Overview

You must work as a member of a pair or a triple. Each group has to send an e-mail (containing the names of the members) to cc07-tutor@gigasun.cs.uni-sb.de until Monday (2007/11/19) 4pm.

In the course project, you have to implement a compiler for a subset of C, where the CMa is the target platform.

You have to use C or C++ for the project. The scanner has to be generated by lex/flex, the parser has to be generated by yacc/bison. From the course homepage you can download a skeleton for both, which provides a good starting point for the project.

Like for any other project, you have to give us the full source code for the review and a suitable makefile. For this reason, you’ll be given an SVN account at our chair. Your project should be compilable by simply typing make all (or nmake all, gmake all etc.), possibly with an automatic configure script before, without requiring additional user intervention.

The project is split into two parts. In the first part, you have to implement the lexical, syntactical and semantical analysis for the required subset of C. The deadline for the first part is 2007/12/10. In the second part, you have to realise the code generation according to the requirements below. The deadline for the second part is 2008/02/11.

Although you may change the implementation of the frontend (i.e., the lexical, syntactical and semantical analysis) during the second half of the project (e.g., to implement additional stuff), each subproject is graded individually (this means, changes to the frontend will not retroactively raise the grade of the first part). We will provide you with test cases to validate your compiler. If you meet the requirements, you will get a 1.3. Implementing some of the extensions will take you to the 1.0, and even allow you to cover potential minor problems in the compelled part.
Requirements

The C Preprocessor

Either your program understands comments (/* */ as well as //), or you rely on an external preprocessor. In the first case, you do not need to implement the C preprocessor directives such as #define or #include. In the latter case, your program should understand the line directives produced by the external preprocessor, because otherwise the user gets confused.

Declarations

Your compiler has to work with the following kinds of declarations:

- global variables
- local variables at the beginning of a function
- function parameters
- function prototypes (i.e., forward declarations)
- basic function return types
- typedef type declarations
- structs, enums and arrays
- pointers and associated C stuff (except for function pointers)

It is allowed to require a strict sequence of type declarations, variable declarations and function declarations.

You need not to implement:

- function pointers
- explicit initialization
- old-style declarations
- variable argument lists
- arbitrary function return types
- labels
- union
- const
• **static local variables**\(^1\)
• **inline**
• variable declarations mingled with statements
• global declarations in arbitrary order
• nested blocks (including variable declarations)

**Expressions**

You compiler has to work with the following kinds of expressions:

• (decimal) integer and floating point constants
• variable names with scoping
• standard binary/unary expressions, sequences and assignments, including +=-like assignments
• bitwise operators
• automatic conversion of floating point/integer expressions
• function calls
• prefix/postfix operators (\(--i, o++\))
• component selectors
• literal string constants as parameters of output statements

You need not to implement:

• explicit casts (but see below)
• general strings or characters `char`
• hexadecimal or octal constants

\(^1\)Static global variables make no sense in our setting, as there is no linking involved.
Statements

Your compiler has to work with the following kinds of statements:

- loops, statement sequences, conditionals, expressions used as statements
- break and continue in loops
- switch statements
- in-/output (see below)

You need not to implement:

- inline assembler (e.g., __cma__("loadc 2");)
- general switch statements (gaps, cases not being terminated by break;)
- goto

General Remarks

If you, at first, do not implement general pointers, treat arrays like first-class data structures; that is, an array of 5 ints is a specific type, variables of which are passed by value. In this case you might want to not allow direct array declarations (e.g., int t[6][3];) but use only primitive (predefined or declared) names in variable declarations: In this case, C++-style call-by-reference declarations are useful.

In-/output need not be implemented with standard C functions; instead you should implement the following functions:

- int readi () to read an integer from user input.
- float readf () to read a float from user input.
- void reads (char *str, int size) to read a string comprising at most size characters from user input into str.
- void printi (int i) to print an integer i.
- void printf (float f) to print a float f.
- void prints (char *str) to print a zero-terminated string str.

You need not to use the code function from the lecture for code generation. Apart from optimizations, an equivalent if-else-cascade as an implementation of a switch is fine, for example.
Implementation Hints

General Remarks

Input and output, as well as heap allocation/deallocation can be done in two ways. On the one hand, one can introduce special keywords in the language such as malloc/free or printi etc., which are then translated into appropriate CMa code sequences. On the other hand, prototypes of functions like this can be supplied in header files and treated like external functions by the compiler during compilation of the main code, after which the compiler includes the CMa code for the functions with the main code in the output file. Although the second approach seems more natural, the first one is actually more useful: A real implementation of malloc requires casting, but these features are basically useless in all of the remaining C input to the compiler. If you use a C-like malloc, you have, of course, to implement sizeof.

For the grammar of expressions and statements, the lcc book is quite helpful. A complete C grammar is printed, for example, in the appendix of Kernighan/Ritchie.

There are basically two ways to approach a project like this. You might start with scanner/parser/code generator for a simple subset (expressions on integers, for example) and work your way up from there, creating and modifying the auxiliary code (symbol table manipulation, code emission, type checking, syntax tree handling, optimization) as you go.

Or you first implement a large skeleton of auxiliary functions which then are used largely as they are during the project. Although the first approach leads to quicker results, the second approach is easier in the long run. Pay particular attention to the table of declared symbols (types and variables): Quick hacks work at first, but get you into trouble as soon as you allow for more complete declarations.

Remember that you can do much work during parsing, using the symbol attributes.

General rule: Implement good data structures for the symbols and the code constituents. If that is properly in place, the actual algorithms to check the C code and generate CMa code are mostly harmless.

Reduction

If you wish to skip some parts of the language, we suggest that you first do away with all non-primitive types and treat functions somehow specially.

Expansion

Nested blocks, explicit initialization, and declarations in arbitrary order are about the only useful extensions you might want to implement. Furthermore, the lecture will point out some easy optimizations (i.e., optimizations that can be performed without a firm background in program analysis or an analysis framework).

You may chose to implement additional features of C (that is, stuff not being part of the “need not to implement”-lists), but this will not affect your grades.

---

2Note that there is no separate linking stage, and it would be mostly useless to introduce one.

3In this case, you will probably have to rewrite the symbol handling code if you choose to implement pointers, structs etc. later on.

4For our target, casts only make sense for malloc/free, which can be coded as special keywords instead.