UNIT III INTERMEDIATE CODE GENERATION 8

Syntax Directed Definitions, Evaluation Orders for Syntax Directed Definitions, Intermediate Languages: Syntax Tree, Three Address Code, Types and Declarations, Translation of Expressions, Type Checking.

SYNTAX – DIRECTED TRANSLATION

Syntax-Directed Translations

- Translation of languages guided by CFGs
- Information associated with programming language constructs
  - Attributes attached to grammar symbols
  - Values of attributes computed by "semantic rules" associated with grammar productions
- Two notations for associating semantic rules
  - Syntax-directed definitions
  - Translation schemes

Semantic Rules

- Semantic rules perform various activities:
  - Generation of code
  - Save information in a symbol table
  - Issue error messages
  - Other activities
- Output of semantic rules is the translation of the token stream

Conceptual View

$$\begin{tikzcd}
\text{input string} \arrow[r] & \text{parse tree} \arrow[r] & \text{dependency graph} \arrow[r] & \text{evaluation order for semantic rules}
\end{tikzcd}$$

- Implementations do not need to follow outline literally
- Many "special cases" can be implemented in a single pass

SYNTAX DIRECTED DEFINITIONS

Syntax directed definition is a generalization of a context free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called synthesized attributes and inherited attributes.

Attributes

- Each grammar symbol (node in parse tree) has attributes attached to it ex: a string, a number, a type, a memory location etc.
- Values of a Synthesized attributes at a node is computed from the values of attributes at the children of that node in the parse tree.
- Values of a Inherited attributes at a node is computed from the values of attributes at the siblings and parent of that node.
  A dependency graph represents dependencies between attributes
  A parse tree showing the values of attributes at each node is an **annotated parse tree**
- Each semantic rule for production A -> α has the form
  b := f(c_1, c_2, …, c_k)
  - f is a function
  - b may be a synthesized attribute of A or
  - b may be an inherited attribute of one of the grammar symbol on the right side of the production
  - c_1, c_2, …, c_k are attributes belonging to grammar symbols of production
- An attribute grammar is one in which the functions in semantic rule cannot have side effects
  NOTE: a semantic rule may have side effects ex: printing a value or updating a global variable.

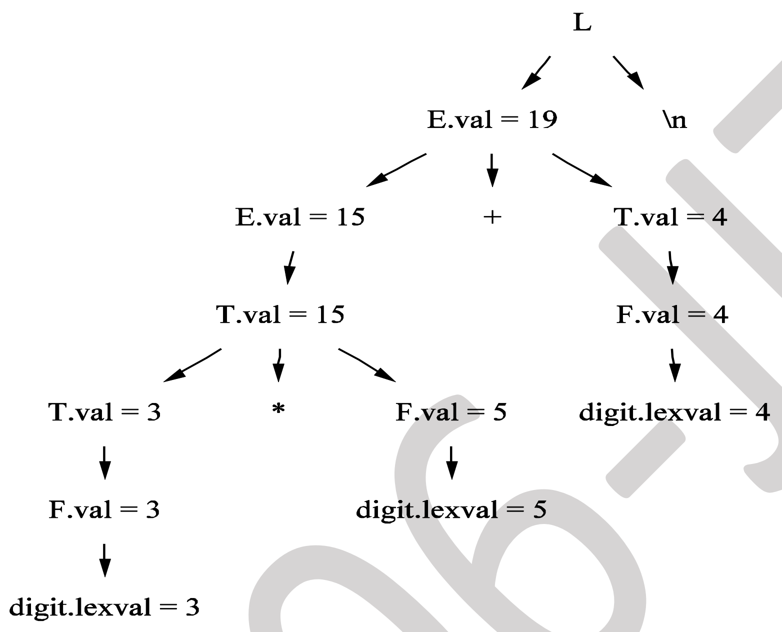
S-attributed Definitions

- Synthesized attributes are used extensively in practice
- S-attributed definition: A syntax-directed definition using only synthesized attributes
- Parse tree can be annotated by evaluation nodes during a single **bottom up pass**
  S-attributed Definition Example
  Desk calculator

Production	Semantic Rules
L → E n	print(E.val)
E → E_1 + T	E.val := E_1.val + T.val
E → T	E.val := T.val
T → T_1 * F	T.val := T_1.val * F.val
T → F	T.val := F.val

F → (E)	F.val := E.val
F → digit	F.val := digit.lexval

Annotated Parse Tree Example



NOTE

In a syntax directed definations, terminals are assumed to have
Synthesized attributes only,as the definations does not provide any semantic rules for terminals.values for attributes of terminals are usually supplied by the lexical analyser.Start symbol is assumed not to have any inherited attribute otherwise stated.

