the symbol-table entry for the array name to be passed as a synthesized attribute array of Elist.
- Use Elist.ndim records the number of dimensions in the Elist.
- The function limit (array , j) returns nj , number of elements along the jth dimension of the array whose symbol – table entry pointed to by array.
- Elist.Place denotes the temporary holding a value computed from index expressions in Elist.

- An l-value L will have two attribute L. Place and L. offset.

  In the case that L is a simple name, L.place will be a pointer to the symbol-table entry for that name, and L.offset will be null, indicating that the l-value is a simple name rather than an array reference.

- The nonterminal E has the same translation E.place, with the same meaning ie. the name that will hold the value of E

Translation Scheme for Addressing array Elements

1. $S \rightarrow L := E$
2. $E \rightarrow E_1 + E_2$
3. $E \rightarrow (E_1)$
4. $E \rightarrow L$
5. $L \rightarrow Elist\ ]$
6. $L \rightarrow id$
7. $Elist \rightarrow Elist_1 , E$
8. $Elist \rightarrow id\ [\ E$

Translation Scheme for Addressing array Elements

Generate a normal assignment if *L* is a simple name, and an indexed assignment into the location denoted by *L* otherwise:

```
1. S → L := E{ if (L.offset is null)  then / *L is a simple id * /
                    emit(L.place ':=' E.place);
               else
                    emit( L.place '[' L.offset ']' ':=' E.place) }
```

The code for arithmetic expressions is exactly the same as previous one

```
2.E → E1 + E2      { E.place: = newtemp;
            emit(E.place ':=' E1.place '+' E2.place) }

3.E → ( E1 )   { E.place: = E1.place }
```

When an array reference *L* is reduced to *E,* we want the r-value of *L.* Therefore we use indexing to obtain the contents of the location *L.place [L.offset]:*

```
4.E → L  { if (L.offset is null) then / *L is a simple id * /
                   E.place := L.place)
               else { E.place := newtemp;
                     emit(E.place ':=' L.place '[' L.offset ']') } }
```

Translation Scheme for Addressing array Elements

L.offset is a new temporary representing the first term of (5); function *width(Elist.array)* returns *w* in (5).

L.place represents the second term of (5), returned by the function *c(Elist.array).*

5.L → Elist]

```
{ L.place := newtemp;
    L.offset := newtemp;
   emit(L.place ':=' c(Elist.array) ));
   emit(L.offset ':=' Elist.place '*' width(Elist.array) )) }
```

A null offset indicates a simple name.

6.L → id { L.place := **id**.place;
L.offset := null }

.

Translation Scheme for Addressing array Elements

In the following action, *Elist1.place* corresponds to *em* -1 in (8) and *Elist.place* to *em*, Note that if *Elist1* has *m* -1 components, then *Elist* on the left side of the production has *m* components.

7.Elist → Elist$_1$, E

```
{ Elist.array = Elist1.array ;
  Elist.place = newtemp;
  Elist.ndim = Elist1.ndim + 1;
  emit(Elist.place ':=' Elist1.place '*' limit(Elist1.array, Elist.ndim));
  emit(Elist.place ':=' Elist.place '+' E.place )}
```

E.place holds both the value of the expression *E* and the value of *(7) for m=1*

8.Elist → id [E

```
{Elist.array := id.place ;
 Elist.place := E.place;
 Elist.ndim := 1; }
```

