SEMANTIC ANALYSIS

Semantic analysis checks the source program for semantic consistency with the language definition i.e. parsers cannot handle context-sensitive features of programming languages

The following are static semantics of programming languages that can be checked by the semantic analyzer

- A variable cannot be used without having been defined
- The same variable/function/class/method cannot be defined twice
- Types match on both sides of assignments
- Parameter types and number match in declaration and use

Compilers can only generate code to check dynamic semantics of programming languages at runtime

- whether an overflow will occur during an arithmetic operation
- whether array limits will be crossed during execution
- whether recursion will cross stack limits
- whether heap memory will be insufficient

Generally, the plain parse-tree constructed during the syntax analysis is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree, for instance, `E → E + T` has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Generally, semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

E.g. `int a = "string value";`

This statement should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis.

Syntax Directed Translation

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar. The following are concepts related to syntax-directed translation:

i). An attribute is any quantity associated with a programming construct. Examples of attributes are data types of expressions, the number of instructions in the generated code, or the location of the first instruction in the generated code for a construct, among many other possibilities.

ii). A translation scheme is a notation for attaching program fragments to the productions of a grammar. The program fragments are executed when the production is used during syntax analysis. The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) in order to provide context-sensitive information. It is generally 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. The idea is to associate attributes with each node in the (abstract) syntax tree. Examples of attributes:

- Type information
- Storage location
- Assignable (e.g., expression vs variable)
- Value (for constant expressions)

It uses the notation X.a, where a is an attribute of node X

Attributes are associated with non-terminals and terminals and then, rules are attached to the productions of the grammar; these rules describe how the attributes are computed at those nodes of the parse tree where the production in question is used to relate a node to its children.

A syntax-directed definition associates

1. With each grammar symbol, a set of attributes, and
2. With each production, a set of semantic rules for computing the values of the attributes associated with the symbols appearing in the production.

A parse tree showing the attribute values at each node is called an annotated parse tree

Example

Consider the following grammar

$$
\begin{aligned}
L &\rightarrow E \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow F \\
F &\rightarrow (E) \\
F &\rightarrow \textbf{num}
\end{aligned}
$$

The following are attributes and rules that calculate the value of an expression

Production	Semantic rules
$L \rightarrow E$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **num**	$F.val =$ **num**$.lexval$

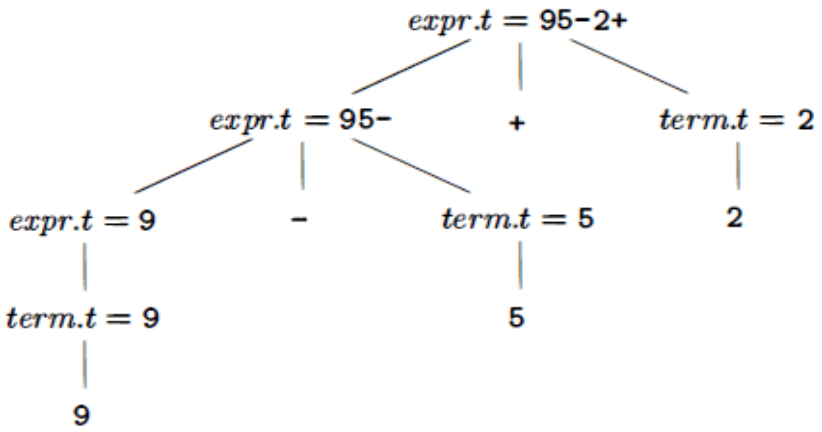
Notice that subscripts allow to distinguish different instances of the same symbol in a rule

Translation of Code

Syntax-directed definitions can be used to translate code, for instance, translating expressions to post-fix notation

Example

The figure below shows an annotated parse tree for 9-5+2 with an attribute t associated with the non-terminals expr and term. The value 95-2+ of the attribute at the root is the postfix notation for 9-5+2.



The above annotated parse tree is based on the syntax directed definition below for translating expressions consisting of digits separated by plus or minus signs into postfix notation. Each nonterminal has a string-valued

attribute t that represents the postfix notation for the expression generated by that nonterminal in a parse tree. The symbol | | in the semantic rule is the operator for string concatenation.