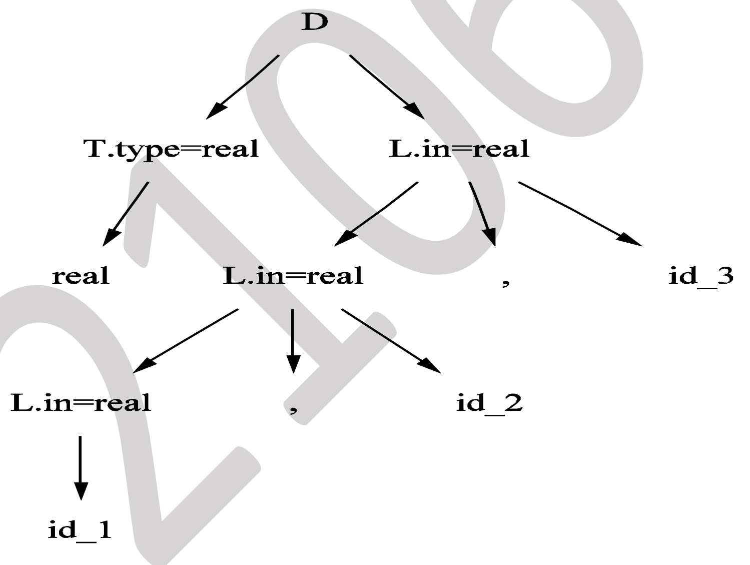
Inherited Attributes

- Inherited Attributes:
  - Value at a node in a parse tree depends
  on attributes of parent and/or siblings
  - Convenient for expressing dependencies of programming language constructs on context
- It is always possible to avoid inherited attributes, but they are often convenient

Inherited Attributes Example

Production	Semantic Rules
D → T L	L.in := T.type
T → int	T.type := integer
T → real	T.type := real
L → L_1, id	L_1.in := L.in addtype(id.entry, L.in)
L → id	addtype(id.entry, L.in)

Annotated Inherited Attributes



Dependency Graphs

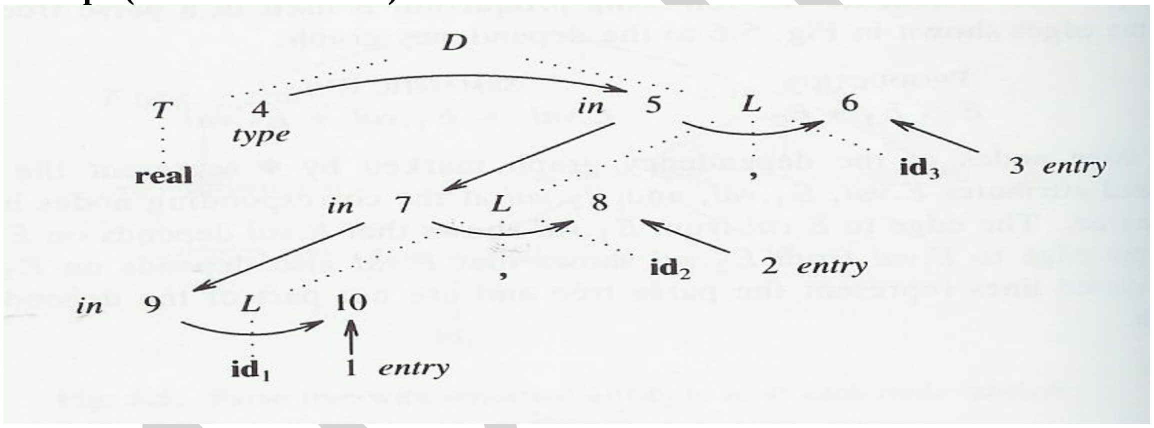
- Dependency graph:
  - Depicts interdependencies among synthesized and inherited attributes
  - Includes dummy nodes for procedure calls
- Numbered with a topological sort

- If $m_i \rightarrow m_j$ is an edge from m_i to m_j, then m_i appears before m_j in the ordering
- Gives valid order to evaluate semantic rules

Creating a Dependency Graph

```
for each node n in parse tree
  for each attribute a of grammar symbol at n
    construct a node in dependency graph for a
for each node n in parse tree
  for each semantic rule b := f(c1, c2, …, ck)
         associated with production used at n
    for i := 1 to k
      construct edge from node for ci to node for b
```

Example(inherited attribute)



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 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

- We 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.

- We evaluate the values of the attributes during reductions.

Bottom-Up Evaluation Example

Production	Code Fragment
(1) L → E \n	Print(val[top])
(2) E → E_1 + t	val[ntop] := val[top-2] + val[top]
(3) E → T	
(4) T → `T_1 * F	val[ntop] := val[top-2] * val[top]
(5) T → F	
(6) F → (E)	val[ntop] := val[top-1]
(7) F → digit	

Bottom-Up Evaluation Example

Input	State	Val	Rule
3*5+4\n	---	---	
*5+4\n	3	3	
*5+4\n	F	3	(7)
*5+4\n	T	3	(5)
5+4\n	T*	3_	

+4\n	T*5	3_5	
+4\n	T*F	3_5	(7)
+4\n	T	3_5	(4)
+4\n	E	15	(3)
4\n	E+	15_	
\n	E+4	15_4	
\n	E+F	15_4	(7)
\n	E+T	15_4	(5)
\n	E	19	(2)
	E\n	19_	
	L	19	(1)

TOP DOWN EVALUATION OF L-ATTRIBUTED DEFINITION

A top-down parser can evaluate attributes as it parses if the attribute values can be computed in a top-down fashion. Such attribute grammars are termed *L-Attributed*. First, we introduce a new type of symbol called an *action* symbol. Action symbols appear in the grammar in any place a terminal or nonterminal may appear. They may also have their own attributes. They may, however, be pushed onto their own stack, called a semantic stack or attribute stack.

We illustrate action symbols using the notation "<>" which indicates that the symbol within the brackets is to be pushed onto the semantic stack when it appears at the top of the parse stack. By inserting this action in appropriate places, we will create a translator which converts from infix expressions to postfix expressions.

