# Compilers and code generation

## What does a compiler do?

A compiler translates a program in a higher level language, such as C, C++, Java, ML, Fortran, LISP, or Scheme, into an equivalent (in some sense) machine language or assembly language program. Consider the following assignment statement.

```
x := y + 13
```

If our goal is to translate this into a sequence of assembly-language instructions for the TC-201 machine, we should get a result resembling the following.

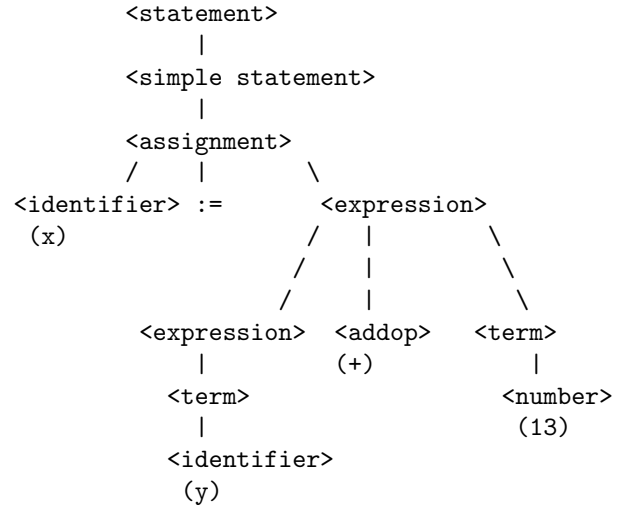
```
load y
add constant13
store x
```

where `x`, `y`, and `constant13` are symbolic addresses of words allocated elsewhere in the final program, with the contents of `constant13` initialized to the constant 13.

A compiler can be viewed as consisting of a sequence of phases comprising lexical analysis, syntactic analysis, semantic analysis, code generation, and optimization.

Lexical analysis takes the string of characters in the input and divides it into a sequence of "tokens." In the example above, the tokens might be the identifiers `x` and `y`, the number `13`, the assignment operator `:=` and the plus sign `+`. In this grammar, whitespace is ignored and does not cause a token to be created.

Syntactic analysis takes the sequence of tokens and parses it according to the grammar of the programming language being compiled. If there is no correct parse, the compiler should (in the best of all possible worlds) produce an informative error message indicating what the problem is. If there is a correct parse, the compiler produces a parse tree, which is used in the subsequent phases. A successful parse of our example might result in the following parse tree.




Here we're assuming a grammar that parses the sequence of tokens

```
<identifier> := <identifier> + <number>
```

and makes the identifier name, constant number, or adding operation an annotation on the parse tree.

The phase of semantic analysis operates on the parse tree, and does type-checking, among other things. For example, if the language being parsed has variables of different types, say integer and floating point number, then it may be that an attempt to add an integer to a floating point number is not permitted (and results in an error message), or is permitted but causes a conversion (eg, of the integer to a floating point number) before the operation.

If the phase of semantic checking detects no errors, then the phase of code generation translates the parse tree into machine language instructions. In the next section we examine this phase more closely.

## Code generation in more detail

Code generation can be understood as processing the parse tree recursively. We consider this idea in connection with the example parse tree above. The assignment statement uses the following production.

```
<assignment> ::=  <identifier> := <expression>
```

We recursively generate code to compute the value of the `<expression>` in the accumulator and then store it in the address assigned to the variable, which according to the annotation is `x`. Thus, we have the following overall structure.

```
[code to compute <expression>, value in accumulator]
store x
```

For the recursive call with `<expression>`, the parse tree indicates the production

```
<expression> ::= <expression> <addop> <term>
```

This can be handled by recursively generating the code for the `<expression>` on the left and then adding to it the value of the `<term>`, which is either an `<identifier>` or a `<number>`. Thus we have

```
[code to compute <expression>, value in accumulator]
add constant13
store x
```

This second `<expression>` in the parse tree involves the productions

```
<expression> ::= <term>
<term> ::= <identifier>
```

This means that the value can be loaded from the location labeled by the identifier, namely `y`. Thus, the following sequence of code is generated.

```
load y
add constan13
store x
```

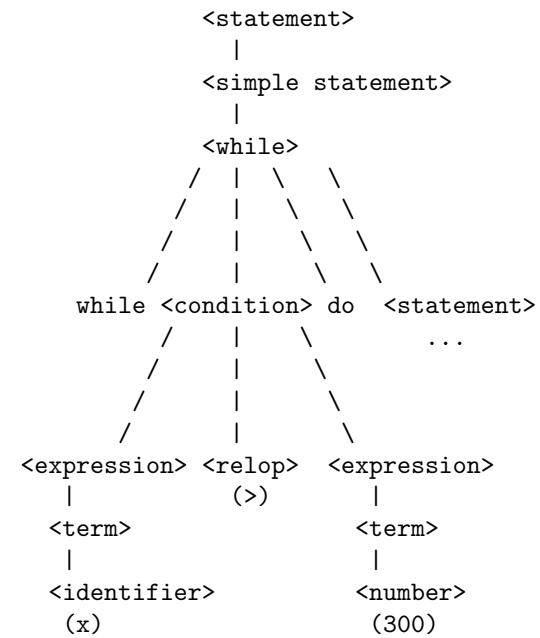
Of course, we also need to make provision for allocating memory words for `x`, `y`, and `constant13`, and initializing them appropriately.

## Generating code for a while statement

Suppose we want to generate code for the following statement.

```
while x > 300 do
  x := x - 13
```

The parse tree for this statement is the following.




The dots (`...`) under the `<statement>` on the right indicate that it is parsed similarly to the preceding assignment statement.

The semantics of a while statement are that we should test the condition before we execute the body. If the condition is false, we should jump to the code following the while statement. If the condition is true, we should execute the code corresponding to the body of the while and then jump back to test the condition again.

The overall structure of the code for the while might be as follows.

```
test1: [compute value of left expression in accumulator]
       store temp1
       [compute value of right expression in accumulator]
       store temp2
       load temp1
       sub temp2
       skippos
```

```
       jump end1
       [code for the body of the while]
       jump test1
end1:  [next block of code]
```

Note that the this test is specific to the relational operator `>`. Other code would be needed for the other relational operators. Recursively, the code to compute the left expression is just `load x` and the code to compute the right expression is just `load constant300`, assuming that there is a `data` statement to define `constant300` somewhere in the program. The code for the assignment statement in the body of the while is determined as for the previous example, resulting in the following translation.

```
test1: load x
       store temp1
       load constant300
       store temp2
       load temp1
       sub temp2
       skippos
       jump end1
       load x
       sub constant13
       store x
       jump test1
end1:  [next block of code]
```

## And optimization?

Note that this process has generated excess loads and stores to temporaries. For example, instead of jumping back to `test1`, because `x` is already in the accumulator, we could jump back to the instruction following `test1`, saving an extra load. If we had computed the expressions in the other order and recognized that they didn't need temporaries, we could have saved some instructions as follows.

```
       load x
here1: sub constant300
       skippos
       jump end1
       load x
       sub constant13
       store x
       jump here1
end1:  [next block of code]
```

This are examples of local optimizations which may be performed in the optimization phase of the compilation process. "Optimization" is a bit of a misnomer, since the goal is to improve the space or time efficiency of the resulting code, with no particular guarantee of its "optimality." A local optimization is one that can be made on the basis of examining a few neighboring instructions; a global optimization requires examination of much more of the program.