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

The postfix form of a digit is the digit itself; e.g. the semantic rule associated with the production term → 9 defines term.t to be 9 itself whenever this production is used at a node in a parse tree. The other digits are translated in a similar manner. When the production expr → term is applied, the value of term.t becomes the value of expr.t

Types of Attributes

Based on the way the attributes get their values, they can be broadly divided into two categories namely:

i). Synthesized attributes
ii). Inherited attributes

i). Synthesized Attributes

These attributes get values from the attribute values of their child nodes i.e. attribute value for the LHS nonterminal is computed from the attribute values of the symbols at the RHS of the rule.

E.g: Consider the following production:

S → ABC

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.

Synthesized attributes never take values from their parent nodes or any sibling nodes. Terminals can have synthesized attributes, computed by the lexer (e.g., id.lexeme), but no inherited attributes.

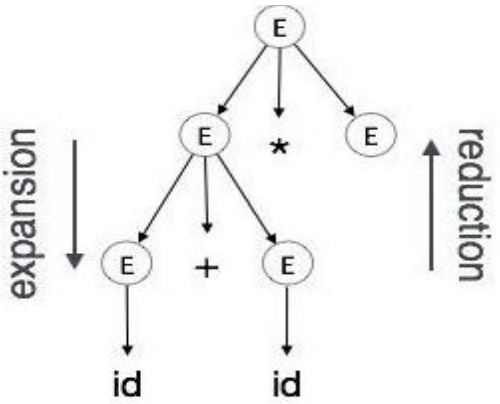
ii). Inherited Attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings i.e. attribute value of a RHS nonterminal is computed from the attribute values of the LHS nonterminal and some other RHS non-terminals.

E.g. S → ABC

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

Expansion: When a non-terminal is expanded to terminals as per a grammatical rule.



Reduction: When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Generally, an attribute grammar is a way of relating strings with "meanings"

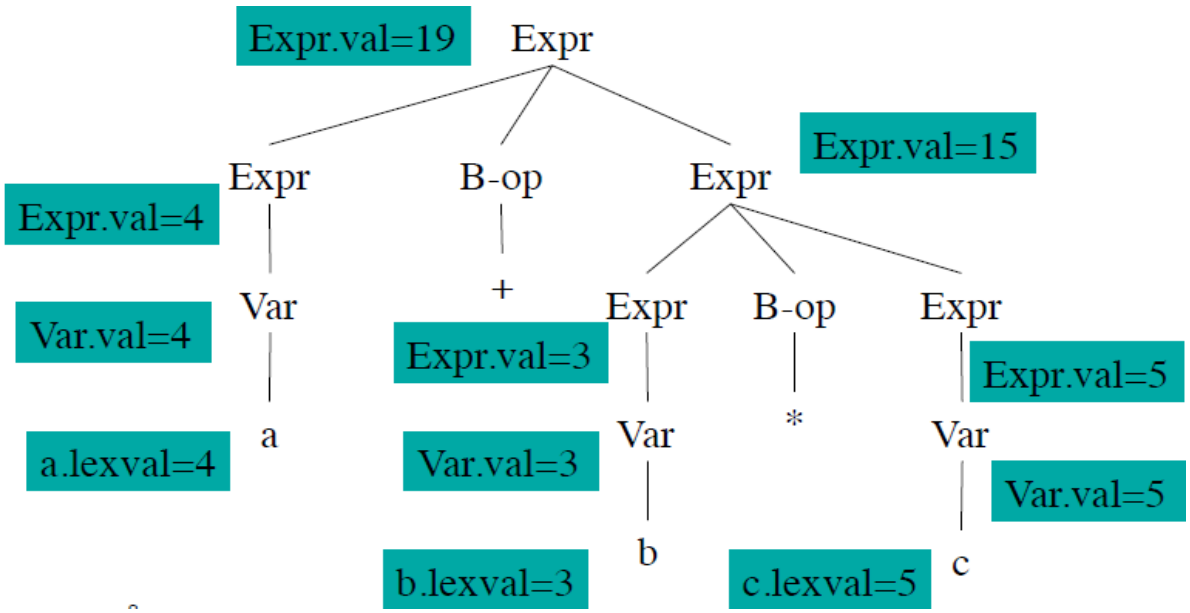
- Since this relation is syntax-directed, we associate each CFG rule with a semantics (rules to build an abstract syntax tree)
- In other words, attribute grammars are a method to *decorate* or *annotate* the parse tree

