ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

Engineering A Compiler: An Incremental Approach

Ch M Krishna Mentada
D No: 11-1-23, Mandala Street,
Srikakulam, A.P – 532001, India.

Abstract

Compilers are actually the magical artifacts of computers. There are complex programs that translate code in a source language to a target language. Real life compilers are kind of very complex for understanding on an academic basis. Many books that teach how to design compilers portray that the design is like killing a giant monster with a very blunt sword. A lot of tradeoff exists between the designs of compilers on an academic scale to the design of compilers for real-time deployment. Hence many people think that it is just feasible to code an interpreter. This paper aims to break this barrier. This paper illustrates all the tools and the various methodologies that are used to design a simple compiler. Here a simple set of python called S3L will be accepted by the compiler and produces assembly code for the x86 architecture.

Every section described in this paper if followed incrementally will result in a full-fledged compiler .Supporting material for the tutorial such as an automated testing facility coupled with a comprehensive test suite are provided with the tutorial. The motivation of this paper is to provide a simplified design strategy compilers so that students can be better exposed to develop high performance compilers in future.

Keywords—Compilers, Interpreters, Instruction Set, Python.

Introduction

Compilers, it seems to be a very lengthy and hard program to code. But that's not a real concept, even though it may be tough but following a correct order and knowing the basics about a compiler [1], it will work out easily. There are many books which will help you out with theory concepts about the compiler but implementing those in the practical world will be difficult.

Since it's a heavy task so people tend to write an interpreter than a compiler, but there are many problems, basically all languages can't be interpreted. These can't be mapped on to compilers because compilers are far better and helpful than interpreters [2].

Before we begin our journey through building a compiler let's talk about what compilers mean, steps involved, at are interpreters so on one by one.

Compilers

A compiler is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (syntax-directed translation), code generation, and code optimization [3]. Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementers invest significant effort to ensure compiler correctness.

Cite this article as: Ch M Krishna Mentada, "Engineering A Compiler: An Incremental Approach", International Journal & Magazine of Engineering, Technology, Management and Research, Volume 5 Issue 2, 2018, Page 72-78.

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

Compilers enabled the development of programs that are machine-independent. Compilers bridge source programs in high-level languages with the underlying hardware. A compiler verifies code syntax, generates efficient object code, performs run-time organization, and formats the output according to assembler and linker conventions. Compilers are sometimes classified as single-pass, multi-pass, load-go depending on how they have been constructed or on what function they are supposed to perform.

Interpreters

An interpreter is a computer program that directly executes, i.e. performs, instructions written in a programming or scripting language, without previously compiling them into a machine language program. An interpreter generally uses one of the following strategies for program execution:

1. Parse the source code and perform its behavior directly.
2. Translate source code into some efficient intermediate representation and immediately execute this.
3. Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

Source programs are compiled ahead of time and stored as machine independent code, which is then linked at run-time and executed by an interpreter (for JIT systems). Interpreters of various types have also been constructed for many languages [2].

Types of compiler

1) A source-to-source compiler is a type of compiler that takes a high level language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language program as an input and then transform the code and annotate it with parallel code annotations (e.g. OpenMP) or language constructs.

2) Bytecode compilers that compile to assembly language of a theoretical machine, like some Prolog implementations. Bytecode compilers for Java, Python are also examples of this category.

3) Just-in-time compiler (JIT compiler) is the last part of a multi-pass compiler chain in which some compilation stages are deferred to run-time. Examples are implemented in Smalltalk, Java and Microsoft .NET's Common Intermediate Language (CIL) systems.

4) Hardware compilers (also known as syntheses tools) are compilers whose output is a description of the hardware configuration instead of a sequence of instructions.

Phases of Compilation:

A compiler is likely to perform many or all of the following operations:

Lexical analysis:

This phase of the compiler does the actual reading of the source program in a stream of characters and the output is a collection of meaningful characters called tokens.

Parsing:

This phase receives the source code in the form of tokens from the lexical analyzer and performs "Syntax Analysis". This phase determines the structural elements of the program and their relationships. The output is a "Parse Tree".

