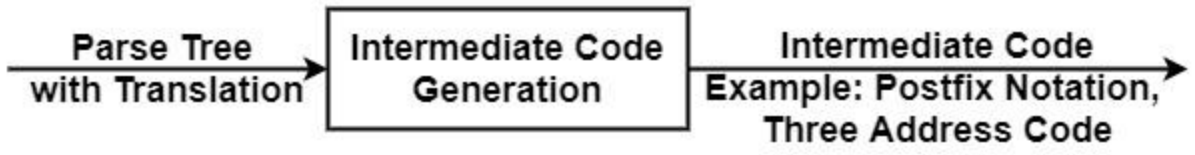
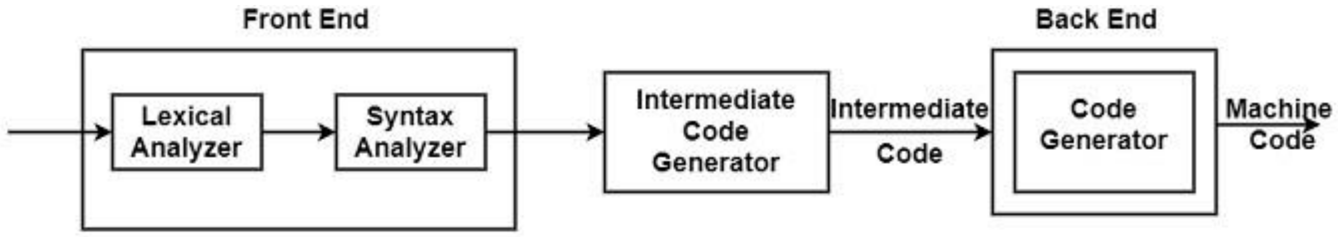
Intermediate Code Generation

Intermediate code can translate the source program into the machine program. Intermediate code is generated because the compiler can't generate machine code directly in one pass. Therefore, first, it converts the source program into intermediate code, which performs efficient generation of machine code further. The intermediate code can be represented in the form of postfix notation, syntax tree, directed acyclic graph, three address codes, Quadruples, and triples.


Parse Tree with Translation
Intermediate Code Generation
Intermediate Code Example: Postfix Notation, Three Address Code

Intermediate Code Generation

If it can divide the compiler stages into two parts, i.e., Front end & Back end, then this phase comes in between.



Position of Intermediate Code Generation

Intermediate representation

An intermediate representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive to further processing, such as optimization and translation. A "good" IR must be *accurate* – capable of representing the source code without loss of information – and *independent* of any particular source or target language. An IR may take one of several forms: an in-memory data structure, or a special tuple- or stack-based code readable by the program. In the latter case it is also called an *intermediate language*.

An **intermediate representation** is any representation of a program "between" the source and target languages.

In a real compiler, you might see *several* intermediate representations!

A true *intermediate* representation is quite **independent** of the source and target languages, and yet very general, so that it can be used to build a whole family of compilers.

We use intermediate representations for at least four reasons:

1. Because translation appears to *inherently* require analysis and synthesis. Word-for-word translation does not work. We need a conceptual model of the program, one that we can easily create from the source code, and one that is easy to construct target code from.

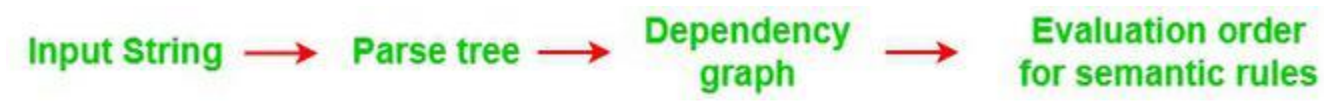
2. To break the difficult problem of translation into two simpler, more manageable pieces.

3. To build re-targetable compilers:

   - We can build new back ends for an existing front end (making the source language *more portable across machines)*.
   - We can build a new front-end for an existing back end (so a new machine can quickly get a set of compilers for different source languages).
   - We only have to write 2n half-compilers instead of n(n−1) full compilers. (Though this might be a bit of an exaggeration in practice!)
4. To perform *machine independent* optimizations.