The semantic analyzer receives Abstract Syntax Tree (AST) from the syntax analysis and then attaches attribute information to the AST. This AST are called Attributed AST.
Attributes are two tuple value, <attribute name, attribute value> as shown below:

```
int value = 5;
<type, "integer">
<presentvalue, "5">
```

i.e. for every production, you need to attach a semantic rule.

Example

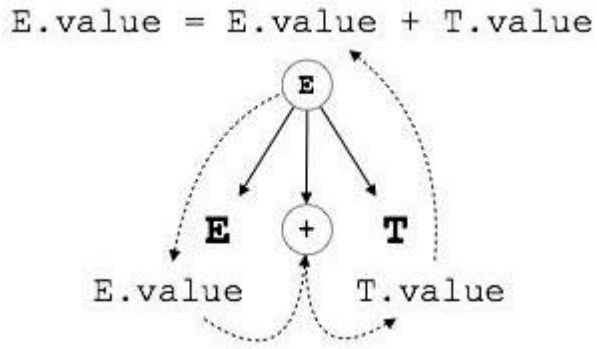


Flow of Attributes in *Expr*

- Consider the flow of the attributes in the *Expr* syntax-directed definition
- The LHS attribute is computed using the RHS attributes
- Purely bottom-up: compute attribute values of all children (RHS) in the parse tree and then use them to compute the attribute value of the parent (LHS)

S-attributed SDT
If an SDT uses only synthesized attributes, it is called S-attributed SDT. Such a grammar plus semantic actions is called an S-attributed definition i.e. a grammar with semantic actions (or syntax-directed definition) can choose to use *only* synthesized attributes

E.g.



As shown above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings. In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production,

```
S → ABC
```

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

Notice that if a definition is S-attributed, then it is also L-attributed, as L-attributed definition encloses S-attributed definitions.

Two important classes of SDTs:
i). LR parser, syntax directed definition is S-attributed
Implementing S-attributed definitions in LR parsing is easy: execute action on reduce, all necessary attributes have to be on the stack
ii). LL parser, syntax directed definition is L- attributed
Implementing L-attributed definitions in LL parsing is similarly easy: we use an additional action record for storing synthesized and inherited attributes on the parse stack

Applications of Syntax-Directed Definitions
SDD can be used at several places during compilation:
i). Building the syntax tree from the parse tree
ii). Various static semantic checking (type, scope, etc.)
iii). Code generation
iv). Building an interpreter

Type Checking

Type checking is verifying that each operation executed in a program respects the type system of the language, i.e., that all operands in any expression are of appropriate types and number**. I**t is used in the following ways:

i). Allow programmers to limit what types may be used in certain circumstances.
ii). Assign type to values.
iii). Determine whether these values are used in an appropriate manner.

Apart from verifying the code to be correct, like checking if a function calls has the correct number and types of parameters, type checking also helps in deciding which code is generated as in the case of arithmetic expressions. There are many variants of type checking, it may be used to check the type of objects and report a type error in case of violation or the incorrect type may be corrected (coercing).

Static Checking and Dynamic Checking

i) Static Checking

In many modern languages type checking done prior to execution (compile-time). It is also known as static checking since properties are verified before the program is run. The following are two important checking:

- Scope checking: checks that all variables and functions used within a given scope have been correctly declared
- Type checking: ensures that an operator or function is applied to the correct number of arguments of the correct types

Advantages:

i) Can catch many common errors.
ii) It is desirable when speed is important i.e. it can result in faster code that does not perform any type checking during execution.

ii) Dynamic checking (runtime checking)

Dynamic checking is performed during execution of a program.

