Compiler

> Word soup of the modules in a compiler
> List of compiler pieces

A **compiler** is a computer program that translates computer code written in one programming language into another programming language. The first language is called the source language, and the code is called source code. The second language is called the target and can usually be understood by computers. In that case, the instructions become machine code.

If a compiler can convert the same instruction text into machine code for different computers (like smartphones or video game machines), it is a 'cross-compiler'. If the compiler can make instruction text that is easier for people to read, it is a 'de-compiler'. People who write these instructions are called programmers. Some even made programs that can translate the instructions that describe how a compiler should work, into a compiler. That kind of program is called a compiler-compiler.

A compiler usually has three steps. It reads the text and makes notes about how the instructions go together. If the instructions don't make sense, it will try to tell the programmer. Then it will use what it knows about the target language to make the instructions fit better. It then writes down the instructions in the target language. If the source instructions are on different pages, it may have to compile several before it can write everything down.

Compiling the language

[change | change source]

A compiler has six parts :

The first piece, a lexical analyzer, reads a page of instruction text and splits it into words and sentences. It also marks the type for each word. For example, a word may be number, a variable, a verb, a math sign, or an adjective. The output of a lexical analyzer is a list of tokens.

```
int x = 5 + y;
```

```
(type, int) (unknown word, x) (math sign, =) (number, 5) (math
sign, +) (word already seen, y)
```

A parser reads in the output of the lexical analyzer, and verifies the syntax of the language. If the input program is syntactically incorrect or has a type error, it generates errors and warnings. This is called semantic analysis. For example, the parser might complain about the example above, if it *had not* already seen y with its type adjective. The parser uses all the information to make a structure typically called an Abstract Syntax Tree (AST).

```
`( sentence
  ( write value to unknown
    `( unknown
      `( type, int ),
      `( name, x ) ,
      `( value
        (add
          `( number, 5 ),
          `( seen
            `( type, int ),
             `( name, y ),
             `( value, 0 )
) ) ) ) ) )
```

The next piece, an optimizer rearranges the tree structure so the target language is optimized. 'Optimized' includes using fewer instructions to perform the same work. This could be important if the final program needs to check a lot of data. (Like seeing how many people, in the whole country, are fifty years old *and* buy medicine.) An 'optimized' result might also mean breaking long instructions into smaller ones. Unused and unaccessible code will be removed at this step.

A programmer usually tries to write instructions in small, related groups. That way, they can keep track of fewer changes in the program. But, that means the code may go on several pages. When a compiler sees that the target program uses several pages to explain the whole recipe, it may use a linker. The linker will put instructions that say where to find the code that's next. Finally, the compiler writes down the instructions in the target language.

```
# instruction, spot, spot, target
ADD 0, 5, spot_1
LOAD location_y, , spot_2
ADD spot_1, spot_2, spot_3
SAVE spot_3, , location_x
```

Programmers who write compiler code try to make it as perfect as possible. If the programmer writes incorrect code, the compiler informs of them an error, but if the compiler code itself contains errors, it may be hard to tell where exactly the problem lies.

Variants

[change | change source]

At the end of each compilation step the partial finished product could be stored and then only processed later on. A language like Java uses this successfully, where they lack the final translation step to instructions the processor understands. They only do the final translation step once the Java program is running on a computer. This is either called "interpreting" or "JIT"ting, depending on the technique used.

Example

[change | change source]

For example, the source code might contain an equation, such as "x = 5*10 + 6 + 1". The lexical analyzer would separate each number and symbol (such as "*" or "+") into separate tokens. The parser would note the pattern of tokens, as being an equation. The intermediate-code generator would write a form of coding which defines a storage variable named "x" and assigns the numerical product of 5*10 plus 6 and 1. The optimizer would simplify the calculation, of 5*10+6+1, as being just 57. Hence, the target machine-code generator would set a variable named "x" and put the value 57 into that storage place in the computer's memory, using the instructions of whichever computer chip is being used.

Other websites

[change | change source]

- Compiler -Citizendium

This short article about technology can be made longer. You can help Wikipedia by adding to it.