EXAMPLE 6 Converting from infix to postfix via an action symbol

```
E  →  TE'
E  →  T
E' → + T E' <+>
        | ε
T  → F T'
T  → F
T' → * F T' <*>
        | ε
F  → (E)
      | Lit    <Lit>
      | Id     <Id>
```

We parse and translate *a* + *b* * *c*. The top is on the left for both stacks.

Parse Stack	Input	Semantic Stack
E $	a + b * c $	
T E' $	a + b * c $	
F E' $	a + b * c $	
a <a> E' $	a + b * c $	
E' $	+ b * c $	
E' $	+ b * c $	<a>
+ T E' <+> $	+ b * c $	<a>
T E' <+> $	b * c $	<a>
F T' E' <+> $	b * c $	<a>
b <b> T' E' <+> $	b * c $	<a>
T' E' <+> $	* c $	<b> <a>
* F T' <*> E' <+> $	* c $	<b> <a>
F T' <*> E' <+> $	c $	<b> <a>
c <c> T' <*> E' <+> $	c $	<b> <a>
T' <*> E' <+> $	$	<c> <b> <a>
ε <*> E' <+> $	$	<c> <b> <a>
<*> E' <+> $	$	<c> <b> <a>
E' <+> $	$	<*> <c> <b> <a>
ε <+> $	$	<*> <c> <b> <a>
<+> $	$	<*> <c> <b> <a>
$	$	<+> <*> <c> <b> <a>

When the semantic stack is popped, the translated string is:

a b c * +

the input string translated to postfix. In Example 6, the action symbol did not have any attached attributes.

The BNF in Example 6 is in LL(1) form. This is necessary for the top-down parse.

The formal definition of an L-attributed grammars is as follows. An attribute grammar is *L-attributed* if and only if for each production $X_0 \rightarrow X_1\ X_2 \ldots X_i \ldots X_n$,

(1) $\{X_i.\mathit{inh}\} = f(\{X_j.\mathit{inh}\}, \{X_k.\mathit{att}\})$ $\quad$ i, j >= 1, 0<=k<i

(2) $\{X_0.\mathit{syn}\} = f(\{X_0.\mathit{inh}\}, \{X_j.\mathit{att}\})$ $\quad$ 1<=j<=n

(3) {ActionSymbol.*Syn*} = f ({ActionSymbol.*Inh*})

(1) says that each ***inherited*** attribute of a symbol on the right-hand side depends only on inherited attributes of the right-hand side and arbitrary attributes of the symbols to the *left* of the given right-hand side symbol.

(2) says that each ***synthesized*** attributes of the left-hand-side symbol depends only on inherited attributes of that symbol and arbitrary attributes of right-hand-side symbols.

(2) says that the **synthesized** attributes of any action symbol depend only on the inherited attributes of the action symbol.

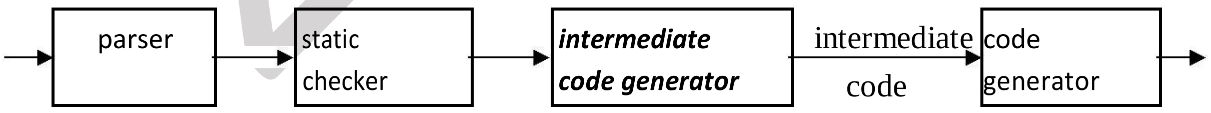
INTRODUCTION – Intermediate Code generator

The front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Position of intermediate code generator



<u>**INTERMEDIATE LANGUAGES**</u>

Three ways of intermediate representation:

- Syntax tree

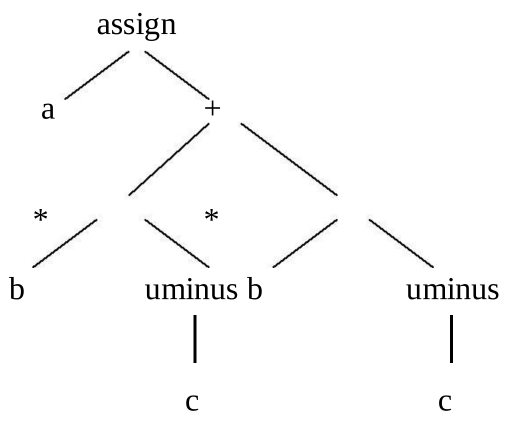
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

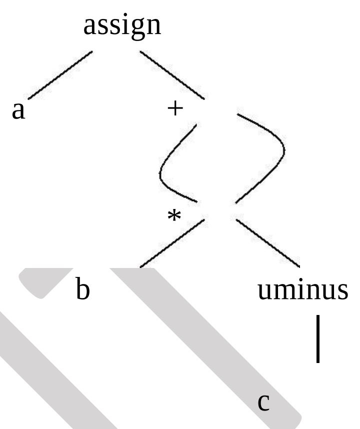
Graphical Representations:

SYNTAX TREE:

A syntax tree depicts the natural hierarchical structure of a source program. A **dag (Directed Acyclic Graph)** gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement **a : = b * - c + b * - c** are as follows:



(a) Syntax tree



(b) Dag

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

Syntax-directed definition for construction of Syntax Tree:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and * are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a : = b * - c + b* - c.

