COMPILER DESIGN

UNIT – III

Syntax Directed Definition:

Syntax Directed Definition (SDD) is a kind of abstract specification. It is generalization of context free grammar in which each grammar production **X –> a** is associated with it a set of production rules of the form s = f(b_1, b_2, ……b_k) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location.

Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

Example:

E --> E1 + T {E.val = E1.val + T.val}

Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features

- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

Types of attributes: There are two types of attributes:

1. Synthesized Attributes: These are those attributes which derive their values from their children nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children nodes in parse tree.

Example:

E --> E1 + T { E.val = E1.val + T.val}

In this, E.val derive its values from E_1.val and T.val

Computation of Synthesized Attributes –

- Write the SDD using appropriate semantic rules for each production in given grammar.
- The annotated parse tree is generated and attribute values are computed in bottom up manner.
- The value obtained at root node is the final output.

Example: Consider the following grammar

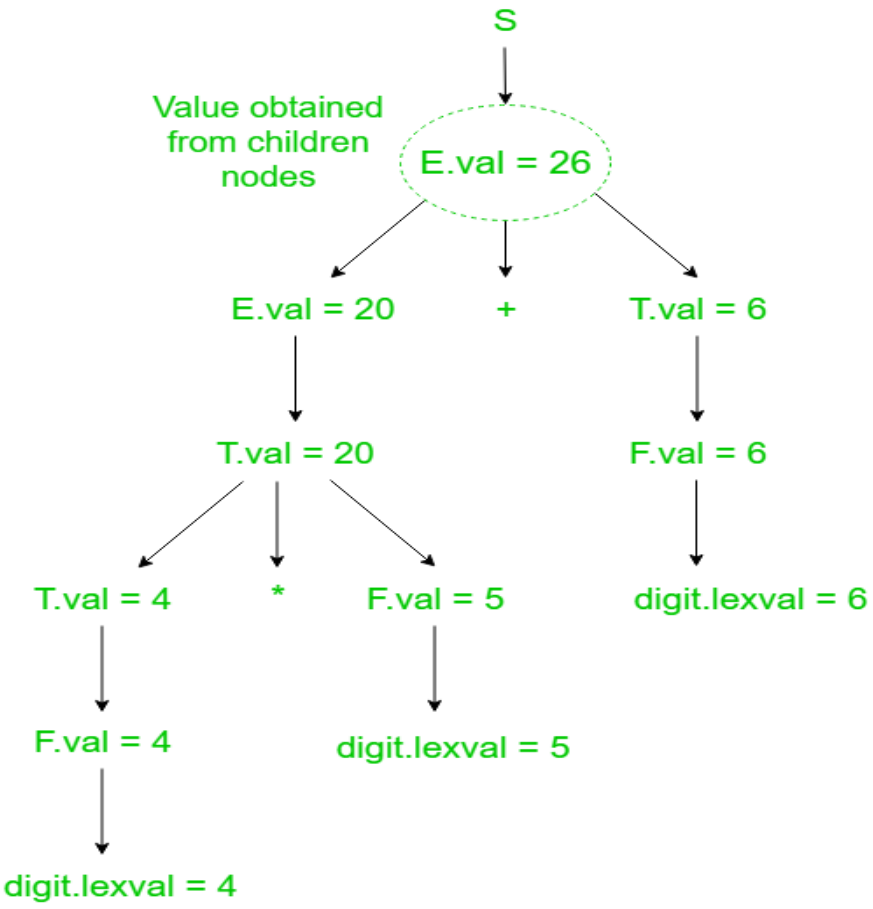
S --> E
E --> E1 + T
E --> T

T --> T1 * F
T --> F
F --> digit

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	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 --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is



Annotated Parse Tree

For computation of attributes we start from leftmost bottom node.

