SEMANTIC ANALYSIS:

Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.

Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination. Semantic analysis is the front end's penultimate (last but one) phase and the compiler's last chance to weed out incorrect programs. We need to ensure the program is sound enough to carry on to code generation.

A large part of semantic analysis consists of tracking variable/function/type declarations and type checking. In many languages, identifiers have to be declared before they're used. As the compiler encounters a new declaration, it records the type information assigned to that identifier.

Then, as it continues examining the rest of the program, it verifies that the type of an identifier is respected in terms of the operations being performed. For example, the type of the right side expression of an assignment statement should match the type of the left side, and the left side needs to be a properly declared and assignable identifier.

The parameters of a function should match the arguments of a function call in both number and type. The language may require that identifiers be unique, thereby forbidding two global declarations from sharing the same name.

Arithmetic operands will need to be of numeric—perhaps even the exact same type (no automatic **int**-to-**double** conversion, for instance). These are examples of the things checked in the semantic analysis phase.

Some semantic analysis might be done right in the middle of parsing. As a particular construct is recognized, say an addition expression, the parser action could check the two operands and verify they are of numeric type and compatible for this operation. In fact, in a one pass compiler, the code is generated right then and there as well.

In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.

The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.

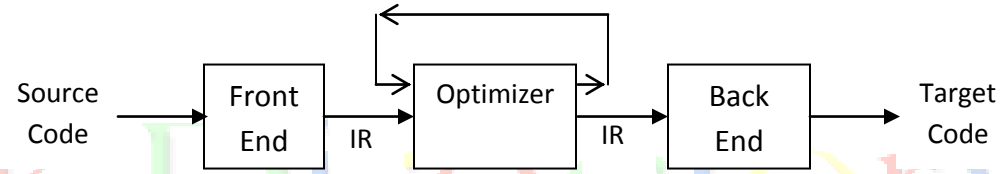
As for Lexical and Syntax analysis, also for Semantic Analysis we need both a Representation Formalism and an Implementation Mechanism.

INTERMEDIATE FORMS OF SOURCE PROGRAM:

An intermediate source form is an internal form of a program created by the compiler while translating the program from a high-level language to assembly-level or machine-level code. There are a number of advantages to using intermediate source forms.

An intermediate source form represents a more attractive form of target code than does assembly or machine code. An intermediate representation is a compile time Datastructure. An intermediate representation encodes the knowledge that the compiler has derived about the source program.

Certain optimization strategies can be more easily performed on intermediate source forms than on either the original program or the assembly-level or machine-level code. Compilers which produce a machine-independent intermediate source form are more portable than those which do not.



Front End:

It produces an **I**ntermediate **R**epresentation (IR) for the program.

Optimizer:

It transforms the code in IR form in to an equivalent program that may run more efficiently.

Back End:

It transforms the code in IR form in to native code for the target machine.

Most compilers translate the source program first to some form of *intermediate representation* and convert from there into machine code. The intermediate representation is a machine- and language-independent version of the original source code.

Although converting the code twice introduces another step, use of an intermediate representation provides advantages in increased abstraction, cleaner separation between the front and back ends, and adds possibilities for retargeting/ cross-compilation.

Intermediate representations also lend themselves to supporting advanced compiler optimizations and most optimization is done on this form of the code. There are many intermediate representations in use.

Intermediate representations are usually categorized according to where they fall between a high-level language and machine code. IRs that are close to a high-level language are called high-level IRs, and IRs that are close to assembly are called low-level IRs. For example, a high-level IR might preserve things like array subscripts or field accesses whereas a low-level IR converts those into explicit addresses and offsets.