Advantages:

i) It permits programmers to be less concerned with types
ii) It may be required in some cases like array bounds checks which can only be performed during execution.
iii) It may give rise to more robust code by ensuring through checking of values for programme identifiers during execution.

Type systems

i). Strong typing means that the language implementation ensures that whenever an operation is performed, the arguments to the operation are of a type that the operation is defined, for example, do not try to concatenate a string and a floating-point number. This is independent of whether this is ensured statically (prior to execution) or dynamically (during execution).

ii). Weakly typed language gives no guarantee that operations are performed on arguments that make sense for the operation. The archetypical weakly typed language is machine code: Operations are just performed with no checks. Weakly typed languages are mostly used for system programming, where you need to manipulate move, copy, encrypt or compress data without regard to what the data represents.

	static	dynamic
strong	Java	Lisp
weak	C	PERL (1-5)

Type Expressions

These are used to represent the type of language constructs. The following are different kinds of type expressions:

i) **The basic type** and which include: integer, real, character, Boolean, and other atomic types that don't have internal structures. Every programming language will have a set of such basic types. The identifiers used in the language can, either, be of basic types or types derived from them. A special type called type-error is used to indicate there is some type violation.

ii) **Arrays** are specified as *array(I,T)* where *T* is a type expression; *I* is an integer or a range of integers for instance, *array(I,T)* denotes an array of type *T* with *I* elements. In case *I* is a range, it gives possible index values.
E.g.
int a [100]; identifies the type *a* to be *array(100,integer)*
Multidimensional arrays such as : `int p[3][2]`
may be represented as: *array(3,array(2,integer))*

iii) If T_1 and T_2 are two typed expressions, then the Cartesian product $T_1 \times T_2$ is also **a type expression**. Products are usually used to denote anonymous records (i.e. field names are absent) and functions argument lists.
E.g. functions argument list passed to function *funct* with first argument as integer, second as real has the associated type as *integer×real.*

iv) **Named records** are products, but with named elements e.g. a record structure with two named fields: length, an integer and word which is type *array(10,character),* the record type is: *record((length×integer)×(word×array(10,character)))*

v) If *T* is a type expression, then *pointer(T)* is a type expression representing objects which are **pointers to objects** of type *T* e.g.`int *p;` may be represented as *pointer(integer)*

vi) **Functions** map a collection of types to another. They are represented by the type expression $D \rightarrow R$, where *D* is the domain and *R* is the length of the function. i.e. Function types map a domain type D to a range type R. Both *D* and *R* are type expressions e.g.

- The type expression *integer×integer→character* represents a function that takes two integers as arguments and returns a character value;
- *integer→(real→character)* Represents a function that takes an integer and returns a function which maps a character from a real. Notice that many languages restrict return values not to include array values or functions.

Example

Consider the following function:

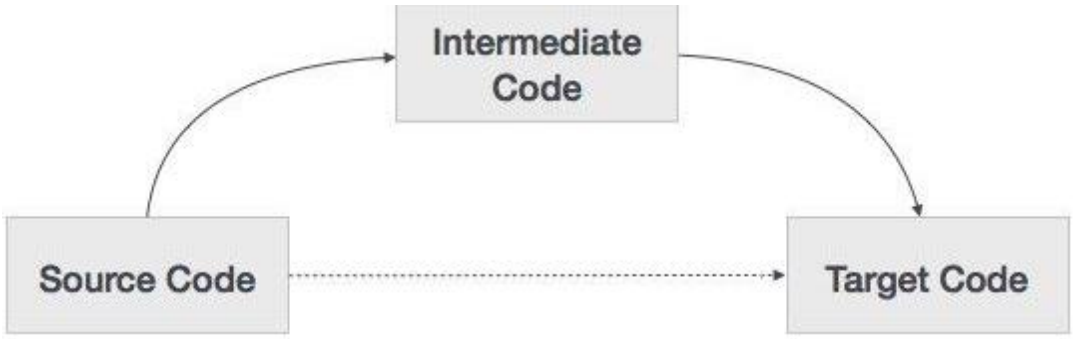
```
int foo (int p, char q){ return2;}
```

The type expression is: $integer \times char \rightarrow integer$

Note: A strongly typed language (sound type system) is one which the compiler can verify that the program will execute without any errors. All checks are made static. It completely eliminates the necessity of dynamic type checking.