Example

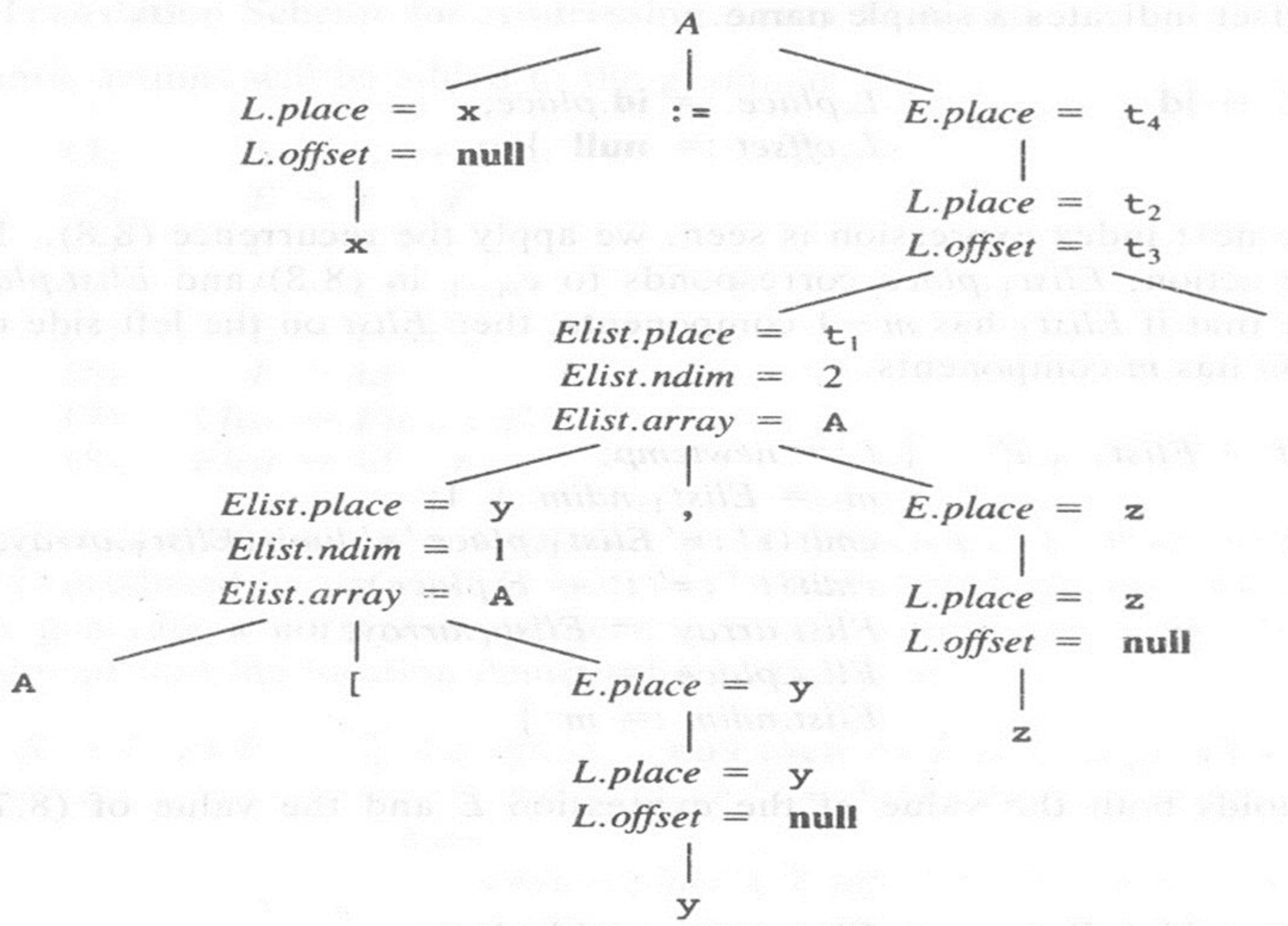
Let A be a 10 x 20 array with *low* 1 = *low2* = l. Take *w* to be 4. Construct Annotated parse tree and three address code for the assignment X:=A[Y,Z]

The assignment is translated into the following sequence of three-address statements:

- t0 : = y * 20
- tl : = t0 + z
- t2 : = c /* constant c = *base* A - 84 */
- t3 : = 4 * tl
- t4 : = t2 [t3]
- x := t4

An annotated parse tree for the assignment x : = A[y, z]

.