PRODUCTION	SEMANTIC RULE
S → id : = E	S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr)
E → E_1 +E_2	E.nptr : = mknode('+', E_1.nptr, E_2.nptr)
E → E_1 * E_2	E.nptr : = mknode('*', E_1.nptr, E_2.nptr)

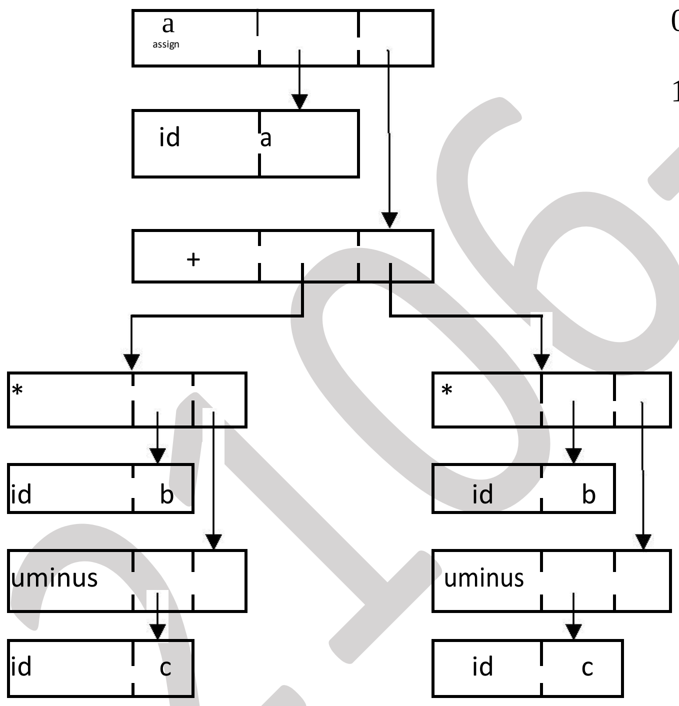
$E \rightarrow - E_1$	E.nptr : = mknode('uminus', E_1.nptr)
$E \rightarrow (E_1)$	E.nptr : = E_1.nptr
$E \rightarrow id$	E.nptr : = mkleaf(id, id.place)

Syntax-directed definition to produce syntax trees for assignment statements

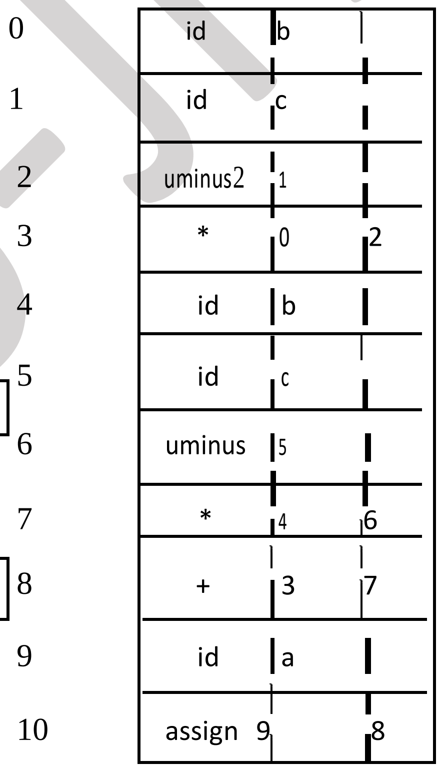
The token **id** has an attribute *place* that points to the symbol-table entry for the identifier. A symbol-table entry can be found from an attribute **id**.*name*, representing the lexeme associated with that occurrence of **id.** If the lexical analyzer holds all lexemes in a single array of characters, then attribute *name* might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.

Two representations of the syntax tree



(a)



0	id	b	
1	id	c	
2	uminus2	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8

(b)

THREE-ADDRESS CODE:

Three-address code is a sequence of statements of the general form x : = y *op* z

where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into asequence

t_1 : = y * z
t_2 : = x + t_1

where t_1 and t_2 are compiler-generated temporary names.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged – unlike postfix notation.

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. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

Three-address code corresponding to the syntax tree and dag given above

$t_1 := - c$	$t_1 := -c$
$t_2 := b * t_1$	$t_2 := b * t_1$
$t_3 := - c$	$t_5 := t_2 + t_2$
$t_4 := b * t_3$	$a := t_5$
$t_5 := t_2 + t_4$	
$a := t_5$	
(a) Code for the syntax tree	**(b) Code for the dag**

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three -Address Statements:

The common three-address statements are:

1. **Assignment statements** of the form **x : = y *op* z**, where ***op*** is a binary arithmetic or logical operation.

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 a fixed-point number to a floating-point number.

3.**Copy Statement*s*** of the form **x : = y** where the value of *y* is assigned to *x*.

3. The **unconditional jump goto** L. The three-address statement with label L is the next to be executed.

4. **Conditional jumps** such as **if *x relop y* goto L**. This instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x* stands in relation

relop to y. If not, the three-address statement following if *x relop y* goto L is executed next, as in the usual sequence.

6. ***param x* and *call p, n*** for procedure calls and *return y*, where y representing a returned value is optional. For example,

```
param  x1
param x2
  . . .
 param  xn
 call p,n
```

generated as part of a call of the procedure p(x_1, x_2, …. ,x_n).

7. **Indexed assignments** of the form x : = y[i] and x[i] : = y.

8. **Address and pointer assignments** of the form x : = &y , x : = *y, and *x : = y.

