Compiler

From OSDev Wiki

A compiler is a program that translates code from a source programming language into target programming language. Usually the target programming language is of a lower level than the source programming language - compilers target languages such as C, byte code, assembly, or raw machine code.

A popular reason for languages to target another high-level language such as C is that compilers can focus purely on translating the language, while getting all of the world-class optimizations of a production quality C compiler. This process is much more difficult for dynamic languages.

Most compilers output assembly of some form or another, and pass it to an assembler. More rarely, some compilers may directly generate byte code or machine code without a third party assembler. (JIT compilers are a special class of embedded compilers that directly generate machine code.)

Contents

Other tools

Here are some other common tools

Assemblers

A compiler takes a high level code that is often severely abstracted away from the underlying machine instructions that will eventually execute.

An assembler takes assembly code and outputs machine code. Assembly code is a one-to-one representation of machine instructions. Often the only transformation an assembler does is fill in labels (or keep a list of 'fixup' positions - which are filled in when multiple objects are linked together). More advanced assemblers may have macros or some kind of preprocessor, but often these macros generate machine instructions, so there is no abstraction away from the underlying instructions as a high level language could.

The assembler can either generate a executable binary file (perfect for running) or an object file. An object file is different to an executable binary file, as it usually includes a list of exported symbols and/or fixup addresses. Object files may either referred to files that can be loaded dynamically at runtime (dynamic libraries) or files that are intended to be merges with other object files, to share their labels and to produce the final binary file.

Linkers

A linker takes multiple object files, and combines them into either a single object file or the final binary file.

Linkers are often used when you want to link your code with a library, or when your program has multiple source files, each compiled individually, and you want to produce one binary file.

Each object file contains a list exported symbols and where they are located, and a list of 'fixup' positions and the symbol it's looking for. The linker then attempts to fill in the fixup positions with symbols from other files.

Stages

Compilers have multiple stages. Advance compilers that support multiple input languages or multiple output languages are usually divided into a front-end, middle-end, and back-end - where the middle and back ends can be swapped. Simpler compilers that only support one input language and one target language often do not bother to separate these ends.

Front End

The front end of a compiler refers to the part of the compiler that reads the input language. In large compilers like GCC and LLVM that support multiple languages, the front ends and back ends can be mixed and matched to support different languages while sharing the same middle end.

Lexical Analysis

Lexical analysis reads the source file and splits it into tokens. A token usually has a type, which may be a symbol such as FOR, INT, VOID. A token may also be a literal such as STRING_LITERAL, IDENTIFIER, FLOAT_LITERAL with extra data (such as the value of the literal or name of the identifier) attached. The lexer usually handles skipping over comments.

Lexers can either be written by hand or they can be generated with a 'lexer generator' - which is a tool that takes in a lexical grammar and generates the code of a lexer.

Parsing

The parsing stage reads the tokens from the lexer, and builds some form of representation of the code in memory. The representation is usually an abstract syntax tree.

A parser can either be written by hand (research Recursive descent parser (http://en.wikipedia.org/wiki/Recursive_descent_parser)) or generated with a 'parser generator' - which is a tool that takes in a language grammar and generates the code of a parser. It is during this stages that local variables and object names are registered as they are encountered.

Some parser generators generate both parsers and lexers in one go and are called 'compiler compilers'.

Abstract syntax tree

An abstract syntax tree is the representation of a program in tree format, and is usually the representation of the source code in tree format. The abstract syntax tree is usually closely tied to the input language and hence part of front-end, but sometimes it may be part of the middle-end. The abstract syntax tree is then converted into a language-independent intermediate representation.

Middle end

The middle end acts as glue between the front and back ends. This is in an intermediate representation (http://en.wikipedia.org/wiki/Intermediate_representation#Intermediate_representation) that is language-agnostic, and most optimizations are done here. Not all compilers have middle ends, sometimes abstract syntax trees directly produce machine code and skip through these steps.

Bytecode

Some compilers have an internal bytecode language (which can be different from bytecode targets) for communicating between front ends and back ends. These bytecodes may either be register-based or stack based (stack based bytecode is incredibly easy to generate from an abstract syntax tree (http://en.wikipedia.org/wiki/Stack_machine#Simple_compilers)).

Single static assignment

The abstract syntax tree or bytecode may be translated into single static assignment (http://en.wikipedia.org/wiki/Static_single_assignment_form) , or SSA, form. SSA form is an intermediate representation of code where all variables are only assigned to once. SSA form is great for optimizations such as constant folding and dead code elimination. It is also an easy form that maps nicely to registers in the backend.

Back end

The back end of a compiler outputs the code into the target language. Backends often perform target-specific optimizations.

Interpreters

Interpreters do not output code so they are technically not compilers, but do implement their front and middle ends similarly to compilers. They may execute the immediate representation, they abstract syntax tree, or as the language is parsing - bypassing a middle end completely. JIT (just-in-time) compilers generate machine code in memory and execute it immediately.

Bytecode

A compiler may target bytecode. Bytecode is designed to be executed inside of a virtual machine, and is often simpler to target (due to not having to allocate a limited number of registers, and since it's higher level) than assembly.

Assembly

Compilers may target assembly language. This is similar to targeting bytecode, except the compiler must allocate registers, and often deal with register and stack allocations, and calling conventions (http://en.wikipedia.org/wiki/Calling_convention) .

By outputing assembly language, the compiler leaves it up to the assembler to implement the object and executable formats, and to calculate code positions, labels, and encode the instructions in machine code.

Machine code

Compilers may generate machine code (such as JIT compilers). This is often done when the compiler has all of the input code needed and does not need the output file to be merged with other output files. The process for outputting machine code is very similar to outputting assembly code, but the compiler must now resolve label positions, encode machine instructions, and deal with outputting in an executable format.

See Also

- Category:Compilers

Retrieved from "http://wiki.osdev.org/index.php?title=Compiler&oldid=17369"

- This page was last modified on 14 December 2014, at 21:43.
- This page has been accessed 2,543 times.