Type Checking

- A compiler has to do semantic checks in addition to syntactic checks.

- (Semantic Checks) Static and Dynamic Checking of Types.
  - Checking done by a compiler is said to be **static**,
  - Checking done when the target program runs is termed **dynamic**

- *Type checking* is one of these static checking operations.

  – we may not do all type checking at compile-time.

  – Some systems also use dynamic type checking too.

Type Systems

- A ***type system*** is a collection of rules for assigning type expressions to the various parts of a program
- A **type checker** implements a type system
- Different type systems may be used by different compilers or processors of the same language
- A ***sound*** **type system** eliminates the need for dynamic checking for type errors
- A language is ***strongly typed*** if its compiler can guarantee that the programs it accepts will execute without type errors.
  - In practice, some of type checking operations are done at run-time (so, most of the programming languages are not strongly-typed)

Rules for Type Checking

✔ Type checking can take on two forms: synthesis and inference.

✔ Type synthesis builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used.

✔ The type of El + E2 is defined in terms of the types of El and E2.

✔ A typical rule for type synthesis has the form

if f has type s □ t and x has type s,
then expression f (x) has type t ……………..(1)

✔ Here, f and x denote expressions, and s □ t denotes a function from s to t.

✔ This rule for functions with one argument carries over to functions with several arguments. The rule (1) can be adapted for El + E2 by viewing it as a function application add(E1 , E2)

Rules for Type Checking

- ✔ Type inference determines the type of a language construct from the way it is used.
- ✔ Let null be a function that tests whether a list is empty. Then, from the usage null(x), we can tell that x must be a list.
- ✔ The type of the elements of x is not known; all we know is that x must be a list of elements of some type that is presently unknown.
- ✔ Variables representing type expressions allow us to talk about unknown types. We shall use Greek letters α, β, …. for type variables in type expressions.
- ✔ A typical rule for type inference has the form

  if f (x) is an expression,

  then for some α and , β, f has type α □ β and x has type α …………. (2)

- ✔ Type inference is needed for language, which check types, but do not require names to be declared

Specification of A Simple Type Checker

- The type checker is a translation scheme that synthesizes the type of each expression from the types of its sub expressions.
- The type checker can handle arrays, pointers, statements, and functions

A Simple Type Checking System

$P \rightarrow D\ ;E$	
$D \rightarrow D\ ;D$	
$D \rightarrow$ **id** :T	{ addtype(id.entry,T.type) }
$T \rightarrow$ char	{ T.type=char }
$T \rightarrow$ int	{ T.type=int }
$T \rightarrow$ real	{ T.type=real }
$T \rightarrow \uparrow T_1$	{ T.type=pointer(T_1.type) }
$T \rightarrow$ array[**num**] of T_1	{ T.type=array(1.. num.val,T_1.type) }

A Simple Type Checking System

- The language itself has three basic types, ***char, integer and real;*** a fourth basic type ***type_error*** is used to signal errors .

- **array [256] of char** leads to the **type expression *array*(1…..256, char)** consisting of the constructor *array* applied to the subrange 1…256 and the type *char.*

- ↑ **integer** leads to the type expression ***pointer(integer)****,* consisting of the constructor *pointer* applied to the type *integer*
- In the translation scheme, the action associated with the production D → id : T saves a type in a symbol-table entry for an identifier.

  The action addtype (id.entry, T.type) is applied to synthesized attribute entry pointing to the symbol-table entry for id and a type expression represented by synthesized attribute type of nonterminal T.
- Since *D* appears before *E* on the right side of *P* → *D* ; *E,* we can be sure that the types of all declared identifiers will be saved before the expression generated by *E* is checked.