Syntax-Directed Translation into Three-Address Code:

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. For example, **id : = *E*** consists of code to evaluate ***E*** into some temporary t, followed by the assignment **id**.*place* : = **t.**

Given input a : = b * - c + b * - c, the three-address code is as shown above. The synthesized attribute *S.code* represents the three-address code for the assignment *S.* The nonterminal *E* has two attributes :

1. *E.place*, the name that will hold the value of E , and
2. *E.code*, the sequence of three-address statements evaluating E.

Syntax -directed definition to produce three-address code for assignments

PRODUCTION	SEMANTIC RULES
S → id : = E	***S.code : = E.code || gen(id.place ':=' E.place)***
E → E₁ + E₂	***E.place := newtemp;*** ***E.code := E₁.code || E₂.code || gen(E.place ':=' E₁.place '+' E₂.place)***
E → E₁ * E₂	***E.place := newtemp;***

	E.code := E1.code || E2.code || gen(E.place ':=' E1.place '*' E2.place)
E → - E1	***E.place := newtemp;*** ***E.code := E1.code || gen(E.place ':=' 'uminus' E1.place)***
E → (E1)	***E.place : = E1.place;*** ***E.code : = E1.code***
E → id	***E.place : = id.place;*** ***E.code : = ' '***

- The function *newtemp* returns a sequence of distinct names $t_1, t_2, \ldots$ in response to successive calls.
- Notation *gen(x ':=' y '+' z)* is used to represent three-address statement x := y + z. Expressions appearing instead of variables like *x, y* and *z* are evaluated when passed to *gen*, and quoted operators or operand, like '+' are taken literally.
- Flow-of–control statements can be added to the language of assignments. The code for ***S*** **→ while *E* do *S₁*** is generated using new attributes *S.begin* and *S.after* to mark the first statement in the code for *E* and the statement following the code for S, respectively.
- The function *newlabel* returns a new label every time it is called.
- We assume that a non-zero expression represents true; that is when the value of ***E*** becomes zero, control leaves the while statement.

Write three address code for the below expression.

a=b+c*d-e/f

```
t1=c*d

t2=e/f

t3=b+t1

t4=t3-t2

a=t4
```

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. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which are, ***op, arg1, arg2*** and ***result.***
- The *op* field contains an internal code for the operator. The three-address statement **x : = y op z** is represented by placing *y* in *arg1*, *z* in *arg2* and *x* in *result*.
- 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.

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: *op, arg1* and *arg2*.
- The fields *arg1* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as *triples*.

(a) Quadruples

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

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

(b) Triples

Quadruple and triple representation of three-address statements given above

A ternary operation like x[i] : = y requires two entries in the triple structure as shown as below while x : = y[i] is naturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ] =	X	i
(1)	assign	(0)	y

(a) x[i] : = y

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

(b) x : = y[i]

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

Declarations in a Procedure:

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say *offset*, can keep track of the next available relative address.

In the translation scheme shown below:

- Nonterminal P generates a sequence of declarations of the form **id :** ***T.***
- Before the first declaration is considered, *offset* is set to 0. As each new name is seen , that name is entered in the symbol table with offset equal to the current value of *offset*, and *offset* is incremented by the width of the data object denoted by that name.
- The procedure *enter(name, type, offset*) creates a symbol-table entry for *name*, gives its type *type* and relative address *offset* in its data area.
- Attribute *type* represents a type expression constructed from the basic types *integer* and *real* by applying the type constructors pointer and array. If type expressions are represented by graphs, then attribute *type* might be a pointer to the node representing a type expression.
- The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be 4.

Computing the types and relative addresses of declared names

P → D	***{ offset : = 0 }***
D → D ; D	
D → id : T	***{ enter(id.name, T.type, offset); offset : = offset + T.width }***
T → integer	***{ T.type : = integer; T.width : = 4 }***
T → real	***{ T.type : = real; T.width : = 8 }***

T → array [num] of T_1	***{ T.type : = array(num.val, T_1.type); T.width : = num.val X T_1.width }***
T → ↑ T_1	***{ T.type : = pointer (T_1.type); T.width : = 4 }***

Keeping Track of Scope Information:

When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:

$P \rightarrow D$

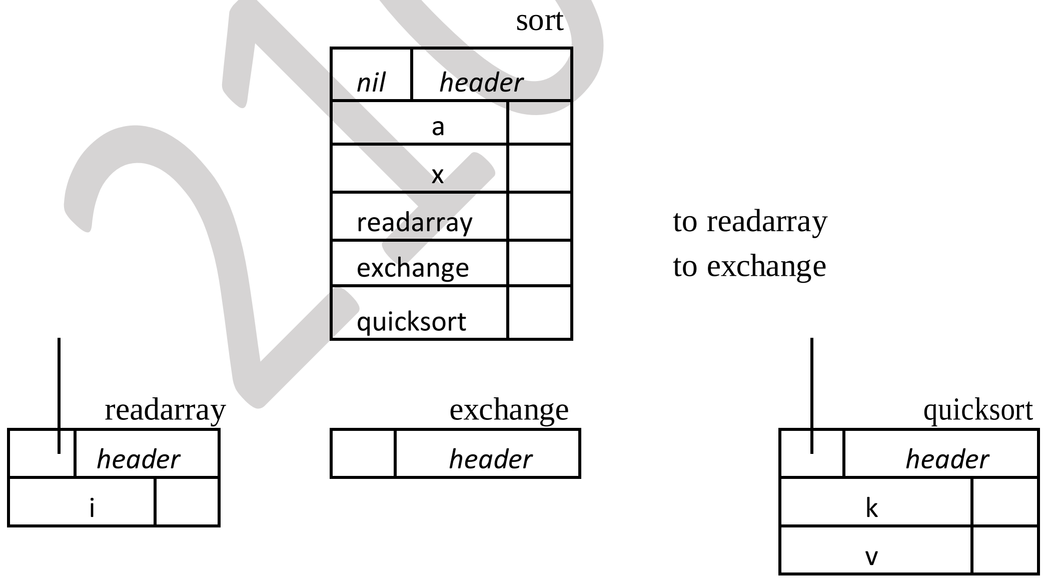
$D \rightarrow D\ ;\ D$ | **id** $: T$ | **proc id** $;\ D\ ;\ S$

