The UNBC CPSC 425 W16 Compiler Project

The Compiler Project Guidelines

General Guidelines

Jernej Polajnar

January 7, 2016

This document states the general guidelines that must be followed in the compiler project in the *CPSC 425* W16 class. The project is to be developed and submitted in four phases, according to the schedule in the *Course Outline*. Additional guidelines will be provided for every phase of the project. The course documents referenced here will be available on the course web site, `http://web.unbc.ca/mathcs/courses/cpsc425/w16`.

1 Project objectives

The main project objective is to produce a complete, error-free, well-structured and well-documented compiler that complies with the given programming language specifications. The compiler must always terminate, regardless of what input it is given. Any limitations of the compiler must be clearly documented. If the compilation cannot proceed because the program being compiled exceeds some size limitations within the compiler, the compilation should be terminated with a clear error message. Similarly, if a compiled program during execution exceeds the limitations of its execution environment (e.g., the run-time stack overflows), the execution should be terminated with a suitable error message. It is expected that good software engineering practices will be followed in the project.

2 Programming languages

Your compiler must be able to translate any given correct program in the *source* language (Subsection 2.1) to a program in the *target* language (Subsection 2.2). For incorrect source programs it must produce diagnostics as detailed later in the project guidelines. The compiler must be written from scratch (i.e., without the use of specialized compiler-generation tools such as *Lex* or *Yacc*). It must be implemented in an object-oriented *host* language (Subsection 2.3).

2.1 The source language

The source language is a small programming language defined specifically for this project. It is called *C*16* . Its specification, posted on the course web site, includes:

Introduction

Lexical structure

Syntax

Semantics

The source file names have the extension `.cs16`.

2.2 The target language

The mandatory part of the project requires your compiler to generate *three-address intermediate code* in the form of *quadruples*. The details of the language of quadruples for this project are specified in a separate document. You can execute quadruples programs using an *interpreter* for the quadruples language provided at the class web site.

An optional addition to your compiler (for bonus credit) is the generator of TM machine assembly code, as described in *Compiler Construction – Principles and Practice* by K. C. Louden, PWS Publishing, 1997. This is an additional pass that inputs the intermediate program representation in the form of quadruples and generates a target program in TM assembly language. There exist a simulator for the TM machine, written in C, that can execute the generated machine code. Students interested in implementing this phase should contact the course instructor for details.

2.3 The host language and programming style

The host language, in which the compiler is implemented, may be C++ or Java; any other language requires prior authorization by the course instructor.

3 The compiler architecture

The tasks of a compiler are organized into several different *phases*, with each providing its results to the next one. The major phases are: scanning (lexical analysis), parsing (syntax analysis), semantic analysis, and code generation & optimization. If all those tasks are performed during a single traversal of the source program, we have a *single-pass* compiler. Alternatively, if the tasks are performed during several sequential *passes* through the entire program, we have a *multi-pass* compiler. The first pass reads the source program, the last pass produces the target code, and each pass between them receives an intermediate representation of the entire program from the preceding pass and generates another intermediate program representation for the pass that follows. The project described in the main textbook (Louden) is multi-pass.

In your term project you will design and implement a multipass compiler according to the instructions in these guidelines. The compiler will be structured as in Figure 1.