Translation Of Declarations

Parser uses a CFG(Context-free-Grammar) to validate the input string and produce output for the next phase of the compiler. Output could be either a parse tree or an abstract syntax tree. Now to interleave semantic analysis with the syntax analysis phase of the compiler, we use Syntax Directed Translation.

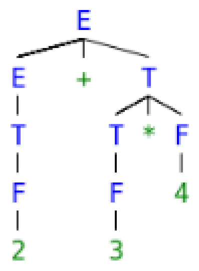


Conceptually, with both syntax-directed definition and translation schemes, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse tree nodes. Evaluation of the semantic rules may generate code, save information in a symbol table, issue error messages, or perform any other activities. The translation of the token stream is the result obtained by evaluating the semantic rules.

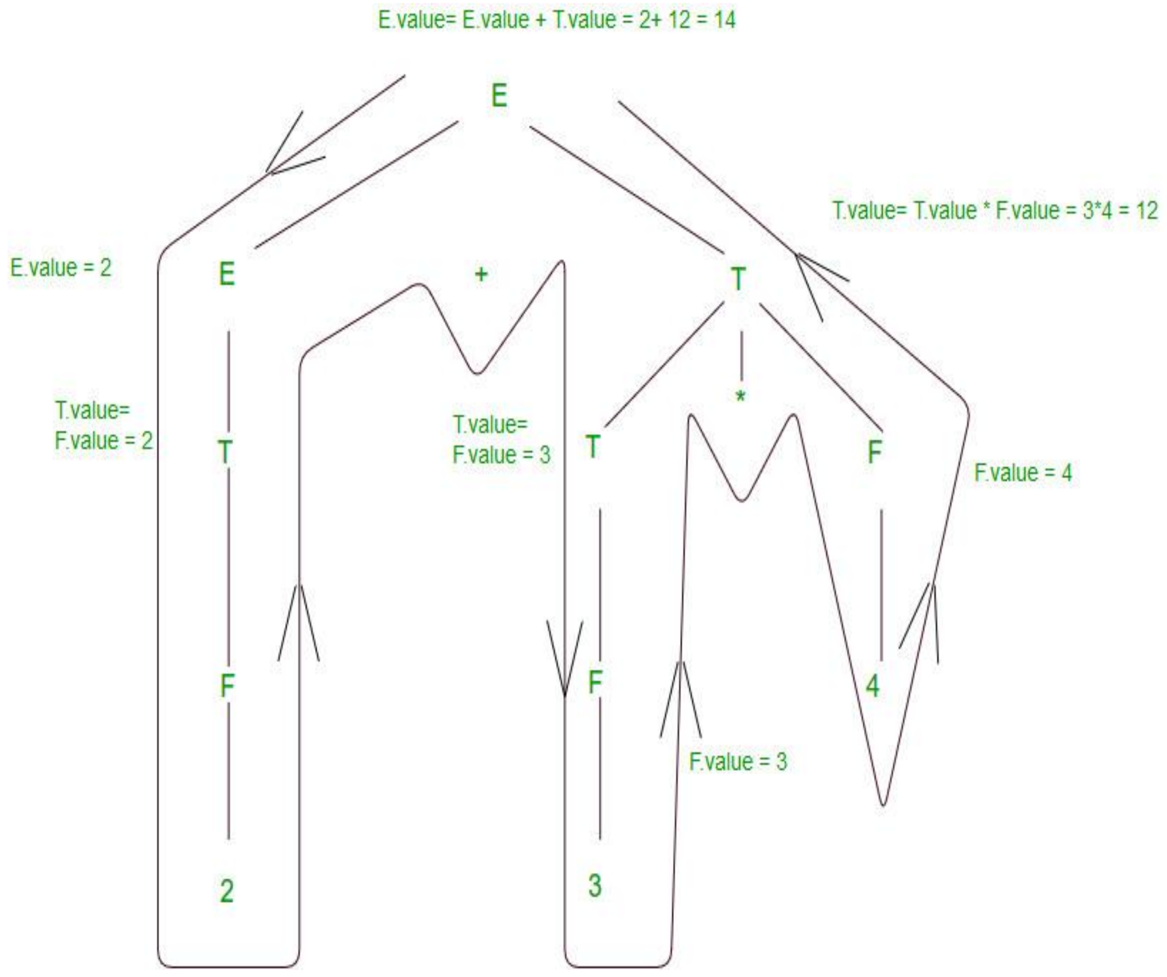
Definition

Syntax Directed Translation has augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down to the parse tree in form of attributes attached to the nodes. Syntax-directed translation rules use 1) lexical values of nodes, 2) constants & 3) attributes associated with the non-terminals in their definitions.