INTERMEDIATE CODE GENERATION

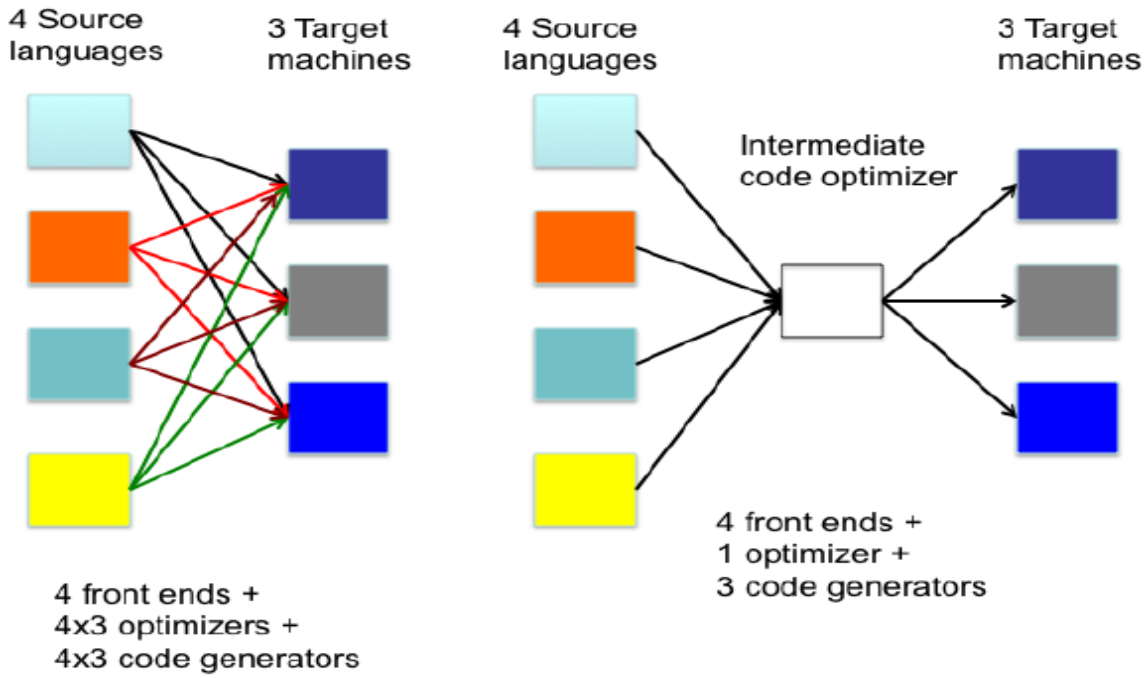
Compilers are designed to produce an intermediate output which represents the input program in some hypothetical language or data structure known as intermediate code that signifies representation between the source language and the target machine language program as depicted below



Generally, while generating machine code directly from source code is possible, it entails two problems

- With m languages and n target machines, we need to write m front ends, m× n optimizers, and m × n code generators
- The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused

By converting source code to an intermediate code, a machine-independent code optimizer may be written This means just m front ends, n code generators and 1 optimizer



Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language-specific (e.g., Byte Code for Java) or language-independent (three-address code).