2. Inherited Attributes: These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or sibling nodes.

Example:

A --> BCD { C.in = A.in, C.type = B.type }

Computation of Inherited Attributes:

- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

Example: Consider the following grammar

S --> T L

T --> int

T --> float

T --> double

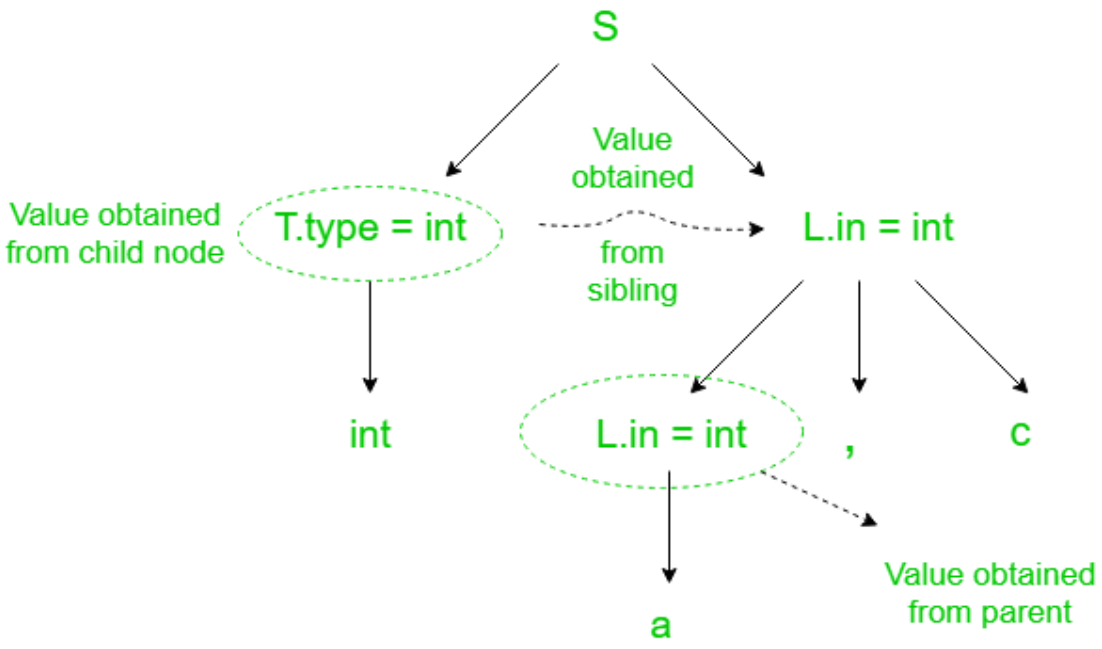
L --> L1, id

L --> id

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> T L	L.in = T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is



Annotated Parse Tree

The value of L nodes is obtained from T.type (sibling) which is basically lexical value obtained as int, float or double. Then L node gives type of identifiers a and c. The computation of type is done in top down manner or preorder traversal. Using function Enter_type the type of identifiers a and c is inserted in symbol table at corresponding id.entry.

Applications of Syntax-Directed Translation:

Here are some applications of SDT in Compiler Design:

1. Syntax Directed Translation is used for executing arithmetic expressions
2. Conversion from infix to postfix expression
3. Conversion from infix to prefix expression
4. For Binary to decimal conversion
5. Counting the number of Reductions
6. Creating a Syntax Tree
7. Generating intermediate code
8. Storing information into the symbol table
9. Type checking

Syntax Direct Translation (SDT):

Syntax Direct Translation (SDT) method is used in compiler design, associating the translation rules with grammar production. Syntax Directed Translation can identify informal notations, called semantic rules along with the grammar.

Conceptually, with both syntax-directed definition and translation schemes, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse tree nodes.

We can represent it as:

Grammar + semantic rule = SDT

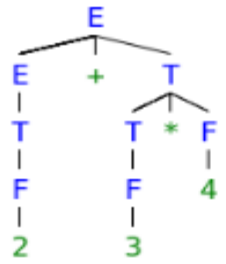
The 'val' attribute in the semantic rules may contain strings, numbers, memory location or a complex record.

S.No	Production	Semantic Rules
1.	E → E + T	E.val := E.val + T.val
2.	E → T	E.val := T.val
3.	T → T * F	T.val := T.val * F.val
4.	T → F	T.val := F.val
5.	F → (F)	F.val := F.val
6.	F → num	F.val := num.lexval

The right side of the translation rule is always in correspondence to the attribute values of the right side of the production rule. From this, we come to the conclusion that SDT in compiler design associates:

