UNIT – IV
Semantic Analysis and Intermediate Code Generation

Final Year BTECH

Subject : **System Software and Compiler Design**

Unit IV : Contents

Semantic Analysis:

- Need, Syntax Directed Translation
- Syntax Directed Definitions
- Translation of
  - Assignment Statements
  - Iterative statements,
  - Boolean expression
  - conditional statements,
- Type Checking and Type conversion

Intermediate Code Formats:

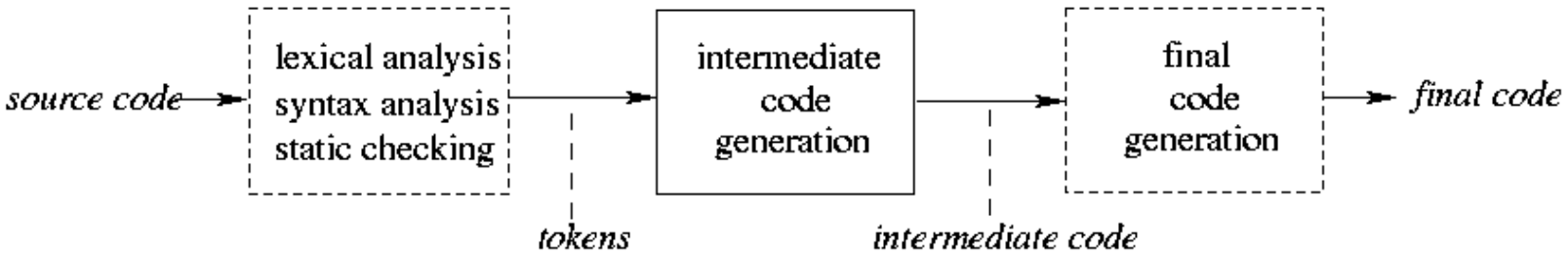
- Postfix notation, Parse and syntax tress,
- Three address code, quadruples and triples

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.

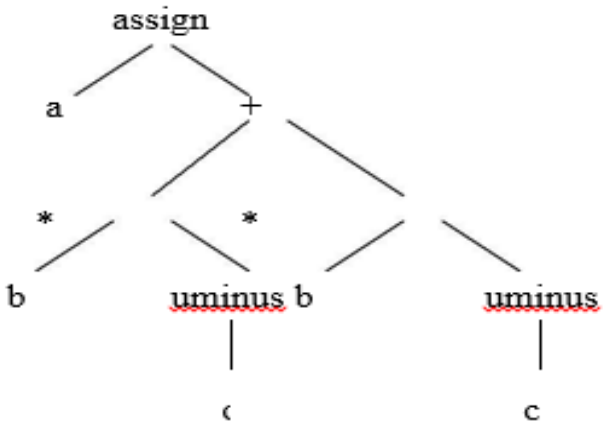


Intermediate Languages

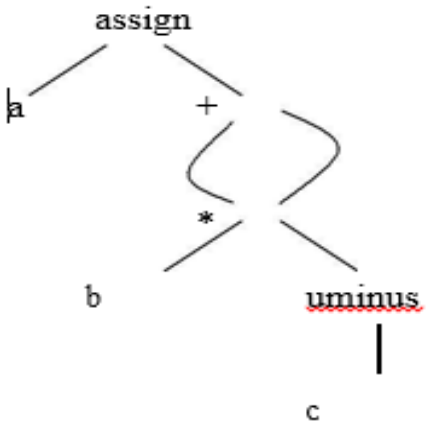
- Three ways of intermediate representation:
- Syntax tree
- Postfix notation
- Three address code
- The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

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

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 a sequence

  t1 : = y * z

  t2 : = x + t1

- where t1 and t2 are compiler-generated temporary names.

Advantages of Three-Address Code

- **Advantages of three-address code**:

1. The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for Target code generation and optimization.

2. 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 Example

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

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

(a) Code for the syntax tree

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

(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.

   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

2. Copy Statements 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.

Types of Three -Address Statements

5. 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(x1, x2, …. ,xn).
6. Indexed assignments of the form x : = y[i] and x[i] : = y.