Type Checking of Expressions

- The synthesized attribute *type* for *E* gives the type expression assigned by the type system to the expression generated by *E.*
- In the semantic rules constants represented by the tokens **literal** and **num** have type *char* and *integer,* respectively:

  E → **literal** { E.type:= char }

  E → **num** { E.type:= integer }
- Use a function ***lookup(e)*** to fetch the type saved in the symbol-table entry pointed to *by e.* When an identifier appears in an expression, its declared type is fetched and assigned to the attribute *type:*

  E → **id** { E.type=lookup(id.entry) }
- The expression formed by applying the mod operator to two subexpressions of type *integer* has type *integer;* otherwise, its type is *type_error:*

  E → *E1* ***mod*** *E2* { *E.type* := if *E1type* = *integer* and *E2.type* = *integer*
  then *integer* else *type-error* }

Type Checking of Expressions

- In an array reference *E1* [*E* 2], the index expression *E* 2 must have type integer, in which case the result is the element type *t* obtained from the type *array (s, t)* of *E1;* otherwise, its type is type_error:

  $E \rightarrow E1 \, [\, E2 \,]$ {if (E_2.type=int and E_1.type=array(s,t)) then
  E.type=t else E.type=type-error }

- Within expressions, the postfix operator ↑ yields' the object pointed to by its operand. The type of *E* ↑ is the type *t* of the object pointed to by the pointer *E:*

  $E \rightarrow E_1 \uparrow$ { if (E_1.type=pointer(t)) then E.type=t
  else E.type=type-error }

Type Checking of Statements

```
S → id := E       { if (id.type=E.type then S.type=void
                    else S.type=type-error }
S → if E then S1     { if (E.type=boolean then S.type=S1.type
                    else S.type=type-error }
S → while E do S1 { if (E.type=boolean then S.type=S1.type
                    else S.type=type-error }
```

Rules for checking statements.

- The first rule checks that the left and right sides of an assignment statement have the same type.
- The second and third rules specify that expressions in conditional and while statements must have type *Boolean*

Type Checking of Functions

The rule for checking the type of a function application is

$E \rightarrow E_1 (E_2)$ { if (E_2.type=s and E_1.type=s$\rightarrow$t) then E.type=t
else E.type=type-error }

This rule says that in an expression formed by applying *E1* to *E2,* the type of *E*1 must be a function *s* $\rightarrow$ *t* from the type *s* of *E* 2 to some range type *t;* the type of *E1(E2*) is *t.*

Type Conversions

- Consider expressions like x + i where x is of type real and i is of type integer. Since the representation of integers and reals is different within a computer, and different machine instructions are used for operations on integers and reals, the compiler may have to first convert one of the operands of + to ensure that both operands are of the same type when the addition takes place
- The language definition specifies what conversions are necessary. When an integer is assigned to a real, or vice versa, the conversion is to the type of the left side of the assignment.
- In expressions, the usual transformation is to convert the integer into a real number and then perform a real operation on the resulting pair of real operands
- x + i, might be **x i inttoreal real +.** Here, the inttoreal operator converts i from integer to real and then **real+** performs real addition on its operands

Back patching

- A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump, ie in one single pass we may not know the labels that control must go to at the time the jump statements are generated.
- We addressed this problem by generating series of branch statements with the targets of the jump is temporarily left unspecified.
- Each such statement will be put on a list of goto statement whose labels are to be filled in when the proper label can be determined.
- We call this subsequent filling in of labels **Backpatching**

Back Patching

- Back patching can be used to generate code for Boolean expressions and flow of-control statements in one pass.
- For specificity, we generate instructions into an instruction array, and labels will be indices into this array.
- To manipulate lists of jumps, we use three functions:
  - **makelist(i)**
    - Creates a new list containing only i, and index into the array of quadruples
    - Returns a pointer to the new list
  - **merge(p1, p2)**
    - Concatenates the lists pointed to by p1 and p2
    - Returns a pointer to the new list
  - **Backpatch(p, i)**
  - inserts i as target label for each statements on the list pointed to by p

UNIT IV RUN-TIME ENVIRONMENT AND CODE GENERATION

Issues in the Design of a Code Generator

- Input to the Code Generator
- Target Programs
- Memory management
- Instruction Selection
- Register allocation and
- Evaluation order

1.Input to the Code Generator

- Input to the code generator is the intermediate representation of the source program produced by the front end.
- Along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.
- Intermediate representation can be :

  a. Linear representation such as postfix notation

  b. Three address representation such as quadruples

  c. Virtual machine representation such as stack machine code

  d. Graphical representations such as syntax trees and Dags.
- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking.
- Input to code generation is assumed to be error-free.

