Code Optimization and Transformation Course Project on:

ACSE

The Code Optimization and Transformation course exam is composed by two parts. One is an oral test, the other is an homework, to be terminated before course last call. To pass the whole exam, you must get a pass grade from both the test and the homework. The homework must be taken in pairs.

During the lab classes, the LLVM [6] compiler has been introduced. The homework must use the 3.0 release version of LLVM. A sample project – COT passes [8] – is available on GitHub [4]. It must be used as a starting point for the homework. LLVM testing framework [3] must be used to validate the implementation.

Sources must versioned using Git [9]. A good tutorial can be found here [2]. Sources must be published on GitHub [4].

Assignment

The ACSE [1] compiler is a simple compiler used for lab classes of the Formal Languages and Compiler course [5]. The language it compiles is C-based, but some features are missed.

The goal of this project is to extend the ACSE grammar, in order to recognize a wider language, and implement a simple syntax-driven translator targeting LLVM Intermediate Representation.

You are required to face the following problems:

ACSE Grammar: the ACSE grammar available on Formal Languages and Compilers course site [5] does not correctly implements the precedence rules between unary and binary minus. You are required to fix this problem in grammar definition. The grammar must then improved in order to support C-style function definition. The entry point of the program is the **void** main(**void**) function.

ACSE-to-LLVM Compiler: an ACSE program is entirely contained in a single file. You are required to parse it, and performing translation into the following forms: LLVM

human-readable byte-code, LLVM machine-oriented bit-code, target machine human-readable assembly, and target machine binary-code. The user can select the format of the output, as well the name of the output file using command-line options.

Please notice that the only component you can reuse of the existent ACSE compiler is the grammar!

Advices

On LLVM website [6] there is a good tutorial explaining how you can use LLVM as the back-end for your language [7].

You are free to chose which parsing technique to adopt. You can either implement an hand-coded LL parser or an LALR parser based on `bison`.

Do not be scared from the 4 different output formats you have to support. It is not up to you emitting the code, you have only to correctly setup the LLVM framework in order to build the compilation pipeline leading from LLVM IR to one of the output formats. Look at the implementations of `opt` and `llc` in the sources `tools` directory.

Your compiler must be a stand-alone tool, like `llc`. Please contact the teaching assistant in order to known how you have to modify the sample project [8] in order to edit the building system in order to support tool creation/testing.

References

[1] A. Di Biagio and G. Agosta. Advanced Compiler System for Education. URL `http://corsi.metid.polimi.it`.

[2] Scott Chacon. Pro Git. URL `http://git-scm.com/book`.

[3] John T. Criswell, Daniel Dunbar, Reid Spencer, and Tanya Lattner. LLVM Testing Infrastructure Guide. URL `http://llvm.org/releases/3.0/docs/TestingGuide.html`.

[4] GitHub Inc. GitHub. URL `http://github.com`.

[5] Formal Languages and Compilers Group. Formal Languages and Compilers – CorsiOnline. URL `http://corsi.metid.polimi.it`.

[6] University of Illinois at Urbana-Champaign. Low Level Virtual Machine, . URL `http://www.llvm.org`.

[7] University of Illinois at Urbana-Champaign. LLVM Tutorial, . URL `http://llvm.org/releases/3.0/docs/tutorial`.

[8] Ettore Speziale. Compiler Optimization and Transformation Passes. URL `https://github.com/speziale-ettore/COTPasses`.

[9] Linus Torvalds. Git. URL `http://git-scm.com`.