Semantic Analysis

The semantic analysis phase of compiler connects variable definition to their uses ,and checks that each expression has a correct type.

This checking called "**static type checking**" to distinguish it from "**dynamic type checking**" during execution of target program. This phase is characterized be the maintenance of symbol tables mapping identifiers to their types and locations.

Examples of static type checking:-

1. **Type checks :** A compiler should report an error if an operator is applied to an incompatible operand.
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. For example, a "*break*" statement in 'C' language causes control to leave the smallest enclosing *while* , *for* , or *switch* statement ;an error occurs if such an enclosing statement does not exist.
3. **Uniqueness checks:-** There are situations in which an object must be defined exactly once. For example, in 'Pascal' language, an identifier must be declared uniquely.
4. **Name-related checks:-** Sometimes, the same name must appear two or more times. For example, in **'Ada'** language a loop or block may have a name that appear at the beginning and end of the construct. The compiler must check that the same name is used at both places.

Type system:-

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

The following excerpts are examples of information that a compiler writer might have to start with.

- If both operands of the arithmetic operators "*addition*", "*subtraction*", and "*multiplication*" are of type *integer* , then the result is of type *integer*.

- The result of Unary **&** operator is a pointer to the object referred to by the operand. If the type of operand is *T* , the type of result is ' pointer to *T* '.

We can classify type into :

1. **Basic type:** This type are the atomic types with no internal structure , such as *Boolean, Integer, Real, Char, Subrange, Enumerated,* and a special basic types " *type-error , void* ".
2. **Construct types:** Many programming languages allows a programmer to construct types from *basic types* and other *constructed types.* For example *array, struct, set.*
3. **Complex type:** Such as *link list, tree, pointer.*

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*.

Specification of a simple type checker:-

The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. In this section, we specify a type checker for simple language in which the type of each identifier must be declared before the identifier is used.

Suppose the following grammar to generates program, represented by *nonterminal* P, consisting of a sequence of declarations D followed by a single expression E.

P ⟶ D ; E

D ⟶ D ; D | id : T

T ⟶ char | int | array[num] of T | ↑T

E ⟶ literal | num | id | E mod E | E[E] | E↑

Type checker (translation scheme) produce the following part that saves the type of an identifier:

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

- **The type checking of expression:** the following some of semantic rules:

E ⟶	literal	{E.type=char}	//constants represented
E ⟶	num	{E.type=int}	// = =

We can use a function *lookup*(e) to fetch the type saved in *ST* ,if identifier " e " appears in an expression:

E ⟶ id {E.type=lookup(id.entry)}

The following expression formed by applying (mod) to two subexpression:

E ⟶ E1 mod E2 {E.type= if E1.type=int and E2.type=int then int
Else type-error }

An array reference:

E ⟶ E1[E2] { E.type= if E2.type=int and E1.type=array[s,t] then t
Else type-error}

E ⟶ E1↑ { E.type= if E1.type=pointer(t) then t
Else type-error}

- **The type checking of statements :**

S ⟶ id=E {S.type=if id.type = E.type then void
Else type-error)}

S ⟶ if E then S1 {S.type= if E.type=boolean then S1.type
Else type-error }

S ⟶ while E do S1 {S.type= if E.type=boolean then S1.type
Else type-error }

S ⟶ S1 ; S2 { S.type= if S1.type=void and S2.type=void then void
Else type-error}

Intermediate Code Generation (IR)

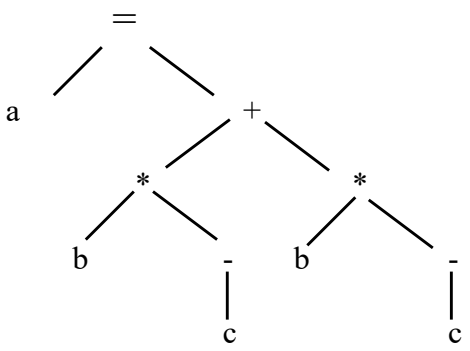
IR is an internal form of a program created by the compiler while translating the program from a *H.L.L* to *L.L.L.*(*assembly* or *machine code*),from IR the back end of compiler generates *target code*.