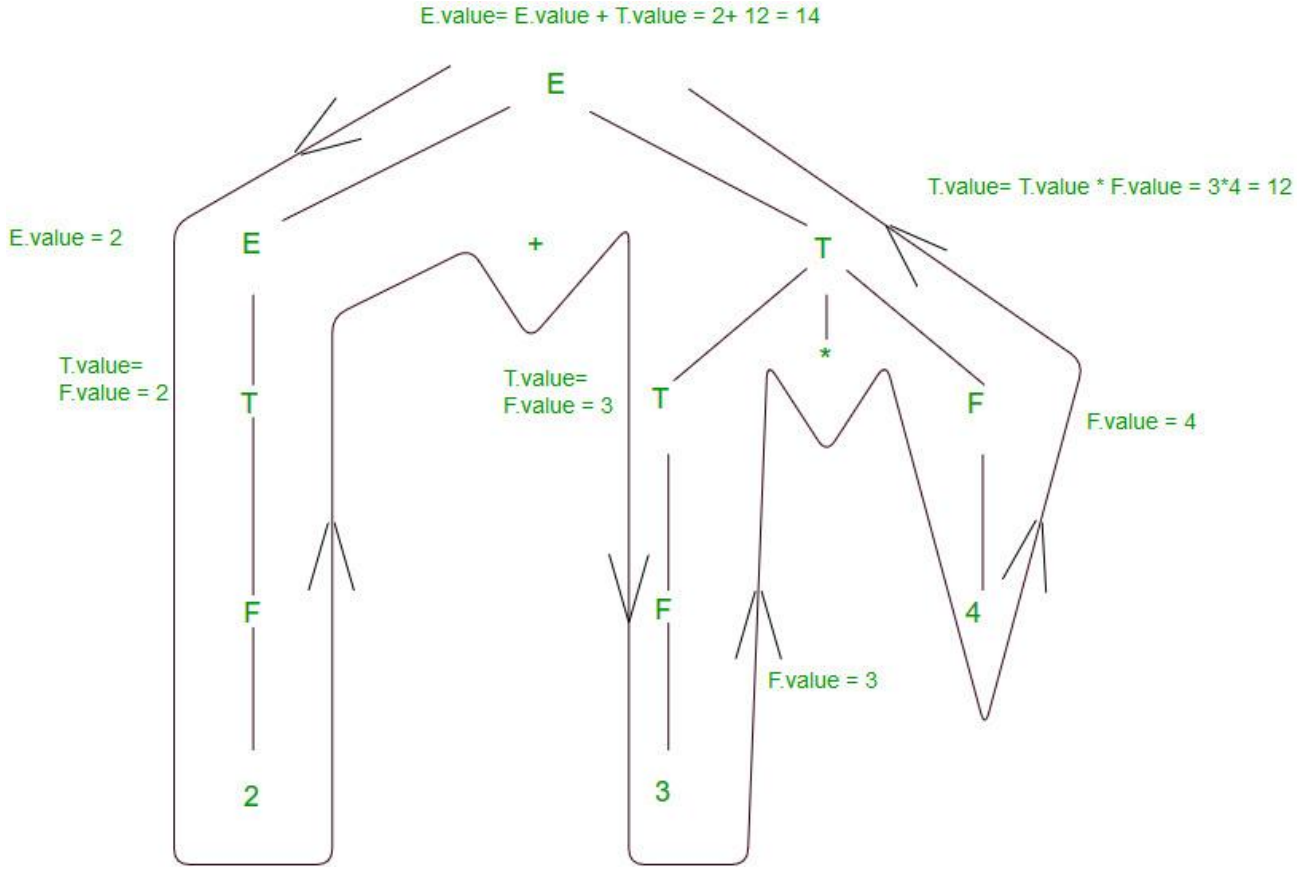
1. A set of attributes to every node in grammar.
2. A set of translation rules to every production rule with the help of attributes.

Let's take a string to see how semantic analysis happens – S = 2+3*4. Parse tree corresponding to S would be



To evaluate translation rules, we can employ one depth-first search traversal on the parse tree.

For better understanding, we will move bottom-up in the left to right fashion for computing the translation rules of our example.



The above diagram shows how semantic analysis could happen.

S – Attributed in Syntax directed translation:

S-attributed SDT:

The **S-attributed** definition is a type of syntax-directed attributes in compiler design that solely uses synthesized attributes. The symbol attribute values in the

production's body are used to calculate the attribute values for the non-terminal at the head.

The nodes of the parse tree can be ranked from the bottom up when evaluating an S-attributed SDD's attributes. i.e., by conducting a post-order traverse of the parse tree and evaluating the characteristics at a node once the traversal finally leaves that node.

Let us see an example of S-attributed SDT.