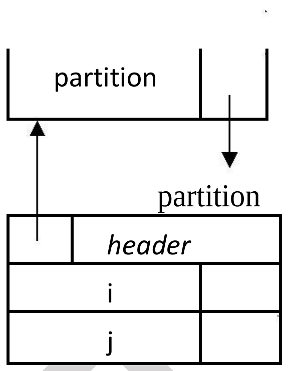
One possible implementation of a symbol table is a linked list of entries for names.

A new symbol table is created when a procedure declaration $D \rightarrow$ ***proc id*** $D_1;S$ is seen, and entries for the declarations in D_1 are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations is that the procedure *enter* is told which symbol table to make an entry in.

For example, consider the symbol tables for procedures *readarray*, exchange, and *quicksort* pointing back to that for the containing procedure sort, consisting of the entire program. Since *partition* is declared within *quicksort*, its table points to that of *quicksort*.

Symbol tables for nested procedures




partition
partition
header
i
j

The semantic rules are defined in terms of the following operations:

1. *mktable(previous)* creates a new symbol table and returns a pointer to the new table. The argument *previous* points to a previously created symbol table, presumably that for the enclosing procedure.

2. *enter(table, name, type, offset)* creates a new entry for name *name* in the symbol table pointed to by *table.* Again, *enter* places type *type* and relative address *offset* in fields within the entry.

3. *addwidth(table, width)* records the cumulative width of all the entries in table in the header associated with this symbol table.

4. *enterproc(table, name, newtable)* creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.

Syntax directed translation scheme for nested procedures

```
P → M D                      { addwidth ( top( tblptr) , top (offset));
                               pop (tblptr); pop (offset) }

M → ε                        { t : = mktable (nil);
                               push (t,tblptr); push (0,offset) }

D → D1 ; D2

D → proc id ;  N D1 ; S   { t : = top (tblptr);
                               addwidth ( t, top (offset));
                               pop (tblptr); pop (offset);
                               enterproc (top (tblptr), id.name, t) }

D → id : T                   { enter (top (tblptr), id.name, T.type, top (offset));
                               top (offset) := top (offset) + T.width }

N → ε                        { t := mktable (top (tblptr));
                               push (t, tblptr); push (0,offset) }
```

- The stack *tblptr* is used to contain pointers to the tables for **sort, quicksort,** and **partition** when the declarations in **partition** are considered.

- The top element of stack *offset* is the next available relative address for a local of the current procedure.
- All semantic actions in the subtrees for B and C in

  $$A \rightarrow BC \{action_A\}$$

  are done before $action_A$ at the end of the production occurs. Hence, the action associated with the marker M is the first to be done.

- The action for nonterminal M initializes stack *tblptr* with a symbol table for the outermost scope, created by operation *mktable(nil)*. The action also pushes relative address 0 onto stack offset.
- Similarly, the nonterminal N uses the operation *mktable(top(tblptr))* to create a new symbol table. The argument *top(tblptr)* gives the enclosing scope for the new table.
- For each variable declaration **id:** T, an entry is created for **id** in the current symbol table. The top of stack offset is incremented by T.width.
- When the action on the right side of $D \rightarrow proc\ id;\ ND_1;\ S$ occurs, the width of all declarations generated by D_1 is on the top of stack offset; it is recorded using *addwidth*. Stacks *tblptr* and *offset* are then popped.
  At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (**and, or,** and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 **relop** E_2, where E_1 and E_2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

E → E **or** E | E **and** E | **not** E | (E) | **id relop id | true | false**

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

- To encode true and false ***numerically*** and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by ***flow of control***, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly

convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

- The translation for
  a **or** b **and not** c is the three-address sequence
  t_1 : = **not** c t_2
  : = b **and** t_1 t_3
  : = a **or** t_2
- A relational expression such as a < b is equivalent to the conditional statement
  if a < b then 1 else 0

which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

```
100 :  if a < b goto 103
101 :  t : = 0
102 :  goto 104
103 :  t : = 1
104 :
```

Translation scheme using a numerical representation for booleans

E → *E₁* **or** *E₂*	*{ E.place : = newtemp;*
	 emit(E.place '': ='' E₁.place '**or**'*E₂.place)}*

E → *E₁* **and** *E₂*	*{ E.place : = newtemp;*
	 emit(E.place '': ='' E₁.place '**and**'*E₂.place)}*

E → **not** *E₁*	*{ E.place : = newtemp;*
	 *emit(E.place '': ='' '***not***' E ₁.place)}*