For example, consider the following three code examples:

Original	*High IR*	*Mid IR*	*Low IR*

`float a[10][20];`
`a[i][j+2];`
	`t1 = a[i, j+2]`	`t1 = j + 2`
		`t2 = i * 20`
		`t3 = t1 + t2`
		`t4 = 4 * t3`
		`t5 = addr a`
		`t6 = t5 + t4`
		`t7 = *t6`
			`r1 = [fp - 4]`
			`r2 = [r1 + 2]`
			`r3 = [fp - 8]`
			`r4 = r3 * 20`
			`r5 = r4 + r2`
			`r6 = 4 * r5`
			`r7 = fp - 216`
			`f1 = [r7 + r6]`

High-level IRs usually preserves information such as loop-structure and if-then-else statements. They tend to reflect the source language they are compiling more than lower-level IRs.

Medium-level IRs often attempts to be independent of both the source language and the target machine.

Low-level IRs tends to reflect the target architecture very closely, and as such are often machine-dependent. They differ from actual assembly code in that there may be choices for generating a certain sequence of operations, and the IR stores this data in such a way as to make it clear that choice must be made.

Sometimes a compiler will start-out with a high-level IR, perform some optimizations, translate the result to a lower-level IR and optimize again, then translate to a still lower IR, and repeat the process until final code generation.

Some of the types of intermediate source forms are Polish notation, abstract syntax trees.

ABSTRACT SYNTAX TREES:

A parse tree is an example of a very high-level intermediate representation. You can usually completely reconstruct the actual source code from a parse tree since it contains all the information about the parsed program.

More likely, a tree representation used as an IR is not quite the literal parse tree (intermediate nodes may be collapsed, groupings units can be dispensed with, etc.), but it is

winnowed down to the structure sufficient to drive the semantic processing and code generation. Such a tree is usually referred to as an *abstract syntax tree*.

Each node represents a piece of the program structure and the node will have references to its children sub trees (or none if the node is a leaf) and possibly also have a reference to its parent.

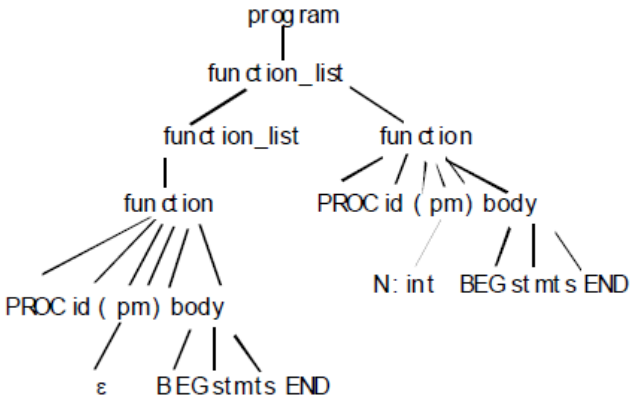
Consider the following excerpt of a programming language grammar:

Program –> *function_list*
function_list –> *function_list function | function*
function –> *PROCEDURE ident (params) body*
params –> …

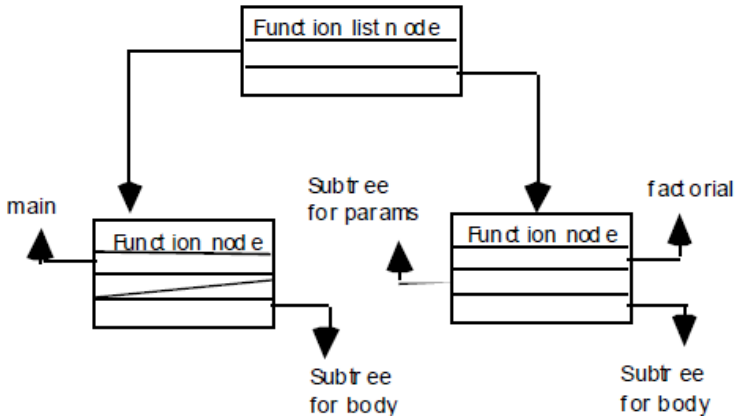
A sample program for this language is given below:

```
PROCEDURE main()
BEGIN
statement...
END
PROCEDURE factorial(n:INTEGER)
BEGIN
statement...
END
```


The literal parse tree (or concrete syntax tree) for the sample program looks something like:



Here is what the abstract syntax tree looks like (notice how some pieces like the parens and keywords are no longer needed in this representation):



The parser actions to construct the tree might look something like this:

function: PROCEDURE ident (params) body

{ $$ = MakeFunctionNode($2, $4, $6); }

function_list: function_list function

{ $$ = $1; $1->AppendNode($2); }

POLISH NOTATION:

The Polish mathematician JAN ŁUKASIEWICZ made the observation that, by writing all operators either before their operands or after them, it is not necessary to keep track of priorities, of parentheses bracketing expressions, or to make repeated scans to evaluate an expression.