The grammar is given below:

$S \rightarrow E$

$E \rightarrow E1+T$

$E \rightarrow T$

$T \rightarrow T1*F$

$T \rightarrow F$

$F \rightarrow digit$

The S-attributed SDT of the above grammar can be written in the following way.

Production	Semantic Rules
S → E	S.val = E.val
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → digit	F.val = digit. lexval

- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

L – Attributed in Syntax directed translation:

L-attributed definitions are syntax-directed attributes in compiler design in which the edges of the dependency graph for the attributes in the production body can go from left to right and not from right to left. L-attributed definitions can inherit or synthesize their attributes.

If the traits are inherited, the calculation must come from the following:

- A quality that the production head inherited.
- By a production-related attribute, either inherited or synthesized, situated to the left of the attribute being computed.
- An inherited or synthesized attribute is linked to the attribute in question in a way that prevents cycles from forming in the dependency network.

Let us see an example of L-attributed SDT.

Example

The grammar is given below:

G → TL
T → int
T → float
T → double
L → Ll, id
L → id.

The L-attributed SDT of the above grammar can be written in the following way.

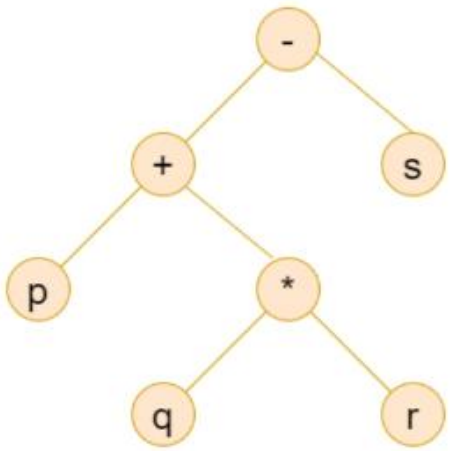
Production	Semantic Rules
G → T L	L.in = T.type
T → int	T.type = int
T → float	T.type = float
T → double	T.type = double
L → L1, id	L1.in = L.in Enter_type(id. entry, L.in)
L → id	Entry_type(id. entry, L.in)

- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

Variants of Syntax Tree:

A syntax tree is a tree in which each leaf node represents an operand, while each inside node represents an operator. The Parse Tree is abbreviated as the syntax tree. The syntax tree is usually used when representing a program in a tree structure.

Draw Syntax Tree for the string p + q * r − s.



Rules for Constructing Syntax Tree

The syntax tree nodes can all be treated as data with several fields. The operator is identified by one node element, whereas the remaining areas include a pointer to the operand nodes. The node's label is also known as the operator. The following functions are used to generate the syntax tree nodes for expressions using binary operators. Each method returns a pointer to the most recently developed node.

1. **mknode (op, left, right):** It creates an operator node with the name op and two fields, containing left and right pointers.
2. **mkleaf (id, entry):** It creates an identifier node with the label id and the entry field, which contains a reference to the identifier's symbol table entry.
3. **mkleaf (num, val):** It creates a number node with the name num and a field containing the number's value, val. For example, construct a syntax tree for an expression p − 4 + q. In this sequence, a1, a2, ... a5 are pointers to the symbol table entries for identifier 'p' and 'q' respectively.

   a1− mkleaf (id, entry p);
   a2− mkleaf (num, 4);
   a3− mknode ('−', a1, a2)
   a4− mkleaf(id, entry q)
   a5− mknode('+', a3, a4);

The tree is constructed from the ground up. The leaves for p and 4 are created using mkleaf (id, entry p) and mkleaf (num 4), respectively. a1 and a2 are used to hold references to these nodes. The internal node with the leaves for p and 4 as children is created by using mknodes ('', a1, a2). The syntax tree will look like this:

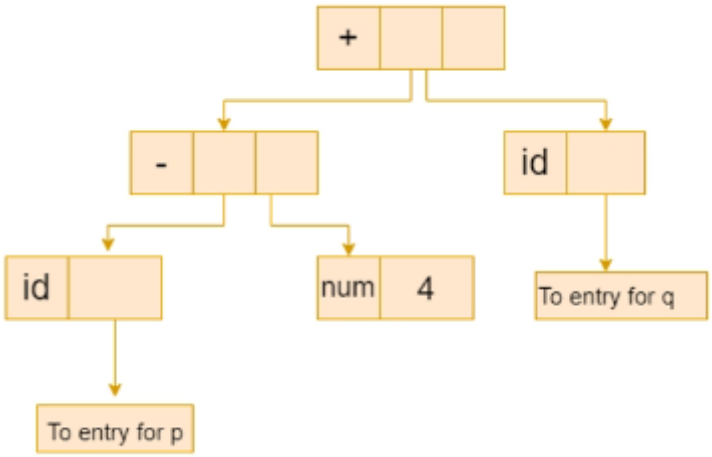


Fig: Syntax Tree for p - 4 + q

Directed Acyclic Graphs for Expressions (DAG):