E → (*E*$_1$)	{ *E.place : = E*$_1$*.place* }
E → **id**$_1$ **relop** **id**$_2$	{ *E.place : = newtemp;*
	emit('if' **id**$_1$*.place* **relop**.*op* **id**$_2$*.place* '**goto**' *nextstat* + **3***);*
	*emit(E.place ': =' '***0***');*
	*emit('***goto***' nextstat +***2***);*
	*emit(E.place ': =' '***1***') }*
E → **true**	{ *E.place : = newtemp;*
	*emit(E.place ': =' '***1***') }*
E → **false**	{ *E.place : = newtemp;*
	*emit(E.place ': =' '***0***') }*

Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called "**short-circuit**" or "**jumping**" code. It is possible to evaluate boolean expressions without generating code for the boolean operators **and, or,** and **not** if we represent the value of an expression by a position in the code sequence.

Translation of a < b or c < d and e < f

```
100 :  if a < b goto   103          107 : t2 : = 1
101 :   t1 : = 0                    108 : if e < f goto 111
102 :  goto 104                     109 : t3 : = 0
103 :  t1 : = 1                     110 : goto 112
104 :  if c < d goto   107          111 : t3 : = 1
105 :  t2 : = 0                     112 : t4 : = t2 and t3
106 :  goto 108                     113 : t5 : = t1 or t4
```

Control-Flow Translation of Boolean Expressions:

With the help of control flow mechanism, the Boolean operator and conditional statements in which Boolean expression are part of it are translated into three address code as follows.

Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1$ **or** E_2	E_1*.true : = E.true;* E_1*.false : = newlabel;* E_2*.true : = E.true;* E_2*.false : = E.false;*

	E.code : = E_1.code || gen(E_1.false ':') || E_2.code
E → *E_1* **and** *E_2*	*E_1.true : = newlabel;* *E_1.false : = E.false;* *E_2.true : = E.true;* *E_2.false : = E.false;* *E.code : = E_1.code || gen(E_1.true ':') || E_2.code*
E → **not** *E_1*	*E_1.true : = E.false;* *E_1.false : = E.true;* *E.code : = E_1.code*

E → *(E1)*	$E_1.true := E.true;$ $E_1.false := E.false;$ $E.code := E_1.code$
E → **id**$_1$ **relop id**$_2$	*E.code : = gen*(‘**if**’ **id**$_1$.*place* **relop.***op* **id**$_2$.*place* ‘**goto**’ *E.true) || gen*(‘**goto**’ *E.false*)
E → **true**	*E.code : = gen(*‘**goto**’ *E.true)*
E → **false**	*E.code : = gen(‘***goto***’ E.false)*

Flow-of-Control Statements

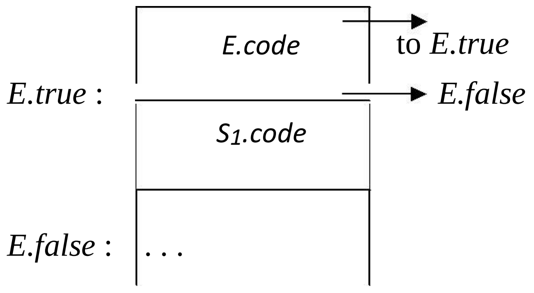
We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S → **if** E **then** S_1
| **if** E **then** S_1 **else** S_2
| **while** E **do** S_1

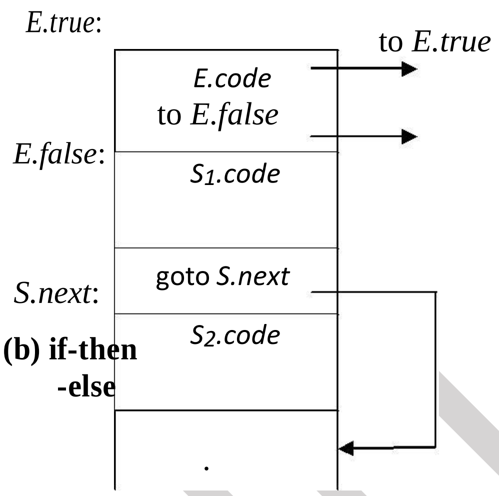
In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

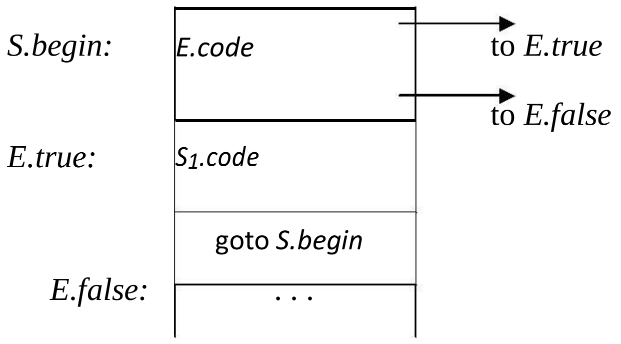
Code for if-then , if-then-else, and while-do statements



(a) if-then



(b) if-then -else



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES

$S \rightarrow$ **if** E **then** S_1	*E.true : = newlabel;*
	E.false : = S.next;
	S_1*.next : = S.next;*
	S.code : = E.code || gen(E.true ':') || S_1*.code*