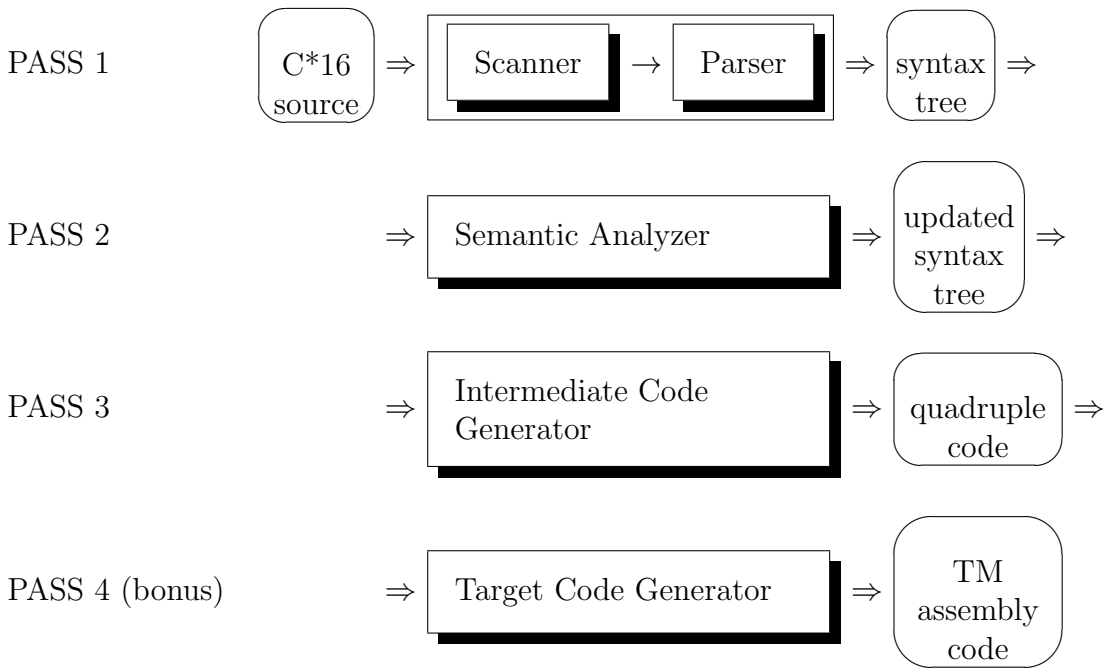

Figure 1. Phases and passes of the compiler in your project. Shadowed boxes represent the compiler phases, oval boxes program representations.

4 Project phases and marking

The compiler is viewed as an integral software product. Relatively little credit will be given for its working parts unless the project as a whole is brought to a satisfactory completion.

However, in order to maintain the necessary pace and provide feedback, the project

will be submitted in four phases, plus the submission of the final project documentation, according to the time-table given in the course outline.

The project will be graded as a whole after its final documentation is submitted. However, 20% of the grade for each phase will be based on the timeliness and quality of the phase submission.

After a phase, such as the scanner, is submitted, the work on it will not be considered frozen. On the contrary, the students will be encouraged to fix all problems and perfect the phase as the project continues, so that other phases can proceed normally. Additional assistance may be provided to facilitate this process. However, the grade structure indicated above will favor those projects whose phases were fully completed at their prescribed submission times.

Note that each phase submission is an incremental step towards the development of a full compiler, and includes the code for all previous stages.

5 Testing

While one tries to produce error-free code in every software project, this requirement is particularly critical in compiler development, since an error in a compiler may affect all software developed with it. The basic requirement is that you assure yourself of the correctness of your code as you write it. However, a full set of suitable, specifically constructed, test cases must also be developed for each phase.

The test cases must ensure that every statement in the compiler code is executed at least once during testing, and also that every entry of any constant table (such as the table of reserved words) is used at least once. Whenever the compiler can be exhaustively tested for a set of legal inputs (e.g, the set of all legal characters), the test suite should provide for that; if not, the endpoints of ranges and other potentially critical values must be covered by the tests.

The test programs must be submitted along with each phase, as well as the test scripts that illustrate their application and display the resulting output. The tests must be submitted in electronic form, so that they can be independently executed as necessary during the grading process. You should be aware that your compiler phases may be subject to additional tests during grading.

6 Test program sharing

Every group is responsible for the testing of their compiler project, including the development of a suitable set of test programs in C*16 . However, sharing of test programs is legal if three conditions are satisfied:

1. The author's name, date, and brief statement of the test's objective must be included

in the test program's header. This information must be preserved by anyone using the test program or its derivatives. (If the program is modified by others, all relevant names and dates must be added to the header.)

2. A test program offered for sharing must be offered to *all teams*. We can do this by posting test programs on the course website. The intent is to have everyone in class working under the same conditions.

3. Anyone using tests developed by other teams must acknowledge that in the introduction to the project report.

Under the above terms, we encourage the sharing of tests as it provides additional challenge and improves the general quality of work.

7 Submissions

The submissions are performed electronically, by e-mailing your work to **cpsc425@unbc.ca**. The submission must be a tar file that produces a directory named `<last-names>-<phase>`, e.g., `lastname1-lastname2-scanner`. The subject line of the e-mail must contain the same name prefixed by "CPSC 425", e.g., `CPSC 425 lastname1-lastname2-scanner`.

Each phase submission will contain, as separate files in the directory:

1. A phase report text in pdf;

2. The (fully documented) source code (including the up-to-date versions of the source code from all previous phases);

3. The object code executable under Ubuntu Linux (in the UNBC CS lab environment);

4. A subdirectory containing test files and outputs.

8 Report structure

The report must contain the following sections:

1. *Introduction*, explaining what is submitted;

2. *Participation*, explaining what each group member contributed;

3. *Project Status*, explaining if everything has been fully completed and tested, and whether the current phase has required any modifications of previous phases;

4. *Architecture and Design*, explaining the approach taken, the software structure, and any design aspects of interest;

5. *Implementation*, documenting any functionality or size limitations of your implementation;

6. *Building and Use*, explaining how to build executable code, and how to run it, including command-line arguments and options;

7. *Code*, including the list of source files, with a brief description of their content; the source files should be submitted separately, within the same package.

8. *Tests and Observations*, explaining the approach to testing, the test cases, results, and observations; the test programs and scripts should be submitted separately, within the same submission package.

The report must document any *amendments to previous phases* made since the last report.

9 Literature

The background theory for the project will be covered in class and in the posted guidelines and lecture notes.

The recommended text, by Aho, Lam, Sethi, and Ullman, is the standard compiler textbook and can be used as additional reading and as reference text for clarifications.