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

Three -Address Statements: Example

- Consider the statement

  do { i = i+l;} while (a[i] < v) ;

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

(a) Symbolic labels.

```
100: t1 = i + 1
101: i = t1
102: t2 = i * 8
103: t3 = a [ t2 ]
104: if t3 < v goto 100
```

(b) Position numbers.

The multiplication i * 8 is appropriate for an array of elements that each take 8 units of space.

Implementation of Three-Address Statements

- Three address code instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called
  - ***Quadruples*** A quadruple (or just "quad') has four fields, which we call op, arg,, arg2, and result
  - ***Triples:*** A triple has only three fields, which we call op, arg1, and arg2. the DAG and triple representations of expressions are equivalent
  - ***Indirect Triples:*** consist of a listing of pointers to triples, rather than a listing of triples themselves.

- The benefit of **Quadruples** over **Triples** can be seen in an optimizing compiler, where instructions are often moved around**.**
- With ***quadruples***, if we move an instruction that computes a temporary ***t***, then the instructions that use ***t*** require no change. With ***triples***, the result of an operation is referred to by its position, so moving an instruction may require to change all references to that result. ***This problem does not occur with 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.

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

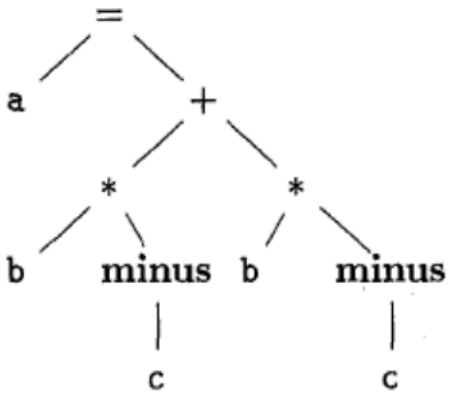
(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

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) Syntax tree

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

(b) Triples

Triples

- 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

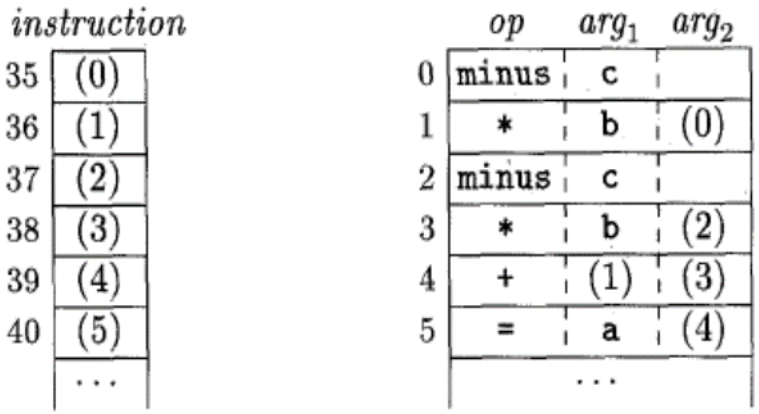
	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

(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:



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

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

NEED FOR SEMANTIC ANALYSIS

- Semantic analysis is a phase by a compiler that adds semantic information to the parse tree and performs certain checks based on this information.
- It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate/target) code is generated. (In a compiler implementation, it may be possible to fold different phases into one pass.)
- Typical examples of semantic information that is added and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding).
- Sometimes also some early code optimization is done in this phase. For this phase the compiler usually maintains symbol tables in which it stores what each symbol (variable names, function names, etc.) refers to.

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Syntax Directed Definitions

- Syntax Directed Definitions are a generalization of context-free grammars in which:
  1. Grammar symbols have an associated set of Attributes;
  2. Productions are associated with Semantic Rules for computing the values of attributes.
- Such formalism generates Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).
- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

ATTRIBUTE GRAMMAR

- Attributes are properties associated with grammar symbols. Attributes can be numbers, strings, memory locations, data types, etc.

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.

- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

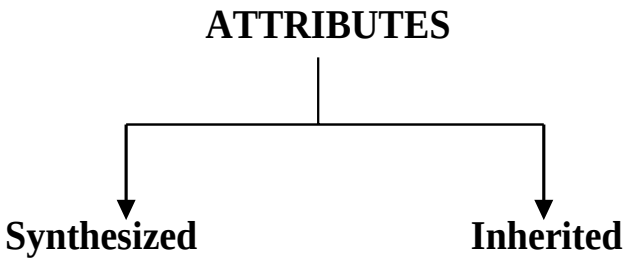
  Ex: $E \rightarrow E + T \{ E.value = E.value + T.value \}$

- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.

ATTRIBUTE TYPES

- Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes



Synthesized Attributes

- These are those attributes which get 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.
- To illustrate, assume the following production:

  S -> ABC

  S.a= A.a,B.a,C.a

- If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

- **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.

Synthesized Attributes: Example

- Consider the following grammar:

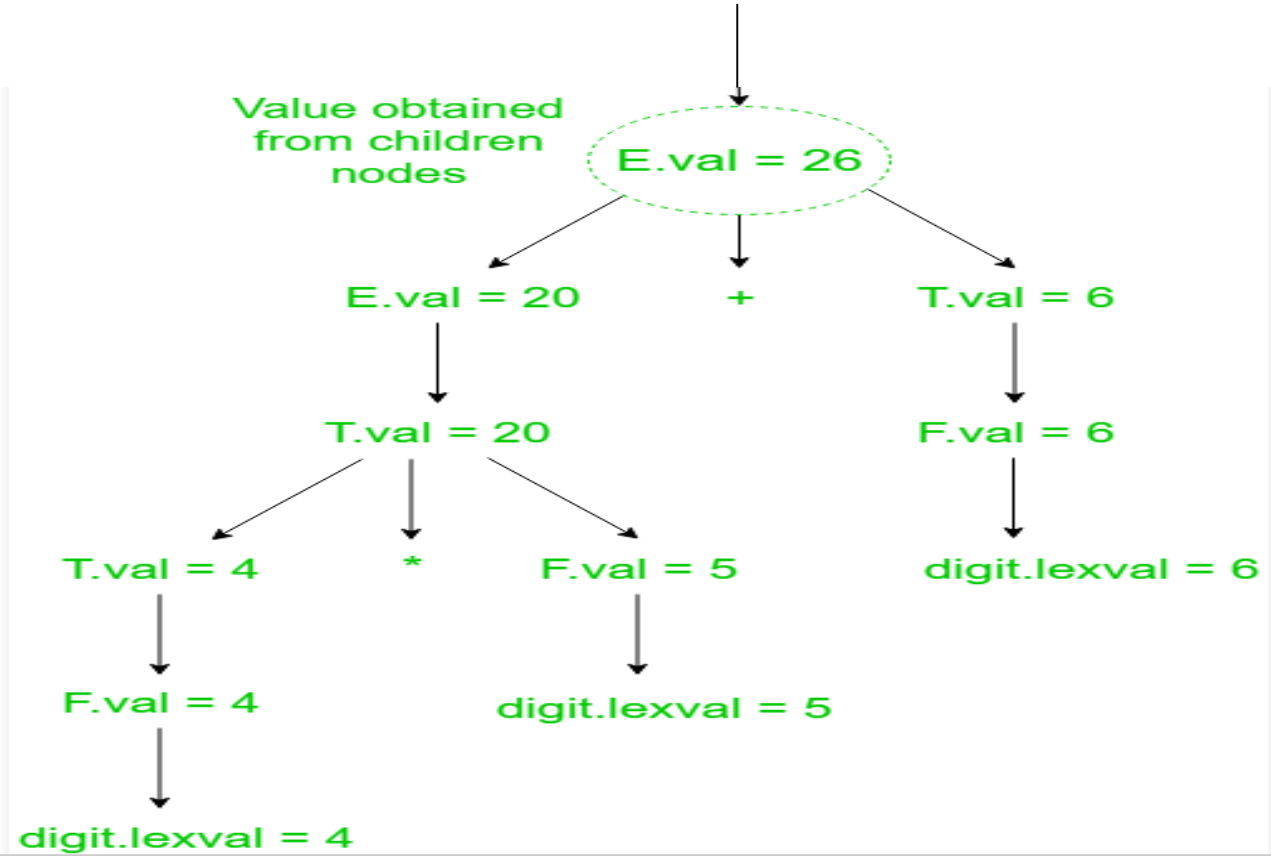
  S --> E
  E --> E1 + T
  E --> T
  T --> T1 * F
  T --> F
  F --> digit