Polish forms:

- When the operators are written before their operands, it is called the ***prefix form***.
- When the operators come after their operands, it is called the ***postfix form***, or, sometimes, ***reverse Polish form*** or ***suffix form***.
- The ***infix form*** means writing binary operators between their operands.

Polish notation, also known as Polish prefix notation or simply prefix notation is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity.

Polish notation is invented in order to simply the sentential logic. The term Polish notation is sometimes taken (as the opposite of infix notation) to also include Polish postfix notation, or Reverse Polish notation, in which the operator is placed after the operands.

When Polish notation is used as syntax for mathematical expressions by interpreters of programming languages, it is readily parsed into abstract syntax trees and can, in fact, define a one-to-one representation for the same.

Suffix or reverse Polish notation was one of the earliest intermediate source forms developed and remains among the most popular. It has been used in some FORTRAN compilers as an intermediate form for arithmetic expressions.

It has also been used in some interpreters, such as Hewlett Packard's interpreter for BASIC. We introduced suffix Polish notation as a method of specifying arithmetic expressions unambiguously without recourse to parentheses.

An algorithm to translate the Polish intermediate source form of an arithmetic expression into a simple machine language can easily be written.

When Polish notation is extended to include non-arithmetic language constructs, the more powerful notation which results is often called extended reverse Polish notation.

For example, operators can be added to handle the assignment operation, conditional branching, and subscripted variables. To illustrate this, let us examine how Polish notation can be extended to include conditional branching.

Consider an "if" statement with the following format:

If <expr> then <stmt1> else <stmt2>

This statement cannot be represented in Polish as

<expr> <stmt1> <stmt2> If

Because both (stmt1) and (stmt2) would be evaluated before being placed on the stack; since the purpose of the "if" statement is to perform only one of (stmt1) and (stmt2), a different representation must be adopted. Writing the statement as

<expr> <label1> BZ <stmt1> <label2> BR <stmt2>

THREE ADDRESS CODE:

Three-address code is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations. Each three address code instruction has at most three operands and is typically a combination of assignment and a binary operator.

For example, t1 := t2 + t3. The name derives from the use of three operands in these statements even though instructions with fewer operands may occur.

Since three-address code is used as an intermediate language within compilers, the operands will most likely not be concrete memory addresses or processor registers, but rather symbolic addresses that will be translated into actual addresses during register allocation. It is also not uncommon that operand names are numbered sequentially since three-address code is typically generated by the compiler.

In three-address code there is at most one operator on the right side of an instruction. Thus a source language expression like x + y * z might be translated into the sequence of three address instructions as given below:

t1 = y * z

t2 = x + t1

Where t1 and t2 are compiler generated temporary names. The three address code is a linearized representation of a syntax tree or a DAG in which the names correspond to the interior nodes of the graph.

Three-Address code is built from two concepts: address and instructions. An address can be one of the following 1) name 2) constant and 3) compiler-generated temporary.

Name: we allow source program names to be appeared as the addresses in the three address code.

Constant: a compiler must deal with many different types of constants and variables.

Compiler-generated temporary is useful in optimizing compilers, to create a distinct name each time when a temporary is needed.

A refinement of three-address code is static single assignment form (SSA). Three Address Codes are a form of IR similar to assembler for an imaginary machine. Each three address code instruction has the form *x := y op z*

- x, y, z are names (identifiers), constants, or temporaries (names generated by the compiler)
- op is an operator, from a limited set defined by the IR.

