CS 4300 - Compiler Theory - Fall 2014

Project Overview

INTRODUCTION

Your term project will be to implement the front-end components of a compiler for a small subset of C++. The subset is minimal, but has been carefully selected to cover most of the important compiler design issues.

It is likely your compiler will need to be written in the C++ language, since we will be using compiler construction tools, which generate C or C++ source code. The final project must run (and will be tested on) a Mac like the ones in the CS lab.

SOURCE PROGRAM SYNTAX

An LALR(1) grammar for most of the subset of C++ you are to implement accompanies this document. You may modify this grammar to correct or add language features as the term progresses.

SEMANTICS

As you implement your compiler, you may have questions about how various language features are supposed to behave or interact. Bring up such questions in class, and we will resolve them. For a start:

1. Parameters to functions may be of type int, float, or array, but they are always passed by reference. Note that this is a variation from standard C++! It is legal to use a constant as an actual parameter but it is a logical program error (i.e. not the responsibility of the compiler) if this constant is modified inside the function.
2. The parameters to main() are ignored, except to check that they are syntactically legal identifier names.
3. All Boolean expressions should be implemented using "short circuit" evaluation, like in standard C++.
4. Real and integer values may be mixed in arithmetic expressions, and the integers should be automatically converted to reals when necessary for type-compatibility. Similarly, an integer expression may be assigned to a real variable. If a real expression is assigned to an integer variable, the value is truncated immediately before the assignment.

IMPLEMENTATION

You will translate the source program into quadruples, which can then be interpreted by the quadruple interpreter available from the course website. The format of these quadruples will be described in detail in a later handout.

Design your compiler to read the source file name from the command line and write the quadruples to an output file with a name derived from the source file name (e.g. source.c -> source.q) . Your design should involve the following steps. For each step, a target "week of completion" is given in parentheses; note that in order to meet these dates you will need to be working on several projects concurrently. Here, "completion" means a working, debugged version – due to interactions with other pieces, you may need to continue to work on "completed" pieces to add functionality later.

1. (1-2) Write a driver program and a skeletal symbol table class. The driver should interpret commands from cin lookup <variable-name> and quit. The lookup command adds the given <variable-name> to the symboltable if it is not already present, and prints the address in hexadecimal where the <variable-name> was found or stored.

   The symbol table class which this driver uses contains a list of strings (or better: multiple lists, chosen by hashing – see item 5), and just one method, called *lookup*, with signature string *lookup(string *lexeme) which looks to see if the lexeme is already in the table, adds it if not, and returns the address of the list member it found or added.

2. (1-2) Design and write the error-handling routines. When an error is discovered anywhere in the compiler, the error handler should be called with a severity level indicator and a textual error message. The error handler will print line number and error diagnostics and keep track of the maximum severity encountered in the compilation.

   Severity levels should include *warning*, *error*, and *fatal*; any errors of severity *error* will prevent producing an output file, and *fatal* errors will terminate the compiler immediately.

3. (1-2) Write a skeletal parser, which contains the syntax rules for the language and compiles successfully when run through bison. You will not yet be able to test the compiled code, until the lexical analyzer is working.

4. (1-2) Write skeletal initialization code, which performs any initialization required by your symbol table package, locates and opens the source file, derives the object file name, etc. Use the old C I/O functions (fopen() etc.) in <cstdio>, since *flex* will require the input to be a FILE *.

5. (2-3) Improve (1) to a true symbol-table class, organized via hashing to linked lists (see the text beginning on page 434). Each symbol table record contains the name of the symbol and a kind field, which differentiates the different kinds of things in the table: keywords (different value for each keyword), variable and names, numeric constants, etc. Leave the rest of the record structure flexible for now; you will add things later. Improve (4) to insert all C++ keywords into the table, along with a unique kind for

each (Make these #define symbols, like #define WHILE 7, because the numeric values will eventually be generated automatically by bison.). Maintain an array of linked lists of a size given by a manifest constant, and a hashing algorithm to spread names among the lists as uniformly as possible. As in (1), provide a method to search the symbol table for a given name, create a new entry for that name if none is present, and in either case return a pointer to the record for that name.

6. (2-3) Write the lexical analyzer, using *flex*. Count lines for use by the error handler. Design your lexical analyzer to be a function called by the parser, returning a token and, when necessary, an attribute value. Errors detected by your lexical analyzer should be routed through the error-handling routines described above.

7. (3) Integrate the symbol table with the lexical analyzer. Have the lexical analyzer lookup all numbers and symbols, insert new ones, and return appropriate tokens for identifiers, numbers, and keywords.

8. (4) Fill out the parser to a version that works with the lexical analyzer. This will be the major component of the compiler. It will be written incrementally; this target completion date refers to a version that does nothing but check syntax and and recognize correct programs. This initial version needs to be integrated with the lexical analyzer to recognize tokens, but does not yet do any semantics.

9. (5) Augment the parser, symbol table, and lexical analyzer to recognize block structure and scope. This involves implementing semantic routines for declaration statements, including any function definitions.

10. (5+) Augment the parser with semantic routines to generate the quadruples. In addition to the changes described above, the grammar will need to be modified in places to make the translation easier, as we will discuss class. Do semantic analysis such as type checking at this time, converting integers to reals when necessary.

    Begin with the semantics of simple expressions, then add control statements, function linkage, and subscripting.

11. (Ongoing) Test programs. The example program from below can serve as a simple test routine. You will have to design and use many more test programs; develop a suite of tests that exercise all of the features of the compiler.

12. (Date TBD) Turn in the completed compiler; we will test it against test programs ...

PLANNING AND MEASURING PROGRESS

Each team will meet with the instructor for Progress Meetings regularly (every week or two). At each such meeting the team should present a short written report, in which you focus on objective measures of progress: verifiable milestones like "the lexical analyzer compiles with no error messages" or "the compiler produces correct quadruples for all of the expressions in our test suite." Try to avoid subjective measures like "the syntax analyzer is 90% complete" or "we made a lot of progress on debugging the symbol table." Organize it as follows:

Heading: Date, and names of all team members

Short term goals

- Work completed since the last report
- Work scheduled for completion within the next week

Medium to long term goals

- Work in progress not likely to be completed within a week, with estimated completion date

Questions for the instructor that should be addressed during the meeting

Other concerns

Appendix: A collection of one-paragraph reports, one from each team member, wherein each member describes what they have been working on and what they have accomplished on the project in the last week, and what they will be working on in the coming week. (Operationally, this information should be collected first, and used as source material for the body of the report.)