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. This is possible only because SDT rules don't impose any specific order on evaluation until children's attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best-suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom-up in the left to right fashion for computing the translation rules of our example.



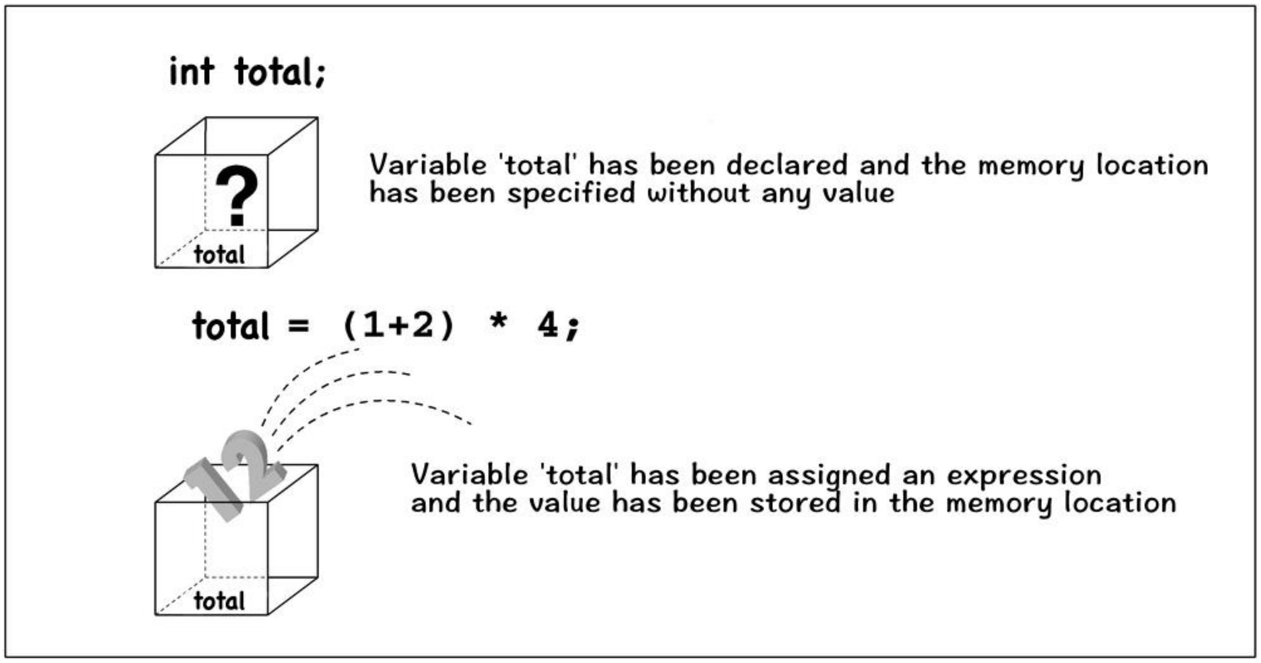
The above diagram shows how semantic analysis could happen.

Assignment

In computer programming, an **assignment statement** sets and/or re-sets the value stored in the storage location(s) denoted by a variable name; in other words, it copies a value into the variable. In most imperative programming languages, the assignment statement (or expression) is a fundamental construct.

Today, the most commonly used notation for this operation is *x* = *expr* (originally Superplan 1949–51, popularized by Fortran 1957 and C). The second most commonly used notation is[1] *x* := *expr* (originally ALGOL 1958, popularised by Pascal).[2] Many other notations are also in use.