- The SDD for the above grammar can be written as follow

PRODUCTIONS	SEMANTIC RULES
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

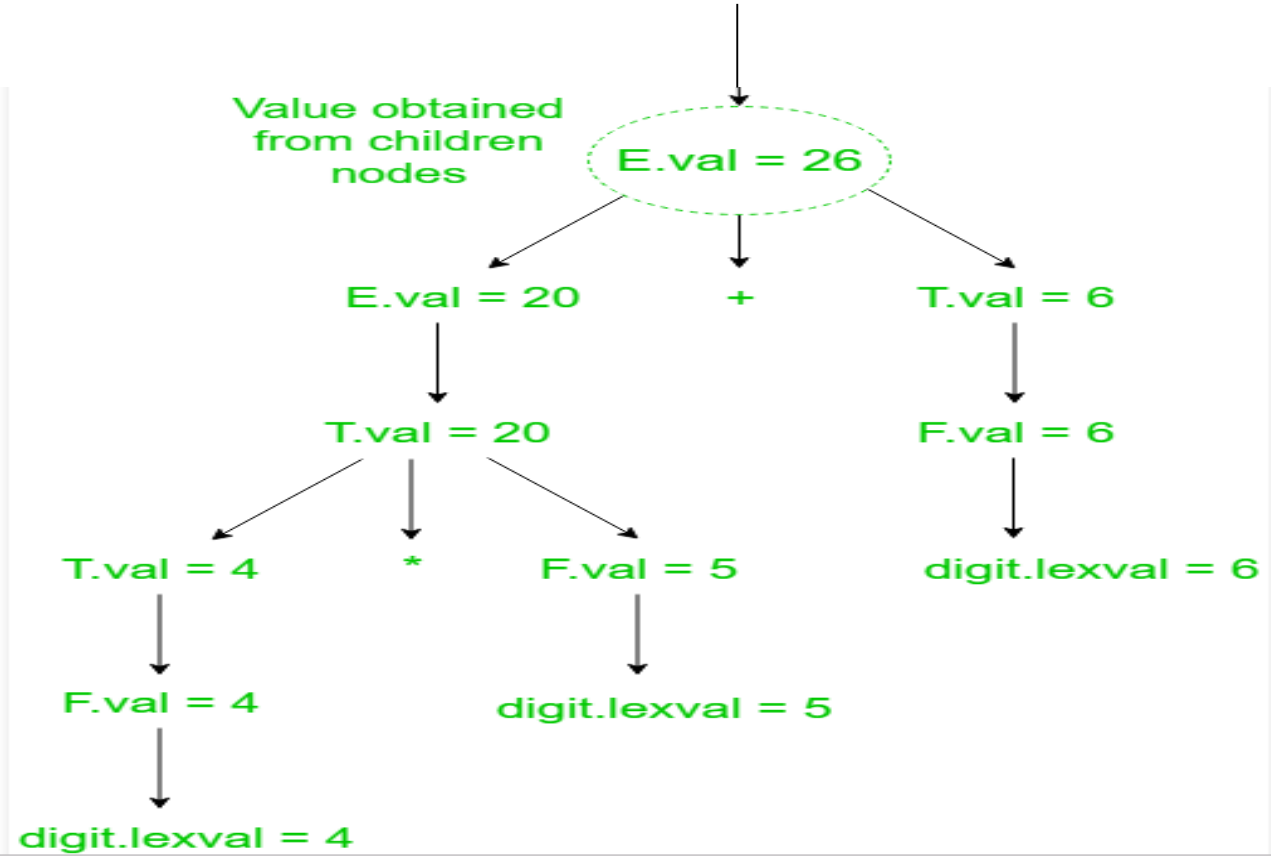
Synthesized Attributes: Example

- 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

- The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree



Inherited Attributes

- These are the attributes which inherit 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 }

- B can get values from A, C and D. C can take values from A, B, and D. Likewise, D can take values from A, B, and C.