$S \rightarrow$ **if** E **then** S_1 **else** S_2	*E.true : = newlabel;* *E.false : = newlabel;* S_1*.next : = S.next;* S_2*.next : = S.next;* *S.code : = E.code || gen(E.true ':') ||* S_1*.code ||* *gen('***goto***' S.next) ||* *gen(E.false ':') ||* S_2*.code*

$S \rightarrow$ **while** E **do** S_1	*S.begin : = newlabel;*
	E.true : = newlabel;
	E.false : = S.next;
	S_1*.next : = S.begin;*
	S.code : = gen(S.begin ':')|| E.code ||

gen(E.true ':') || S_1.code ||
*gen('**goto**' S.begin)*

TYPE CHECKING

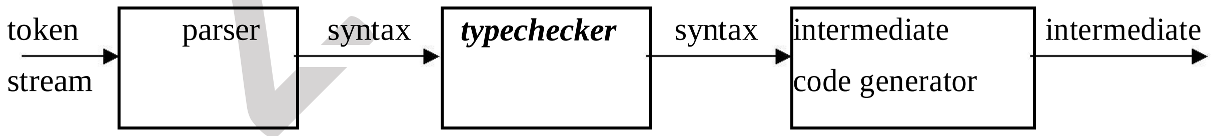
A compiler must check that the source program follows both syntactic and semantic conventions of the source language.
This checking, called *static checking*, detects and reports programming errors.

Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. **Flow-of-control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

Position of type checker



- A ***type checker*** verifies that the type of a construct matches that expected by its context. For example : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.

- Type information gathered by a type checker may be needed when code is generated.

TYPE SYSTEMS

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

- The type of a language construct will be denoted by a "type expression."
- A type expression is either a basic type or is formed by applying an operator called a ***type constructor*** to other type expressions.
- The sets of basic types and constructors depend on the language to be checked.

The following are the definitions of type expressions:

1. Basic types such as *boolean, char, integer, real* are type expressions.

   A special basic type, *type_error* , will signal an error during type checking; *void* denoting "the absence of a value" allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression.
   Constructors include:
   Arrays : If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

   Products : If T_1 and T_2 are type expressions, then their Cartesian product T_1 X T_2 is a type expression.
   Records : The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.

For example:

type row = record
address: integer;
lexeme: array[1..15] of char
end;
var table: array[1...101] of row;

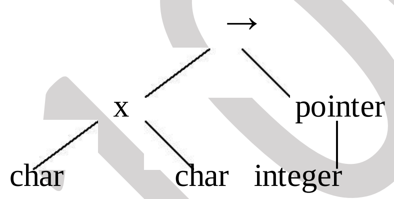
declares the type name *row* representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable *table* to be an array of records of this type.

Pointers : If T is a type expression, then *pointer*(T) is a type expression denoting the type "pointer to an object of type T".
For example, ***var p: ↑ row*** declares variable p to have type *pointer*(row).

Functions : A function in programming languages maps a *domain type D* to a *range type R*. The type of such function is denoted by the type expression D → R

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

Tree representation for char x char → *pointer* (integer)



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. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

- Checkingdone by a compiler is said to be static, while checking done when the target program runs is termed dynamic.

- Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

Sound type system

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type_error* to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

- Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.
- Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

SPECIFICATION OF A SIMPLE TYPE CHECKER

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. 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.

<u>A Simple Language</u>

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E
D → D ; D
D → id : T { *addtype* (id.*entry* , T.*type*)}
T → char { T.*type* : = char }
T → integer { T.*type* : = integer }
T → ↑ T1 { T.*type* : = pointer(T_1.*type*) }
T → array [num] of T1 { T.*type* : = array (1… num.val , T_1.*type*) }

In the above language,
→ There are two basic types : char and integer ;
→ *type_error* is used to signal errors;
→ the prefix operator ↑ builds a pointer type. Example , ↑ **integer** leads to the type expression **pointer (integer)**.

Type checking of expressions

In the following rules, the attribute *type* forE gives the type expression assigned to the expression generated by E.

1. E → **literal** { E.*type* : = *char* }
   E → **num** { E.*type* : = *integer* }
   Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. E → **id** { E.*type* : = *lookup* (**id**.*entry*) }
   lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E_1 **mod** E_2 { *E.type* : = **if** *E_1. type = integer* **and**
   E_2. type = integer **then** *integer*
   else *type_error* }
   The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

4. E → E_1 [E_2] { *E.type* : = if *E_2.type = integer* **and**
   E_1.type = array(s,t) **then** *t*
   else *type_error* }
   In an array reference E_1 [E_2] , the index expression E $_2$ must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E_1.

5. $E \rightarrow E_1 \uparrow$ { *E.type* : = **if** *E_1.type* = *pointer (t)* **then** *t*
else *type_error* }

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.

Type checking of statements

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:

$S \rightarrow$ **id** : = E { S.*type* : = **if id**.*type* = E.*type* **then** *void* **else**
type_error }

2. Conditional statement:

S $\rightarrow$ **if** E **then** S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

3. While statement:

S $\rightarrow$ while E do S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

4. Sequence of statements:

S $\rightarrow S_1 ; S_2$ { S.*type* : = **if** S_1.*type* = *void* and S_2.*type* = *void*
then *void*
else *type_error* }

Type checking of functions

The rule for checking the type of a function application is :

$E \rightarrow E_1 (E_2)$ { E.*type* : = **if** E_2.*type* **=** *s* **and**
E_1.*type* **=** $s \rightarrow t$ **then** *t*
else *type_error* }