Observe that

- Complicated arithmetic expressions are represented as a sequence of 3-address statements, using temporaries for intermediate values. For instance the expression x + y + z becomes:

  t1 := y + z

  t2 := x + t1

- Three address code is a linearized representation of a syntax tree, where the names of the temporaries correspond to the nodes.
- The use of names for intermediate values allows three-address code to be easily rearranged which is convenient for optimization. Postfix notation does not have this feature.
- 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.

Three-address code is a common intermediate representation generated by the front end of a compiler. It consists of instructions with a variety of simple forms:

- Assignment instructions of the form x = y op z, x = op y, or x = y where x, y, and z are names or compiler-generated temporaries. y and z can also be constants.
- Jump instructions of the form goto L, if x goto L, ifFalse x goto L, or if x relop y goto L where L is the label of some three-address instruction and relop is a relational operator such as <, <=, ==, and so on.
- Parameter passing, procedure calling, and procedure returning instructions of the form param x; call p, n; and return y. Here p is the name of the procedure and n is the number of parameters it takes.
- Indexed copy instructions of the form x = y[i] and x[i] = y.
- Address and pointer assignments of the form x = &y x = *y, and *x = y.

Three address code instructions can not specify the representation of the instructions in a data structure rather it specifies only the components of each type of instruction.

In a compiler three address code instructions are implemented as objects or as records. Three such representations are called “quadruples” “triples” and “indirect triples”.

- A **quadruple** has four fields *op, arg1, arg2, result.* For example the three address instruction x = y + z is represented by placing + in op, y and z in arg1, arg2 respectively and x in result.
- Let us consider the assignment statement a = b * -c + b * -c ;
- The three address code for above assignment is given below:

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

- Where the special operator minus is used to distinguish the unary minus (as in –c) and binary minus (as in b –c).
- The three address statement for the unary minus has only two addresses like the statement a = t5.
- The three address implementation for the above quadruples is given below:

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

- One of the exceptions for quadruple is, instructions with unary operators like x = minus y or x = y do not use arg2.
- Quadruple representation is beneficial for code optimization because with a quadruple representation one can quickly access the value of temporary variable.
- Quadruples use a name, sometimes called a temporary name or "temp", to represent the single operation.
- **Triples** are a form of three-address code which do not use an extra temporary variable; when a reference to another triple's value is needed, a pointer to that triple is used.
- A Triple has only three fields; they are op, arg1, arg2.
- Using triple we refer to the result of an operation x op y by its position, rather than by an explicit temporary name, because in quadruples we use a temporary name called as result.

- Thus instead of the temporary t1 in quadruples, triple representation would refer to the position (0). And this parenthesized number i.e. (0) represents a pointer in to the triple structure itself.
- 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, as in figure below:

TRIPLES			
	op	**arg1**	**arg2**
0	minus	C	
1	*	B	(0)
2	minus	C	
3	*	B	(2)
4	+	(0)	(3)
5	=	A	(4)

- Triples are difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. A variant of triples called indirect triples is easier to optimize.
- **Indirect Triples:** Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples.
- For example let us use an array instruction to list pointers to triples in the desired order. Such that the indirect triple representation of three address code is given as follows:

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

- With indirect triples an optimizing compiler moves an instruction by reordering the instruction list without affecting the triples.

ATTRIBUTED GRAMMAR:

An attribute grammar is a formal way to define attributes for the productions of a formal grammar, associating these attributes to values. The evaluation occurs in the nodes of the abstract syntax tree, when the language is processed by some parser or compiler.

It is defined as augmenting the conventional grammar with information to control semantic analysis and translation.

The attributes are divided into two groups: synthesized attributes and inherited attributes. The synthesized attributes are the result of the attribute evaluation rules, and may also use the values of the inherited attributes. The inherited attributes are passed down from parent nodes.

In some approaches, synthesized attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down and across it. For instance, when constructing a language translation tool, such as a compiler, it may be used to assign semantic values to syntax constructions. Also, it is possible to validate semantic checks associated with a grammar, representing the rules of a language not explicitly imparted by the syntax definition.

