EXPL

EXPERIMENTAL LANGUAGE

WRITE YOUR OWN COMPILER!

SOURCE CODE
The entire project code is hosted on Github
http://github.com/silcnitc

Trial version, errors are being corrected as reported.

This project aims to develop an online self-sufficient educational platform to help undergraduate Computer Science students understand the functioning of a compiler for a simple procedural language by writing a small toy compiler themselves. (An object oriented extension for the language has been added subsequently.) The project will provide students with a roadmap for the development process and guide them along the roadmap with supporting documentation. Using the step-by-step guidance offered by the roadmap, the students will be able to build the compiler under minimal expert supervision.

The platform is designed assuming that the student has a basic knowledge of Data structures, Computer organization and a working proficiency of the C programming language. Being instructional in nature, this project tries to give some insight into the working of LEX, YACC and the usage of these tools to develop a compiler for an instructional language custom designed for the project - called ExpL (Experimental language). Brief theoretical explanations are provided on a *need-to-do* basis.

If you wish to get started with the project directly, proceed to the roadmap. If you wish to get a high level overview of the project, a brief outline is given below.

Overview of the project

To design a compiler, one needs the following specifications:

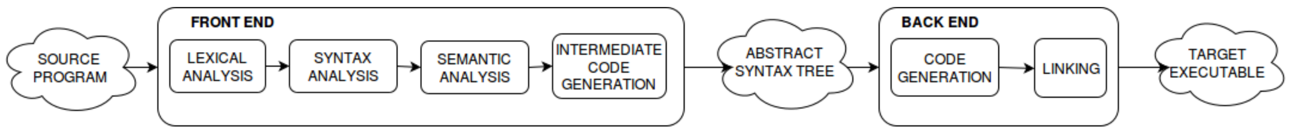
1. *The specification of the source programming language*: The source language for this compiler is called the Experimental Language (ExpL), which is designed solely for instructional purposes. An informal specification of the source language is given here. The specification for an extension of the basic ExpL language with support for object oriented programming is given here.

2. *Application Binary Interface of the target platform*: A compiler must generate a binary executable file that can be loaded and executed by an operating system running on a target machine. Normally, an OS system installation comes with an interface specification, explaining how binary executable files must be formatted to run on the system. This interface is called the Application Binary Interface (ABI). The ABI is dependent on both the OS and the machine architecture. The ABI for this project is that of the Experimental Operating System (ExpOS) running on the Experimental String Machine. [XSM]. The ABI specification is given here. (The simulator allows the target program to contain machine instructions in mnemonic form, avoiding translation to binary format).

A compiler takes a source language program as input and produces an executable target file formatted according to the requirements laid down by the ABI. Of course, the semantics of the program must be preserved by the translation process.

The main intellectual complexity in understanding a compiler is that its inputs and outputs are programs – that is, a compiler's input data is a (source language) program and its output is a (target language) program. The task is to systematically map each construct in the source program to semantically equivalent constructs in the target language.

A text book approach to compilation logically divides the process into several parts. The lexical elements of the input program are identified during a *lexical analysis phase*. The software tool LEX is used in this phase. The lexemes identified by the lexical analysis phase are passed to the next phase – called the *syntax analysis phase* - which checks the program against syntax errors. If the program is free of syntax errors, the next conceptual stage called the *semantic analysis phase*, checks for type and scope errors in the program. Once a program is found to be free of syntax and semantic errors, an intermediate representation of the source program called the *abstract syntax tree* representation is generated by the compiler. (In practice, syntax analysis, semantic analysis and abstract syntax tree construction happens together. The syntax directed translation scheme provided by the software tool YACC is used for completing these phases.) The sequence of phases starting with the lexical analysis of the source program to generation of abstract syntax tree representation is called the *front end* of the compiler.



The abstract syntax tree produced by the front end is the input to the back end phase which generates an assembly language program. Finally, a label translation phase is run to replace symbolic labels in the assembly language program with logical addresses (linking) and generate final target executable file.

The road-map takes you through a "spiral" model of program development. First, a simple expression compiler is built. Static variables are then introduced to the expression compiler and control flow constructs are added. In the next step, support for string type is added to the language and semantic issues are addressed. Subroutines and run time allocation are introduced subsequently and finally support for user defined types and dynamic memory allocation is added. (A final phase of providing support for classes and subtype polymorphism has been added subsequently). In each of these steps, you will encounter lexical, syntax and semantic analysis, syntax tree construction and code generation. The tasks will be very simple during the initial stages and the complexity will grow gradually as more and more features are added to the language.

Wish you good luck working through the road-map.

Github Home | About