A DAG is used in compiler design to optimize the basic block. It is constructed using Three Address Code. Then after construction, multiple transformations are applied such as dead code elimination, and common subexpression elimination. DAG's are useful in compilers because topological ordering can be defined in the case of DAGs, which is crucial for construction of object level code.

The following are some characteristics of DAG.

- DAG is a type of Data Structure used to represent the structure of basic blocks.
- Its main aim is to perform the transformation on basic blocks.
- The leaf nodes of the directed acyclic graph represent a unique identifier that can be a variable or a constant.
- The non-leaf nodes represent an operator symbol.

Three-Address Code:

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.

General Representation

a = b
a = op b
a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple – It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example: Consider expression a = b * – c + b * – c. The three address code is:

t1 = uminus c (Unary minus operation on c)
t2 = b * t1
t3 = uminus c (Another unary minus operation on c)
t4 = b * t3
t5 = t2 + t4
a = t5 (Assignment of t5 to a)

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. Triples – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example – Consider expression a = b * – c + b * – c

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

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

Indirect Triples representation

Types of Three-address codes

Three-address code is a sequence of statements of the general form **A := B op C**, where A, B, C are either programmer defined names, constants or compiler-generated temporary names; **op** stands for an operation which is applied on A, B.In simple words, a code having at most three addresses in a line is called three address codes.

t1=a+b

t2=t1+c

t3=t1*t2

These statements come under following seven categories and can be called as building block for three-address statements-

Statement	Meaning
X = Y op Z	**Binary Operation**
X= op Z	**Unary Operation**
X = Y	**Assignment**
if X(rel op)Y goto L	**Conditional Goto**
goto L	**Unconditional Goto**
A[i] = X **Y= A[i]**	**Array Indexing**
P = addr X **Y = *P** ***P = Z**	**Pointer Operations**

Types and Declarations:

Typical basic types and declarations for a language contain boolean, char, integer, float, and void; here, void denotes "the absence of a value." A type name is a type expression. We can form a type expression by applying the array type constructor to a number and a type expression.

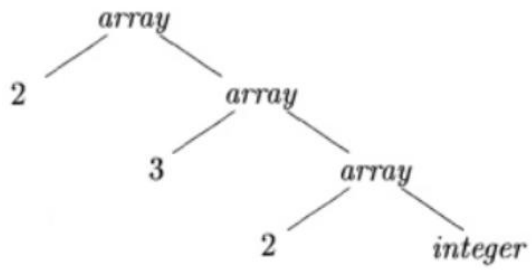
Declaration involves allocating space in memory and entering type and name in the symbol table. Memory allocation is consecutive, and it assigns names to memory in the sequence they are declared in the program.

Type Expressions

A primary type is a type of expression. Typical basic types for a language contain boolean, char, integer, float, and void; the void implies "the absence of a value."

We can form a type name or type expression by applying the array type constructor to a number and a type expression.

Types have structure, which we can represent using *type expressions:* a type expression is either a primary type or is formed by applying a type constructor operator to a type expression. The basic types and constructors depend on the language to be checked.



The array type int [2][3][2] can be read as an "array of 2 arrays each of 3 arrays of 2 integers each" and written as a type expression array(2, array(3, array(2, integer))).

Basic types

char, int, double, float are type expressions.

Type names

It is convenient to consider that names of types are type expressions.

Arrays

If *E* is a type expression and *n* is an int, then

array(*n*, *E*)

is a type expression indicating the type of an array with elements in *T* and indices in the range 0 ... *n* - 1.

Products

If *E*1 and *E*2 are type expressions then

*E*1 × *E*2

is a type expression indicating the type of an element of the Cartesian product of *E*1 and *E*2. The products of more than two types are defined similarly, and this product operation is left-associative. Hence

(*E*1 × *E*2) × *E*3 and *E*1 × *E*2 × *E*3

are equivalent.