Although a source program can be translated directly into the target language,some benefits of using a machine independent IR are:

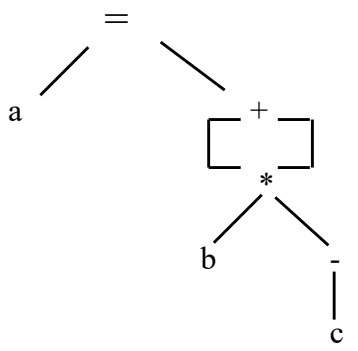
1. A compiler for different machine can be created by attaching a back end for a new machine into an existing front end.
2. Certain optimization strategies can be more easily performed on IR than on either original program or L.L.L.
3. An IR represents a more attractive form of target code.

Intermediate Languages:-

1. Syntax Tree and Postfix Notation are tow kinds of intermediate representations, for example **a=b*-c+b*-c**

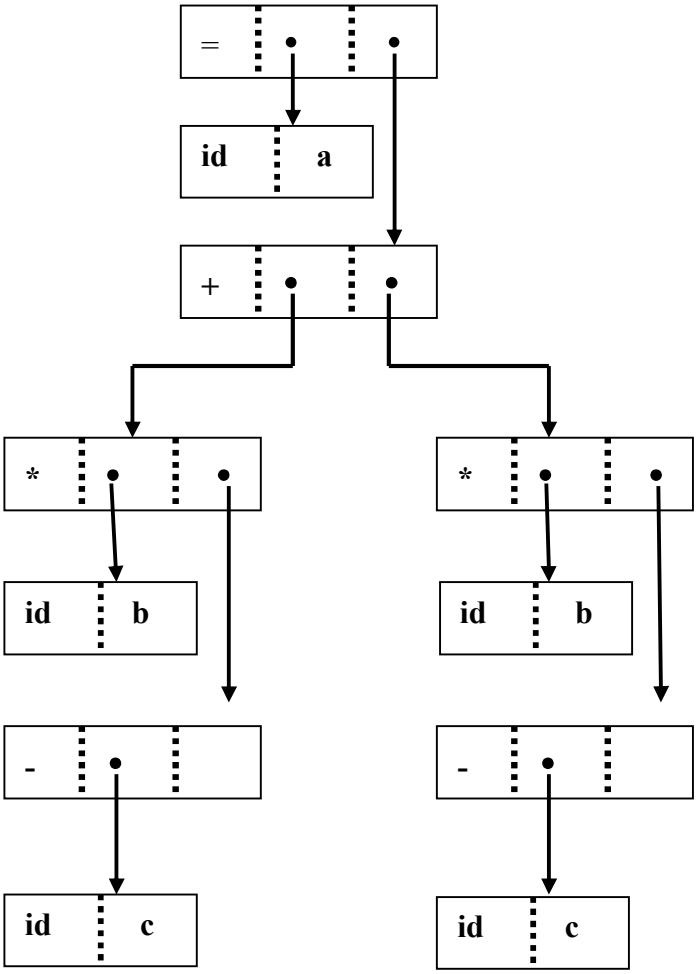


Syntax Tree



DAG

- A *DAG* give the same information in syntax tree but in compact way because common subexpressions are identified.
- *Postfix notation* is a linearized representation of a syntax tree, for example: **a b c - * b c - * + =**
- Two representation of above syntax tree are:



0	id	b	
1	id	c	
2	-	1	
3	*	0	2
4	id	b	
5	id	c	
6	-	5	
7	*	4	6
8	+	3	7
9	id	a	
10	=	9	8
	….	…..	…..
	…..	…..	…..

1 **2**

2. Three-Address Code is a sequence of statements of the general form :

$X = Y \; op \; Z$ // op is binary arithmetic operation

For example : $x + y * z$

$t1 = y * z$
$t2 = x + t1$

where t1 ,t2 are compiler generated temporary.