- **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.

Inherited Attributes: Example

- Consider the following grammar:

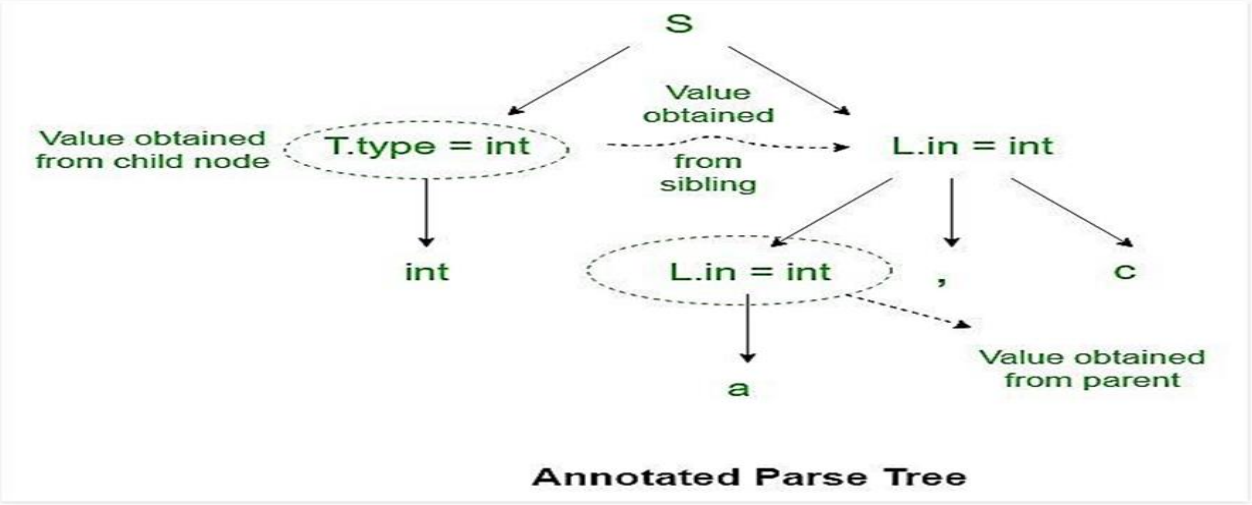
D --> T L
T --> int
T --> float
T --> double
L --> L1, id
L --> id

- The SDD for the above grammar can be written as follow

PRODUCTIONS	SEMANTIC RULES
D→TL	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)

Inherited Attributes: Example

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



  - 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.

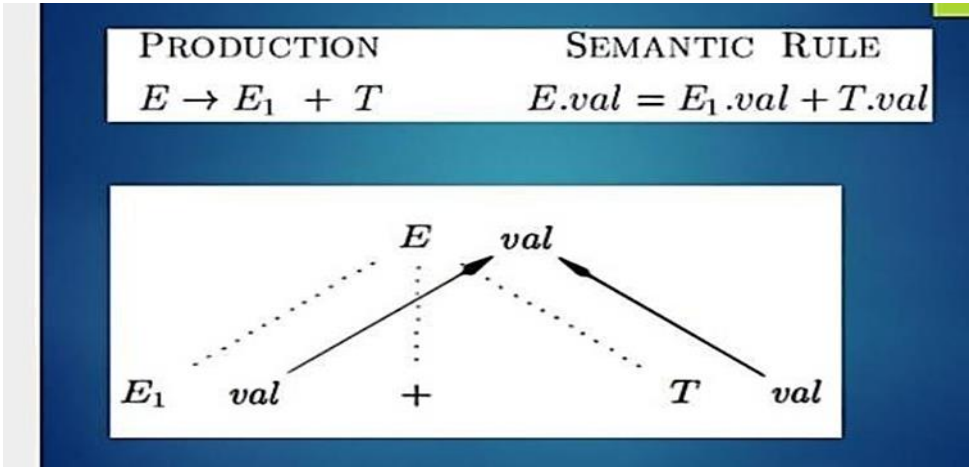
Implementing Syntax Directed Definitions

