PROGRAMMING GUIDE

FOR THE CISOR COMPILER

Version 1, August 2022

Author: Philippe Caron

This document contains information about the overall architecture of the CISOR compiler, as well as specific descriptions of every function in the project.

Contents

Definitions

Assembler — The architecture-specific program or set of functions responsible for translating the machine-code from its mnemonic form to its binary form.

Assembly — The machine-code in its mnemonic form.

AST — Abstract Syntax Tree. It's the tree representation of the abstract syntaxic representation of the code (from Wikipedia).

Compiler — The architecture-specific program or set of functions responsible for translating the IR to assembly.

(E)BNF — (Extended) Bachus-Naur Form. It's a metasyntax for context-free grammars (from Wikipedia). The "extended" is in parenthesies because the parser contains a switch that will turn the extended syntax on and off.

Intermediate representation (IR) — The intermediate representation is the code produced by the intermediate compiler. It should be essentialy similar to a virtual machine code.

IR compiler — The program or set of function that translate the code from human-like language (the programming language) to machine-like language (the IR). This is where the bulk of the compilation occurs. The purpose of separating this step is to allow for easier portability. The IR is the same on every processor architecture, but the compiler has to be different since it's producing architecture-specific mnemonic code (assembly).

Linker — The program or set of functions responsible to adjust address references inbetween compiled binaries.

Macro — A macro in the compiling context is a piece of code destined to be executed at compile-time (vs the rest of the code that will be executed at run-time).

Mnemonic — A mnemonic is a human readable token that represents a machine readable byte (or set of bytes). For instance, consider the follwing mnemonics : `ADD=0x1282`, `R1=0x01`, `R3=0x03`, and the following mnemonic instruction (assembly): `ADD R1 R3`. A simple assembler would easily translate this to `0x12820103`, and although the conversion is trivial, a human will find the former much easier to read.

Preprocessor — The program or set of functions responsible for pretreating the code prior to intermediate compilation. The preprocessor takes a C* file and outputs a C* file, only in the output all macros have been expanded.

Symbol — A token with its location in the document or string it was read from.

Token — An indivisible language unit (word) and type (ex: variable, reserved, punctuation, etc.).

Tokenizer — A program or set of functions responsible to split an input stream into a token stream.

1 Introduction

The CISOR compiler aims to be a self-compilable compiler for the C* language. Although the objectives of C* are clear, the language is bound to evolve over the process of its creation, thus the first version of the compiler should be able to quickly adapt to unforeseen changes. The easiest way to achieve this adaptability is to have the grammar as an input the the CISOR compiler. This will undoubtably significantly impact performance, but I believe the advantages are too great to overlook. A version of the grammar description language (E)BNF was chosen for this purpose. It is the most common and appropriate notation for this use-case. The slight modifications in syntax to the official form are discussed further in the parsing section, but are generally present for the sole purpose of facilitating parsing and readability.

2 Architecture

The overall architecture of the compiler rests on three major groups, the general utilities group (`utils`), the parsing API group (`parsing`), and the compiler group (`cisor`). Since the compiler is designed to be self-compilable, third party library use should be reduced to a minimum (read none). The purpose of the general utilies group is to provide those functions that are going to be frequently used in compilable form, even if those utilies are already implemented in a third party library. The parsing API is meant to provide an intuitive set of functions in order to render extremely complex code **easy to use** and **safe**. Through a robust and thoroughly tested parsing API, the risks of memory leaks are significantly reduced, as well the mental charge required to understand and edit the compiler code. It should contain the (E)BNF parser and tokenizer for the program, as well as provide an API for an eventual editor through informative syntax error detection. Finally the compiler group is comprised of the traditionnal compiler pipeline (preprocessor > IR compiler > compiler > assembler > linker), as well as an API for an eventual editor for semantic error detection.