Records

The only difference between a record and a product is that the fields of a record have names.

If *abc* and *xyz* are two type names, if *E*1 and *E*2 are type expressions then

record(*abc*: *E*1, *xyz*: *E*2)

is a type expression indicating the type of an element of the Cartesian product of $T1$ and $T2$.

Pointers

Let E is a type expression, then

pointer(E)

is a type expression denoting the type *pointer to an object of type T*.

Function types

If $E1$ and $E2$ are type expressions then

$E1$ -> $E2$

is a type expression denoting the type of functions associating an element from $E1$ with an element from $E2$.

Type Equivalence

When graphs represent type expressions, two types are structurally equivalent if and only if one of the following conditions is true:

- They are of the same basic type.
- They are formed by applying the identical constructor to structurally equivalent types.
- One is a type name that refers to the other.

The first two conditions in the above definition lead to the name equivalence of type expressions if type names are interpreted as though they stand for themselves.

"If two type expressions are equal, return a specified type else error," says a common type-checking rule. Ambiguities might develop when type expressions are given names and then utilized in later type expressions.

Declarations

When we encounter declarations, we need to layout storage for the declared variables.

For every local name in a procedure, we create a Symbol Table(ST) entry including:

1. The type of the name
2. How much storage the name requires

Grammar:

D -> real, id

D -> integer, id

D -> D1, id

We use ENTER to enter the symbol table and use ATTR to trace the data type.

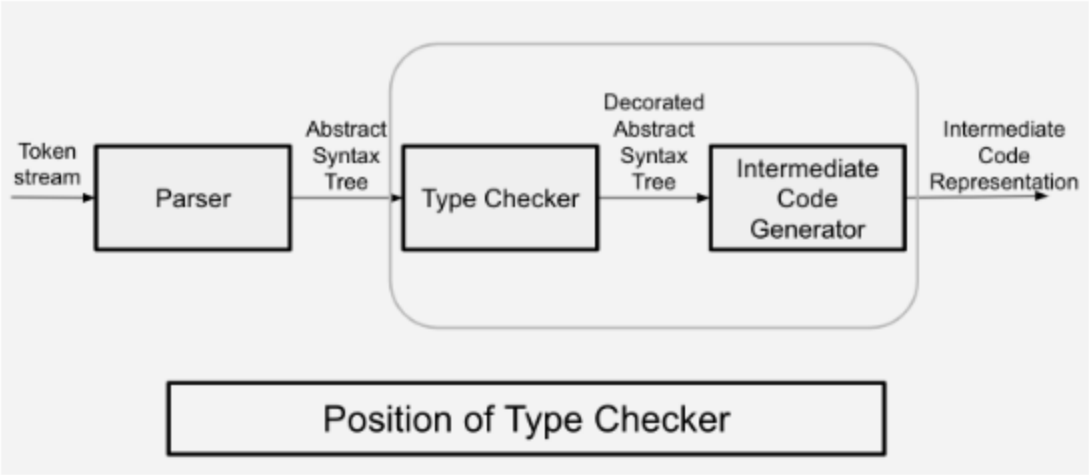
Production Rule (PR)	Semantic Action
D -> real, id	ENTER (id.PLACE, real) D.ATTR = real
D -> integer, id	ENTER (id.PLACE, integer) D.ATTR = integer
D -> D1, id	ENTER (id.PLACE, D1.ATTR) D.ATTR = D1.ATTR

Type checking:

Type checking is the process of verifying and enforcing constraints of types in values. A compiler must check that the source program should follow the syntactic and semantic conventions of the source language and it should also check the type rules of the language.

It checks the type of objects and reports a type error in the case of a violation, and incorrect types are corrected.

Conversion from one type to another type is known as **implicit** if it is to be done automatically by the compiler. Implicit type conversions are also called **Coercion** and coercion is limited in many languages.



Types of Type Checking:

There are two kinds of type checking:

1. Static Type Checking.
2. Dynamic Type Checking.

Static Type Checking:

Static type checking is defined as type checking performed at compile time. It checks the type variables at compile-time, which means the type of the variable is known at the compile time.