1. Dependency Graphs

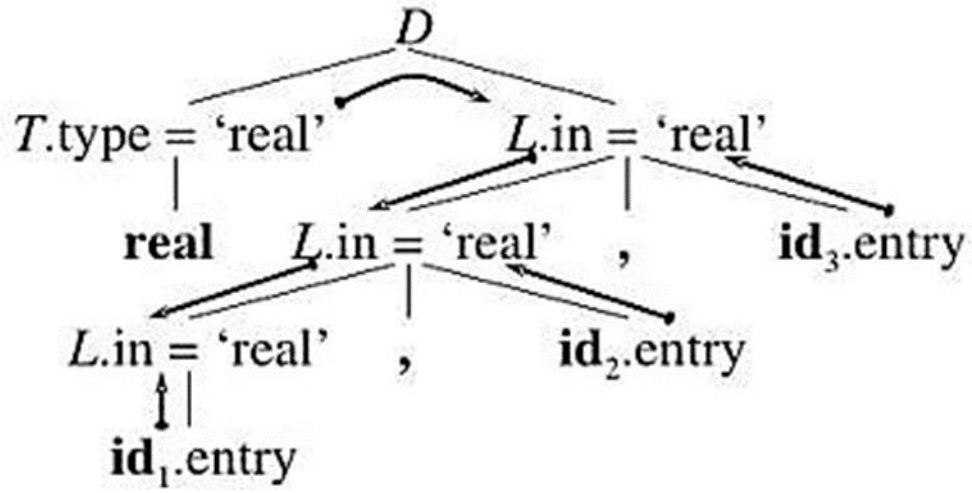
- Implementing a Syntax Directed Definition consists primarily in finding an order for the evaluation of attributes
- Each attribute value must be available when a computation is performed.
- Dependency Graphs are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.
- Annotated parse tree shows the values of attributes, dependency graph helps to determine how those values are computed

Implementing Syntax Directed Definitions

- The interdependencies among the attributes of the various nodes of a parse-tree can be depicted by a directed graph called a dependency graph.
- **There is a node for each attribute;**
- **If attribute b depends on an attribute c there is a link from the node for c to the node for b (b ← c).**
- Dependency Rule: If an attribute b depends from an attribute c, then we need to find the semantic rule for c first and then the semantic rule for b.



Implementing Syntax Directed Definitions



Dependency graph for declaration statements.

Evaluation order

- A dependency graph characterizes the possible order in which we can calculate the attributes at various nodes of the parse tree.
- If there is an edge from node M to N, then the attribute corresponding to M first be evaluated before evaluating N.
- Thus the allowable orders of evaluation are N1,N2…..,Nk such that if there is an edge from Ni toNj 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 is no topologicalsort.ie, there is no way to evaluate SDD on this parse tree.

TYPES OF SDT'S

1. S–attributed definition
2. L –attributed definition

S-attributed Definition

- S stands for synthesized
- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
  EXAMPLE:
  A→BC { A.a=B.a,C.a}
- 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.
  A→BCD{ }.

Note: (Also write SDD for desk calculator as example).

L –Attributed Definition

- L stands for one parse from left to right.

- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from parent and left siblings only, it is called as L-attributed SDT.

  EXAMPLE:

  A→BCD {B.a=A.a, C.a=B.a}

  C.a=D.a This is not possible

- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.
- A→{ }BC

  B{ }C

  BC{ }
- Note: (Also write SDD for declaration statement as example)
- If an attribute is S attributed, it is also L attributed

Type Checking

- A compiler must check that the program follows the **Type Rules** of the language.
- Information about **Data Types** is maintained and computed by the compiler.
- The **Type Checker** is a module of a compiler devoted to type checking tasks.
- **Examples of Tasks.**

  --The operator mod is defined only if the operands are integers;

  --Indexing is allowed only on an array and the index must be an integer;

  --A function must have a precise number of arguments and the parameters must have a correct type;

Type Checking

What is type?