Intermediate Representation Techniques

a) High level representation

The following are the various ways of representing the input program in high level intermediate representation:

i) Abstract Syntax Trees (AST)
ii) Directed Acyclic Graphs (DAG)
iii) P-code

i) Abstract Syntax Trees (Syntax Trees)

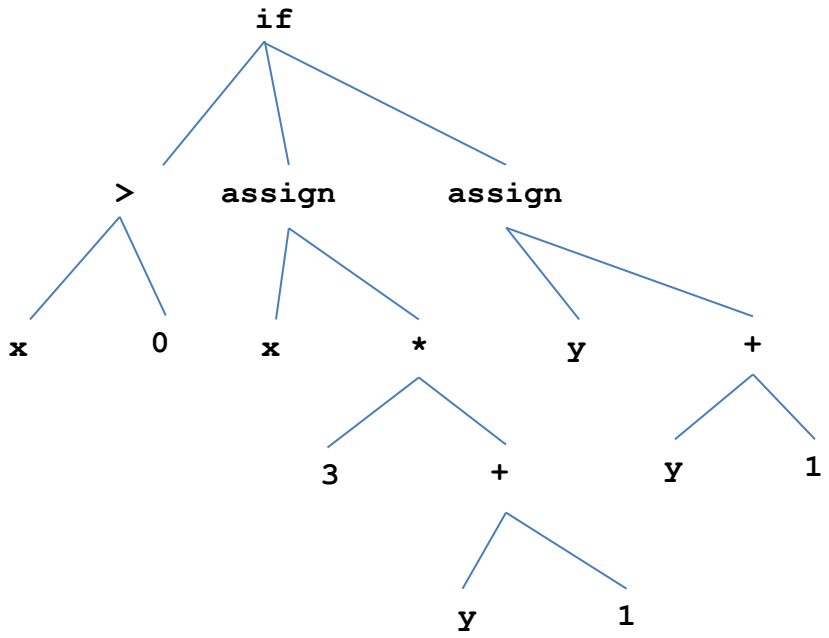
An abstract syntax tree is a compacted form of a parse tree that represents the hierarchical structure of the program. The nodes represent operators while the children of a node represent the operands on which the operators operate.

Example:

Consider the following piece of code in the source language:

```
if x>0 then x=3*(y+1) else y=y+1
```

The AST is as shown below:

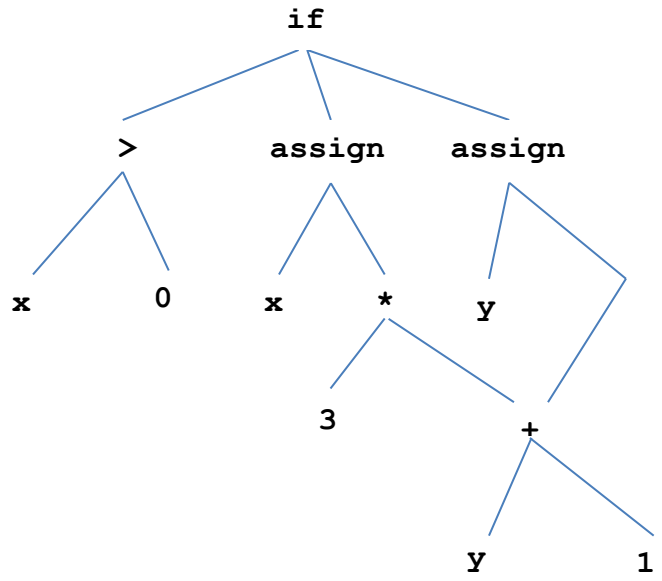


ii) Directed Acyclic Graphs (DAG)

These are similar to AST except that common sub-expressions are represented by a single node.

Example:

The following is the DAG for the AST above:



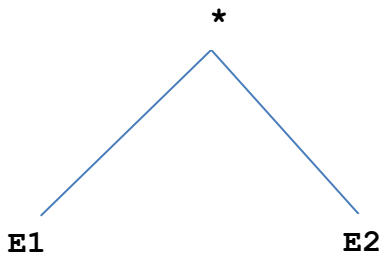
iii) P-code

This representation is used for stack based virtual machines. In this machine, the operands for the operators are always found on top of the stack.

Example

For the following AST, the code is as follows:

```
Code to evaluate E1
Code to evaluate E1
Mult
```



b) Low Level Representation

i) Three Address Code

This is a form of low level intermediate representation that permits only one operator to the right hand side (RHS).

Example:

The source language statement that follows:

$x=y*z+w*a$

is translated as follows:

$t_1=y*z$
$t_2=w*a$
$x=t_1+t_2$