Attribute grammars can also be used to translate the syntax tree directly into code for some specific machine, or into some intermediate language. One strength of attribute grammars is that they can transport information from anywhere in the abstract syntax tree to anywhere else, in a controlled and formal way.

Attributed grammars are classified into two types: 1) S-attributed Grammar 2) L-attributed Grammar where attributes are divided in to two types 1) *synthesized attributes* and 2) *inherited attributes*.

S-attributed Grammar:

S-Attributed Grammars are a class of attribute grammars characterized by having no inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis of the parsing process, is a problem for bottom-up parsing because in bottom-up parsing, the parent nodes of the abstract syntax tree are created after creation of all of their children.

Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing. Yacc is based on the S-attributed approach.

L-attributed grammar:

L-attributed grammars are a special type of attribute grammars. They allow the attributes to be evaluated in one left-to-right traversal of the abstract syntax tree. As a result, attribute evaluation in L-attributed grammars can be incorporated conveniently in top-down parsing. Many programming languages are L-attributed. Special types of compilers, the narrow compilers, are based on some form of L-attributed grammar. These are comparable with S-attributed grammars. Used for code synthesis. Any S-attributed grammar is also an L-attributed grammar.

Synthesized Attributes:

A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. *A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.* Synthesized attributes are evaluated in bottom up fashion. Let us consider the following Context-free grammar which can describe a language made up of multiplication and addition of integers:

Expr → Expr + Term
Expr → Term
*Term → Term * Factor*
Term → Factor
Factor → "(" Expr ")"
Factor → integer

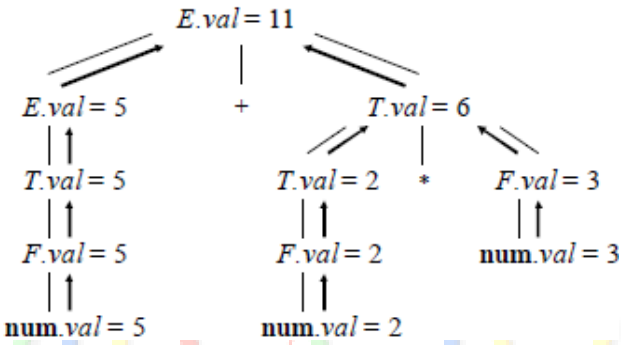
The following attribute grammar can be used to calculate the result of an expression written in the grammar which only uses synthesized values, and is therefore an S-attributed grammar.

Expr1 → Expr2 + Term [Expr1.value = Expr2.value + Term.value]
Expr → Term [Expr.value = Term.value]
*Term1 → Term2 * Factor [Term1.value = Term2.value * Factor.value]*
Term → Factor [Term.value = Factor.value]
Factor → "(" Expr ")" [Factor.value = Expr.value]
Factor → integer [Factor.value = strToInt (integer.str)]

Example:

Production	**Semantic Rules**
E→E+T	E.val = E1.val + T.val
E→T	E.val = T.val
T→T*F	T.val = T1.val + F.val
T→F	T.val = F.val
F→(E)	F.value = E.val
F→num	F.val = num.val

A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree. The annotated parse tree for 5+2*3 to the above grammar is:



Inherited Attributes:

An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. *An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.* Inherited attributes are evaluated in top down fashion.

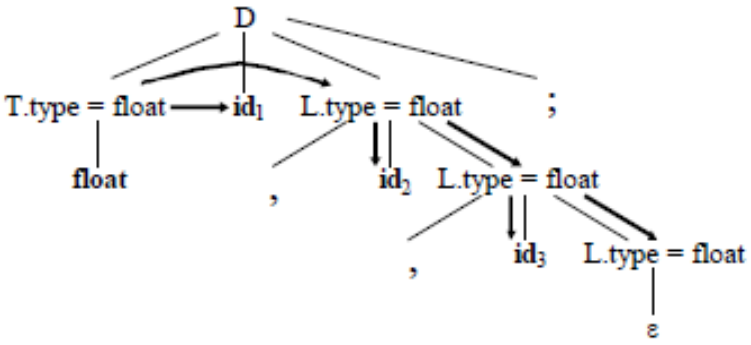
An inherited attribute at node N cannot be defined in terms of attribute values at the children of node N. However, a synthesized attribute at node N can be defined in terms of inherited attribute values at node N itself.