2.Target Programs

- Output of the code generator is the target program.
- Target program may be
  - Absolute machine language
- It can be placed in a fixed location in memory and immediately executed.
- Small Programs can be compiled and executed quickly.
  - Re-locatable machine language
- It *allows subprograms to be compiled separately.*
- set of re-locatable object modules can be linked together and loaded for execution by a linker
  - Assembly language
- Code generation is made easier.

3.Memory Management

- Mapping names in the source program to addresses of the data objects in run time memory is done cooperatively by the front end and the code generator

4.Instruction Selection

- Nature of the instruction set of the target machine determines the difficulty of the instruction selection.
- Uniformity and completeness of the instruction set are important
- Instruction speeds is also important
- If we do not care about the efficiency of the target program, instruction selection is straightforward.
  - Say, x = y + z

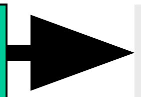
```
Mov y, R0
Add z, R0
Mov R0, x
```

Statement by statement code generation often produces poor code

4.Instruction Selection ….

```
a = b + c
d = a + e
```

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```



If a is subsequently used

4.Instruction Selection

- The quality of the generated code is determined by its speed and size.
- Cost difference between the different implementation may be significant.
  - Say a = a + 1

    ```
    Mov a, R0
    Add #1, R0
    Mov R0, a
    ```

  - If the target machine has increment instruction (INC), we can write

    ```
    inc a
    ```

5.Register Allocation

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- Efficient utilization of register is particularly important in code generation.
- The use of register is subdivided into two sub problems
  - During register allocation, we select the set of variables that will reside in register at a point in the program.
  - During a subsequent register allocation phase, we pick the specific register that a variable will reside in.

6.Choice of evaluation Order

- The order in which computation are performed can affect the efficiency of the target code
- Some computation orders require fewer registers to hold intermediate results than others.
- Picking a best order is another difficult.

TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- Target computer is a byte-addressable machine with 4 bytes to a word.
- It has n general-purpose registers, R0, R1, . . . , Rn-1.
- It has two-address instructions of the form:

  op source, destination

  where, op is an op-code, and source and destination are data fields.
- It has the following op-codes :

  MOV (move source to destination)

  ADD (add source to destination)

  SUB (subtract source from destination)
- Source and destination of an instruction are specified by combining registers and memory locations with address modes.

  For example : MOV R0, M stores contents of Register R0 into memory location M ;

  MOV 4(R0), M stores the value contents(4+contents(R0)) into M.

Instruction costs

- Instruction cost = 1+cost for source and destination address modes.
- This cost corresponds to the length of the instruction.

- **Address modes involving registers have cost zero.**
- **Address modes involving memory location or literal have cost one.**

- **Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.**

Instruction costs

Example :

- ✔ MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.
- ✔ MOV R5 , M copies the contents of register R5 into memory location M. This instruction has cost two, since the address of memory location M is in the word following the instruction.
- ✔ ADD # 1 , R3 adds the constant I to the contents of register 3, and has cost two, since the constant 1 must appear in the next word following the instruction.
- ✔ SUB 4 (R0) , * 12 (R1) stores the value into the destination * 12 (R1) .

  The cost of this instruction is three, since the constants 4 and 12 are stored in the next two words following the intrusions ..