- Type is a notion or a rule that varies from language to language.
- So type checking is done to ensure that whether the source program follows the syntactic and semantic rule of that language.

- Type checking can be of two types:
    1. Static checking (Done at compile time)
    2. Dynamic checking. (Done during run time)

Static Type Checking

- Static type checking is done at compile time.
- The information the type checker needs is obtained via declarations and stored in a master symbol table. After this information is collected, the types involved in each operation are checked.
- It is very difficult for a language that only does static type checking to meet the full definition of strongly typed.

For example, if a and b are of type int and we assign very large values to them, a * b may not be in the acceptable range of ints.

Dynamic Type Checking

- Dynamic type checking is implemented by including type information for each data location at run time
  - For example, a variable of type double would contain both the actual double value and some kind of tag indicating "double type"
  - The execution of any operation begins by first checking these type tags
  - The operation is performed only if everything checks out. Otherwise, a type error occurs and usually halts execution
    For example, when an add operation is invoked, it first examines the type tags of the two operands to ensure they are compatible.
  - Another example is array out of bound
- Normally done in languages that do not require prior data type declaration of variables at compile time – For example LISP, JavaScript etc.
- Dynamic type checking clearly comes with a run time performance penalty, but it usually much more difficult to subvert and can report errors that are not possible to detect at compile time.

Implicit Type Conversion

- Many compilers have built-in functionality for correcting the simplest of type errors.
- Implicit type conversion, or coercion, is when a compiler finds a type error and then changes the type of the variable to an appropriate type.
- This happens in C, for example, when an addition example, when an addition operation is performed on a mix of integer and floating point values.
  – The integer values are implicitly promoted before the addition is performed.
- Other languages are much stricter about type coercion. – Ada and Pascal, for example, provide almost no automatic coercions, requiring the programmer to take explicit actions to convert between various numeric types.
- The question of whether to provide a coercion capability or not is controversial.
- Coercions can free a programmer from worrying about details, but they can also hide serious errors that might otherwise have popped up during compilation.

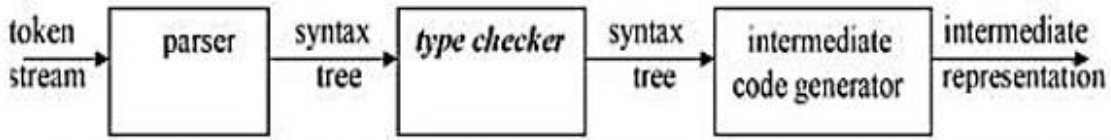
Types of Static Type Checking

- Four types of static type checks
  - Type checks
  - Flow of control checking
  - Uniqueness Checking
  - Name related checks

Type Checks

- To make it sure that the operator is applied to compatible operands
  - A Compiler must report an error if an operator is applied to incompatible operands
  - For example : An integer variable is added with a character variable

Position of type checker



Example

```
int sum;
float a, b;
Sum = a + b;
Product = a * b;
If a.type = int && b.type = int then
Sum.type = int
Else error
If a.type = float && b.type = float then
Sum.type = float
Else error
```

Flow of control checking

- Flow of control checking
  - – Checking of branching statements
    GOTO, BREAK, and CONTINUE etc.
    If branching location exists or not
  - – If it does not exist, then error is reported

Uniqueness Checking

- Make it sure unique declaration of variable in a scope
- No multiple declaration of the same variable in the same scope must exist
- No two variables with the same name in a scope

Name related checks

- If the same name appears both in the beginning and the ending of a construct
  - It is the duty of compiler to make it sure that the same name appears at both the places
    - Example

    Unary statement
    - Unit++
    - Unit = unit +1

Coercions

- In case of type mismatch

– Incompatible operands at the two sides of an operator
  - The compiler may perform implicit type conversion
  - Called Coercions
  - Coercion may occur

– Incompatibility in assignment statement

– Incompatible operands of an arithmetic operator

Lvalue Rvalue

Float = int