An inherited attribute at a node in parse tree is defined using the attribute values at the parent or siblings. Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears. For example, we can use an inherited attribute to keep track of whether an identifier appears on the left or the right side of an assignment in order to decide whether the address or the value of the identifier is needed.

Example:

Production	Semantic Rules
$D \rightarrow T$ **id** L ;	*enter*(**id**.*name*, *T.type*)
	L.type := *T.type*
$T \rightarrow$ **int**	*T.type* := INT_TYPE
$T \rightarrow$ **float**	*T.type* := FLOAT_TYPE
$L \rightarrow$, **id** L^2	enter(**id**.*name*, L.*type*)
	L^2.*type* := *L.type*
$L \rightarrow \varepsilon$	

The annotated parse tree for “float id, id, id; to the above grammar is:



SYNTAX DIRECTED TRANSLATION:

Syntax-directed translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language. The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

A class of syntax-directed translations called "L-attributed translations" (L for left-to-right) includes almost all translations that can be performed during parsing. Similarly, "S-attributed translations" (S for synthesized) can be performed easily in connection with a bottom-up parse. There are two ways to represent the semantic rules associated with grammar symbols.

- Syntax-Directed Definitions (SDD)

- Syntax-Directed Translation Schemes (SDT)

Syntax-Directed Definitions:

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register, strings. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. The attributes are evaluated by the semantic rules attached to the productions.

It is a high level specification in which implementation details are hidden, e.g., $$ = $1 + $2; /* does not give any implementation details. It just tells us. This kind of attribute equation we will be using, Details like at what point of time is it evaluated and in what manner are hidden from the programmer.*/

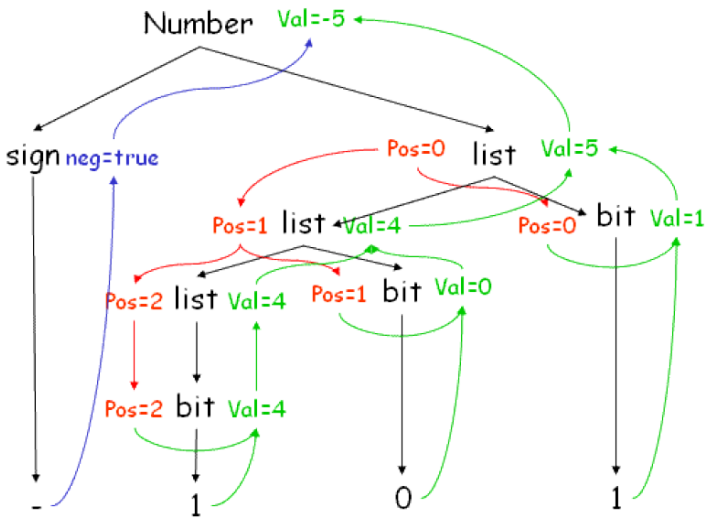
Syntax-Directed Translation Schemes (SDT):

SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed.

Sometimes we want to control the way the attributes are evaluated, the order and place where they are evaluated. This is of a slightly lower level. To avoid repeated traversal of the parse tree, actions are taken simultaneously when a token is found. So calculation of attributes goes along with the construction of the parse tree.

Along with the evaluation of the semantic rules the compiler may simultaneously generate code, save the information in the symbol table, and/or issue error messages etc. at the same time while building the parse tree. This saves multiple passes of the parse tree.

Parse tree and the dependence graph:



Dependence graph shows the dependence of attributes on other attributes, along with the syntax tree. Top down traversal is followed by a bottom up traversal to resolve the dependencies. Number, val and neg are synthesized attributes. Pos is an inherited attribute.