Instruction costs

The three-address statement a : = b + c can be implemented by many different instruction sequences :

```
i)  MOV b, R0
    ADD c, R0        cost = 6
    MOV R0, a
ii)  MOV b, a
     ADD c, a             cost = 6
```

Assuming R0, R1 and R2 contain the addresses of a, b, and c :

```
(iii) MOV *R1, *R0
      ADD *R2, *R0      cost = 2
iv) ADD R2, R1
     MOV R1, a                cost = 3
```

A Code Generator

- Generates target code for a sequence of three-address statements using next-use information
- For each operator in a statement there is a corresponding target language operator
- Computed results are kept in registers as long as possible, storing them only

  - if their Register is needed for another computation

  - just before a procedure call, jump or labeled statement

Register and Address Descriptors

- A *register descriptor* keeps track of what is currently in a register at a particular point in the code,

  e.g. a local variable, argument, global variable, etc.

  MOV a,R0 “**R0** contains **a**”

- An *address descriptor* keeps track of the location where the current value of the name can be found at run time,

  e.g. a register, stack location, memory address, etc.

  MOVa,R0

  MOV R0,R1 “**a** in **R0** and **R1**”

The Code Generation Algorithm

- Code Generation Algorithm takes as input a sequence of 3 AS constituting a basic block

For each statement *x* := *y* op *z*, *Perform the following*

- *Invoke a function getreg to determine the location L where the result of the computation y op z should be stored. L be a register/memory location.*
- Consult address descriptor for y to determine y', the current location of y

  -Prefer the register for y' if the value of y is currently both in memory and a register.

  -If the value of y is not already in L, generate the instruction MOV y', L to place a copy of y in L

The Code Generation Algorithm

- Generate **OP** *z’*, *L* where *z’* is the current location of *z*

  - Prefer the register for z’ if the value of z is currently both in memory and a register.

  - Update address descriptor of *x* to indicate that *x is in location L*

  - *If L is a register , update its descriptor to indicate that it contains the value of x , and remove x from all other register descriptors.*

- *If the current value of y and/or z have no next uses , are not live on exit from the block and are in registers, alter the register descriptor to indicate that , after execution of x := y* op *z, those registers no longer will contain y and / or z respectively.*

The Code Generation Algorithm

If the current 3AS has a unary operator (x:=op y), steps are same, omit some details

If the 3AS x:=y

-If y is in a register, simply change the register and address descriptors to record that the value of x is now found only in register holding the value of y

- if y has no next uses and is not live on exit from the block , the register no longer holds the value of y

- If y is in memory , use getreg to find a register in which to load y and make that register the location of x

The getreg Algorithm

- Function getreg returns the location L to hold the value of x for statement

  x := *y* op *z*

– If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* is not live and has no next use after execution of *x* := *y* op *z*, then return *register of y for L.* Update address descriptor of y to indicate that value *y* is no longer in *L*

– Else, return a new empty register for L if available

– Else ,if x has a next use in the block ,or op is an operator, such as indexing, that requires a register, find an occupied register *R.* Store the value of R in to a memory location (by **MOV** *R*,*M) if it is not already in the memory location M, update the* address descriptor for M and Return register *R*

– *If x not used in the block , or no suitable register can be found, select memory location of x as L*

Code Generation Example

- Assignment statement d:=(a - b) + (a - c) + (a - c)
- Corresponding 3AS statement

```
t:=  a - b
u:= a - c
v:=  t + u
d:= v + u
```

- Assume - a, b, c are always in memory
- Assume – t, u, v being temporaries

Code Generation Example…

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*
		Registers empty	
t := a - b	**MOV a,R0** **SUB b,R0**	**R0** contains **t**	**t** in **R0**
u := a - c	**MOV a,R1** **SUB c,R1**	**R0** contains **t** **R1** contains **u**	**t** in **R0** **u** in **R1**
v := t + u	**ADD R1,R0**	**R0** contains **v** **R1** contains **u**	**u** in **R1** **v** in **R0**
d := v + u	**ADD R1,R0** **MOV R0,d**	**R0** contains **d**	**d** in **R0** **d** in **R0** and memory