Semantic analysis (syntax-directed translation):

Semantics of the program determine the runtime behavior. Static semantics refer to the semantics of the program that can be determined during compilation phase. The job of determining the runtime behavior of the program is the job of the semantic analyzer [3].

Source Code Optimizer:

This part of the compiler includes a number of code improvement or optimization steps. The output is the machine independent intermediate code.

Code generation:

This phase takes the intermediate code as the input and

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

generates the equivalent code for the target machine.

Code optimization:

This phase optimizes the code generated by the previous phase and the output is an optimized machine dependent target code.

The Data Structures Used in a Compiler:

Symbol Table:

This data structure keeps track of the associated identifiers, functions, variables, constants and data types. This is heavily used almost in every phase. So hence the insertion, deletion and access operations should be highly optimized and run in a constant-time.

Literal Table:

This table includes the constants and strings used in the program. This is heavily used almost in every phase. So hence the insertion, deletion and access operations should be highly optimized and run in a constant-time.

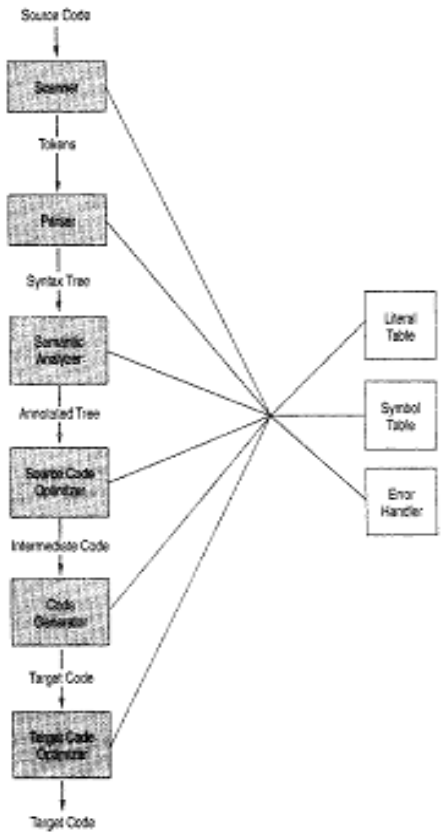

Fig 1. Phases of compiler and data structures

Tools used

Lex and Lex file

It helps in writing the control statements for a program which is given has regular expressions. The regular expressions are written inside a lex file with '.l' extension. It also contains some of the program fragments. All the regular expressions and fragments are translated to programs which read an input stream, which is divided into small parts and that are mapped with the regular expressions given inside a lex file. The validation of the expressions and segments are done by a deterministic finite automation generated by lex. Lex is not a complete language, but rather a generator representing a new language feature.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected.

```
                +-------+
Source ->  |  lex  | -> yylex
                +-------+

                +-------+
Input ->     | yylex | -> Output
                +-------+
```

A lex file

Definition section
%%
Rules section
%%
C code section

Definition section: will import header files and defines macro. In the Definitions section, any line beginning with a "%" character and followed by an alphanumeric word beginning with either s or S defines a set of start conditions. Any line beginning with a "%" followed by a word beginning with either x or X defines a set of exclusive start conditions.

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

Rules section: The rules in lex source files are a table in which the left column contains regular expressions and the right column contains C program fragments to be executed when the expressions are recognized.

C code section: C code section contain functions and C statements.

Yacc and Yacc file

The computer program which takes any input will have to define a structure in which it will accept it. That particular structure is called an "input language". This language can be as complex as a programming or just easy has a set of numbers or numerals. If the input language is not according to the defined language it will be considered or reported as an invalid statement.

Yacc provides a tool for describing the computer program. Yacc coder specifies the structure of his input, together with the segment which has to be invoked when a certain sequence is recognized. Yacc contains subroutines which are invoked; because it is easily possible to go through the code and do any changes inside the subroutine and also it's easy to handle them.