Example: In the rule E → E1 + T {print '+'}, the action is positioned after the body of the production. SDTs are more efficient than SDDs as they indicate the order of evaluation of semantic actions associated with a production rule. This also gives some information about implementation details.

CONVERSION OF POPULAR PROGRAMMING LANGUAGE CONSTRUCTS INTO INTERMEDIATE CODE:

An intermediate source form is an internal form of a program created by the compiler while translating the program from high level language in to assembly level or machine level code.

We will see how to write an intermediate code for various programming language constructs like assignment statements, Boolean expressions, case statements, arrays etc…

Declarations:

In the declarative statements the data items along with either data types are declared.

E.g.

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	{	T.type := pointer(T.type) T.width := 4}

TYPE CHECKING:

Type checking is defined as the process of verifying and enforcing the constraints of types. To do type checking a compiler needs to assign a type expression to each component of the source program. The compiler must then determine that these type expressions conform to a collection of logical rules that is called the type system for the source language.

Type checking has the potential for catching errors in programs. In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element. 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. An implementation of a language is strongly typed if a compiler guarantees that the programs it accepts will run without type errors.

Rules for Type Checking:

Type checking can take on two forms: synthesis and inference. Type synthesis builds up the type of an expression from the types of its sub expressions. It requires names to be declared before they are used. The type of $E_1 + E_2$ is defined in terms of the types of E_1 and E_2. A typical rule for type synthesis has the form

if f has type s -+ t and x has type s,
then expression f (x) has type t

Here, f and x denote expressions, and s -+ t denotes a function from s to *t.* This rule for functions with one argument carries over to functions with several arguments. The above rule can be adapted for $E_1 + E_2$ by viewing it as a function application add(E_1, E_2).

Type inference determines the type of a language construct from the way it is used. It is an important aspect of semantic analysis. A type is a set of values. Certain operations are valid for values of each type. For example, consider the type integer in Pascal. The operation mod can only be applied to values of type integer, and to no other types.

A language's type specifies which operations are valid for values of each type. A type checker verifies that the type of a construct matches which is expected by its context, and ensures that correct operations are applied to the values of each type.

Languages can be classified into three main categories depending upon the type system they employ. These are:

- **Untyped:** In these languages, there are no explicit types. Assembly languages fall into the category of these languages.
- **Statically typed:** In these type of languages, all the type checking is done at the compile time only. Because of this, these languages are also called strongly typed languages. Example of languages in this category is Algol class of languages.
- **Dynamically typed:** In dynamically typed languages, the type checking is done at the runtime. Usually, functional programming languages like Lisp, Scheme etc. have dynamic type checking.

In this course, our main focus will be on statically typed languages, particularly imperative programming languages. There are several advantages of static typing:

- Most of the error in the program is caught at the compile time only.

Most of the programming is done generally in statically typed languages because of above mentioned advantages. However, there are also certain drawbacks of static typing:

- It can be restrictive in certain situations. For example, in Pascal, all the identifiers are identified by a unique name. This can give rise to confusion in many cases.

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 may be either static or dynamic: The one done at compile time is static. In languages like Pascal and C type checking is primarily static and is used to check the correctness of a program before its execution.

Static type checking is also useful to determine the amount of memory needed to store variables. The design of a Type Checker depends on the syntactic structure of language constructs, the Type Expressions of the language, and the rules for assigning types to constructs.

A Type Expression denotes the type of a language construct. A type expression is either a Basic Type or is built applying Type Constructors to other types.

- A Basic Type is a type expression (int, real, Boolean, char). The basic type void represents the empty set and allows statements to be checked.
- Type expressions can be associated with a name: Type Names are type expressions.
- A Type Constructor applied to type expressions is a type expression. Type constructors include: Array, record, pointer, function.

Type System is a collection of rules for assigning type expressions to the various part of a program. Type Systems are specified using syntax directed definitions. A type checker implements a type system.

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