The Benefits of Static Type Checking:

1. Runtime Error Protection.
2. It catches syntactic errors like spurious words or extra punctuation.
3. It catches wrong names like Math and Predefined Naming.
4. Detects incorrect argument types.
5. It catches the wrong number of arguments

Dynamic Type Checking:

Dynamic Type Checking is defined as the type checking being done at run time. In Dynamic Type Checking, types are associated with values, not variables. Implementations of dynamically type-checked languages runtime objects are generally associated with each other through a type tag, which is a reference to a type containing its type information. Dynamic typing is more flexible.

Languages like Pascal and C have static type checking. Type checking is used to check the correctness of the program before its execution. The main purpose of type-checking is to check the correctness and data type assignments and type-casting of the data types, whether it is syntactically correct or not before their execution.

Static Type-Checking is also used to determine the amount of memory needed to store the variable.

Control flow

Control flow is an order in which statements are executed. The code executes in order from the first line in the file to the last line unless it comes across conditionals or loops that change the order.

Control flow statements include **conditional statements**, **branching statements**, and **looping statements**. Control flow can be decided with the help of a boolean expression.

1. Boolean Expressions
2. Short-Circuit Code
3. Flow-of-Control Statements

Boolean Expressions

Boolean expressions are composed of the boolean operators (which we denote &&, I I, and !, using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form $E_{\pm}$ rel E_2, where Ex and E_2 are arithmetic expressions. In this section, we consider boolean expressions generated by the following grammar:

$$B \rightarrow B \,||\, B \mid B \,\&\&\, B \mid !B \mid (B) \mid E \,\textbf{rel}\, E \mid \textbf{true} \mid \textbf{false}$$

We use the attribute rel. op to indicate which of the six comparison operators <, < = , =, ! =, >, or >= is represented by rel.

Short-Circuit Code

In short-circuit (or jumping) code, the boolean operators &&, I I, and ! trans-late into jumps. The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code se-quence.

Example 6.21 : The statement

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

might be translated into the code of Fig. **6.34.** In this translation, the boolean expression is true if control reaches label L_2. If the expression is false, control goes immediately to L_u skipping L_2 and the assignment x = **0.**

```
if x < 100 goto L2
   ifFalse x > 200 goto L1
   ifFalse x != y goto L1
L2: x = 0
L1:
```

Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of statements such as those generated by the following grammar:

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if } (\ B \ ) \ S_1 \\
S & \rightarrow & \textbf{if } (\ B \ ) \ S_1 \ \textbf{else} \ S_2 \\
S & \rightarrow & \textbf{while } (\ B \ ) \ S_1
\end{array}
$$

In these productions, nonterminal *B* represents a boolean expression and non-terminal S represents a statement.

Switch-Statements:

Switch and case statement is available in a variety of languages. The syntax of case statement is as follows: Syntax for switch case statement

switch E

begin

case V1: S1

case V2: S2

…

case Vn-1: Sn-1

default: Sn

end

When switch keyword is seen then a new temporary t and two new labels test and next are generated. When the case keyword occurs then for each case keyword, a new label Li is created and entered into the symbol table. The value of Vi of each case constant and a pointer to this symbol-table entry are placed on a stack.

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

Translation of a switch-statement

```
        code to evaluate E into t
        if t != V1 goto L1
        code for S1
        goto next
L1:     if t != V2 goto L2
        code for S2
        goto next
L2:
        ...
Ln-2:   if t != Vn-1 goto Ln-1
        code for Sn-1
        goto next
Ln-1:   code for Sn
next:
```

Another translation of a switch statement

Intermediate Code for Procedures:

Intermediate code is the link between high-level programming languages and machine code. It aids software portability and optimisation by providing a readily translated and optimised standard language.

Let there be a function f(a1, a2, a3, a4), a function f with four parameters a1,a2,a3,a4.

Three address code for the above procedure call(f(a1, a2, a3, a4)).

```
param a1
param a2
param a3
param a4
call  f, n
```

'call' is a calling function with f and n, here f represents name of the procedure and n represents number of parameters

Now, let's first take an example of a program to understand function definition and a function call.

```
main()
{
swap(x,y);  //calling function
}
void swap(int a, int b)   // called function
{
// set of statements
}
```

In the above program we have the main function and inside the main function a function call swap(x,y), where x and y are actual arguments. We also have a function definition for swap, swap(int a, int b) where parameters a and b are formal parameters.

In the three-address code, a function call is unraveled into the evaluation of parameters in preparation for a call, followed by the call itself.