The input streams can be defines as a set of individual input characters, or in terms of higher language constructs such as names or names. The yacc part used for validating the statements are stored in a file with extension '.y'.

Yacc file

Definition section
%%
Rules section
%%
C code section

Definition section: tokens used in grammar are declared here.

Rules section: contains rule of grammar
A rule has the form:

non_terminal: sentential form
| sentential form
.....
| sentential form
;

Actions may be associated with rules and are executed when the associated sentential form is matched.

LLVM (Low Level Virtual Machine)

The Low Level Virtual Machine (LLVM) is a register-based compiler framework which aims to provide a standardized tool-kit for mid-level and front-end language development. Like many virtual machines it works with an intermediate representation language. Some of the sub-systems in LLVM are synonymous to those of the "traditional compiler", but fundamentally LLVM is different. Some of its features which differentiates it with other compilers are

- It provides a standard API and compiler back-end for many different compilers.
- It ships with tools to make platform specific assembler from on-disk byte-code; this can be assembled and linked in order to make an executable binary code.
- It has very strong type checking, which is far stronger than C.
- Has a comprehensive optimizer framework for many specific platforms.
- Has a number of profiling utilities.
- Can be used solely as an interpreter, using byte-code from disk. Can be interfaced by a number of language bindings, allowing parsers and tokenizers to be implemented in high level languages.
- It has the ability to cal any system integrated with the existing operating system.

The top level component of any LLVM assembler program is the module, which acts like a container for one or more functions and perhaps some global variables. Unlike most assembler implementations,

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

LLVM assembler supports the concept of functions. Each function has its own stack frame.

One feature of LLVM is that it is able to optimize at several stages of the compile/execute life-cycle: compile-time, link-time and run-time. LLVM defines a set of optimizer passes, which the user may turn on individually, according to their needs. A typical optimizer pass will transform the bit-code representation of a program, arriving at a new, functionally identical program, which when applied properly, can improve the performance or size of a program. Some optimizer passes do not transform the program at-all. These passes are profiler passes. Instead of altering the program, such passes only analyze it, allowing the developer to spot possible bottlenecks and shortcomings in their programs.

Connecting LLVM and Yacc

Flex is used by Bison to generate tokens that the tokens as asked by Bison parser. Bison parser is optimised to produce an abstract syntax tree. Abstract syntax tree is a diagrammatic representation of the grammar used to parse.

Node.h is a file that contains the code to construct the AST as asked by the parser. The pseudo variables of bison are used to build the AST as and when a recursive call happens.

The class Value is used to represent the values that are parsed in the parser. The other classes NStatement, NExpression, NVariableDeclaration are used to construct the AST. The other classes also follow.

Node.h

```
#include <stack>
#include <llvm/Module.h>
#include <llvm/Function.h>
#include <llvm/Type.h>
#include <llvm/DerivedTypes.h>
#include <llvm/LLVMContext.h>
#include <llvm/PassManager.h>
#include <llvm/Instructions.h>
#include <llvm/CallingConv.h>
#include <llvm/Bitcode/ReaderWriter.h>
#include <llvm/Analysis/Verifier.h>
#include <llvm/Assembly/PrintModulePass.h>
#include <llvm/Support/IRBuilder.h>
#include <llvm/ModuleProvider.h>
#include <llvm/Target/TargetSelect.h>
#include <llvm/ExecutionEngine/GenericValue.h>
#include <llvm/ExecutionEngine/JIT.h>
#include <llvm/Support/raw_ostream.h>

using namespace llvm;
class NBlock;

class CodeGenBlock {
public:
   BasicBlock *block;
   std::map<std::string, Value*> locals;
};

class CodeGenContext {
   std::stack<CodeGenBlock *> blocks;
   Function *mainFunction;

public:
   Module *module;
   CodeGenContext() { module = new
Module("main", getGlobalContext()); }

   void generateCode(NBlock& root);
   GenericValue runCode();
   std::map<std::string, Value*>& locals() { return
blocks.top()->locals; }
   BasicBlock *currentBlock() { return blocks.top()-
>block; }
   void pushBlock(BasicBlock *block) {
blocks.push(new CodeGenBlock()); blocks.top()-
>block = block; }
   void popBlock() { CodeGenBlock *top =
blocks.top(); blocks.pop(); delete top; }
};
```