Assignment statement allows a variable to hold different types of values during its program lifespan. Another way of understanding an assignment statement is, it stores a value in the memory location which is denoted by a variable name.


int total;
?
total
Variable 'total' has been declared and the memory location has been specified without any value
total = (1+2) * 4;
12
total
Variable 'total' has been assigned an expression and the value has been stored in the memory location

Control flow analysis

In computer science, control-flow analysis (CFA) is a static-code-analysis technique for determining the control flow of a program. The control flow is expressed as a control-flow graph (CFG). For both functional programming languages and object-oriented programming languages, the term CFA, and elaborations such as *k*-CFA, refer to specific algorithms that compute control flow.

For many imperative programming languages, the control flow of a program is explicit in a program's source code. As a result, interprocedural control-flow analysis implicitly usually refers to a static analysis technique for determining the receiver(s) of function or method calls in computer programs written in a higher-order programming language. For example, in a programming language with higher-order functions like Scheme, the target of a function call may not be explicit: in the isolated expression

```
(lambda (f) (f x))
```

it is unclear to which procedure `f` may refer. A control-flow analysis must consider where this expression could be invoked and what argument it may receive to determine the possible targets.

Control flow

Control flow can be defined as the order in which statements are executed or evaluated. Control flow statements include conditional statements, branching statements, and looping statements.

statements.
This control flow can be decided with the help of a boolean expression(an expression that evaluates to either true or false). The translation of boolean expressions is connected to the translation of statements like if-else statements and while statements. Boolean expressions are frequently used in programming languages to-

1. **Compute the logical values**: True or false can be represented as values in a boolean expression. These boolean phrases can be evaluated using three-address instructions and logical operators, just like arithmetic expressions.
2. **Alter the flow control: Boolean expressions are utilized as conditional expressions in statements that change the flow of control. The value of such boolean expressions is implicit in the program's position.** For example, if(E) S, where E is a boolean

expression, and 'S' is the statement. Control will reach S only if the boolean expression E evaluates to true.

Boolean Expression
The translation of conditional statements such as if-else statements and while-do statements is associated with Boolean expression's translation. The main use of the Boolean expression is the following:

- Boolean expressions are used as conditional expressions in statements that alter the flow of control.
- A Boolean expression can compute logical values, true or false.

Boolean expression is composed of Boolean operators like &&, ||, !, etc. applied to the elements that are Boolean or relational expressions. E1 **rel** E2 is the form of relational expressions.
Let us consider the following grammars:
B => B1 | | B2
B => B1 && B2 |
B => !B1
B => (B)
B => E1 **rel** E2
B => true
B => false
If we compute that B1 is true in the first expression, then the entire expression will be true. We don’t need to compute B2. In the second expression, if B1 is false, then the entire expression is false.
The comparison operators <, <=, =, !=, >, or => is represented by **rel**.op.
We also assume that || and && are left-associative. || has the lowest precedence and then &&, and !.

PRODUCTION	**SEMANTIC R RULES**
B => B1 | | B2	B1.true = B.true B1.false = newlabel () B2.true = B.true B2.false = B.false B.code = B1.code || label(B1.false) || B2.code
B => B1 && B2	B1.true = newlabel () B1.false = B.false B2.true = B.true B2.false = B.false B.co B1.code | | label(B1.true) | | B2.code
B => !B1	B1.true = B.false B1.false = B.true B.code = B1.code
B => E1 **rel** E2	B.code = E1.code | | E2.code | | gen(‘if’ E1.addr rel.op E2.addr ‘goto’ B.true) | | gen(‘goto’ B.false)

B => true	B.code = gen('goto' B.true)
B => false	B.code = gen('goto' B.false)

The below example can generate the three address code using the above translation scheme:

```
if ( x < 100 || x > 200 && x ! = y ) x = 0;
          if x < 100 goto L2
          goto L3
L3:      if x > 200 goto L4
         goto L1
L4:      if x != y goto L 2
          goto L1
L2:      x = 0
L1:
```