CodeGen.h is another file that is used to emit the llvm byte code according to the AST as generated by the

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

parser using the Node.h. The stack class of the STL in C++ is used to generate the symbol table. The language proposed here has no scope for global variables. Stack is also used to keep in mind the activation records so that the block entered will be visible to the compiler.

Codegen.h

```
#include <stack>
#include <llvm/Module.h>
#include <llvm/Function.h>
#include <llvm/Type.h>
#include <llvm/DerivedTypes.h>
#include <llvm/LLVMContext.h>
#include <llvm/PassManager.h>
#include <llvm/Instructions.h>
#include <llvm/CallingConv.h>
#include <llvm/Bitcode/ReaderWriter.h>
#include <llvm/Analysis/Verifier.h>
#include <llvm/Assembly/PrintModulePass.h>
#include <llvm/Support/IRBuilder.h>
#include <llvm/ModuleProvider.h>
#include <llvm/Target/TargetSelect.h>
#include <llvm/ExecutionEngine/GenericValue.h>
#include <llvm/ExecutionEngine/JIT.h>
#include <llvm/Support/raw_ostream.h>

using namespace llvm;
class NBlock;

class CodeGenBlock {
public:
   BasicBlock *block;
   std::map<std::string, Value*> locals;
};

class CodeGenContext {
   std::stack<CodeGenBlock *> blocks;
   Function *mainFunction;

public:
   Module *module;
   CodeGenContext() { module = new
Module("main", getGlobalContext()); }
      void generateCode(NBlock& root);
   GenericValue runCode();
   std::map<std::string, Value*>& locals() { return
blocks.top()->locals; }
   BasicBlock *currentBlock() { return blocks.top()-
>block; }
   void pushBlock(BasicBlock *block) {
blocks.push(new CodeGenBlock()); blocks.top()-
>block = block; }
   void popBlock() { CodeGenBlock *top =
blocks.top(); blocks.pop(); delete top; }
};
```

The main.cpp file will run the program and the output will be the target code. Make file is used to automate the process.

Main.cpp

```
#include <iostream>
#include "codegen.h"
#include "node.h"
using namespace std;
extern int yyparse();
extern NBlock* programBlock;

int main(int argc, char **argv)
{
   yyparse();
   std::cout << programBlock << std::endl;

   CodeGenContext context;
   context.generateCode(*programBlock);
   context.runCode();

   return 0;
}
```

Make file

```
all: parser

clean: rm parser.cpp parser.hpp parser tokens.cpp

parser.cpp:
```

ISSN No: 2348-4845
International Journal & Magazine of Engineering, Technology, Management and Research
A Peer Reviewed Open Access International Journal

```
    parser.y
    bison -d -o $@ $^

parser.hpp: parser.cpp

tokens.cpp:
    tokens.l parser.hpp
    lex -o $@ $^

parser:
  parser.cpp codegen.cpp main.cpp tokens.cpp
   g++ -o $@ `llvm-config --libs core jit native --
  cxxflags --ldflags` *.cpp
```

Conclusion

As we have seen the seen the different phases of compiler, learnt about the tools used for construction of a compiler, we can say that if these steps are followed by a programmer he will be able to design a compiler for his own.

This approach also tries to solve the particle and theoretical trade-offs which are present in the construction of any working compiler.

Bibliography

[1] Kenneth C Louden, "Compiler Construction Principles and Practices", Cengage India, 1999.

[2] William M. Waite, Gerhard Goos (1996), “COMPILER CONSTRUCTION”, 347, 111 – 148, 183 – 208.

[3] J. Grosch, H. Emmelmann (1991), “A TOOL BOX FOR COMPILER CONSTRUCTION”, 11.