UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 264
Fall 1995

S. L. Graham
A. B. Downey

Assignment 1: "Cadaver" Compiler

Due: Monday, 11 September 1995

As we discussed in class, instead of writing an incomplete compiler during the course of the semester, you will read and evaluate a more complete and widely-used compiler of your choice. The purpose of this assignment is for you to choose your cs264 "cadaver" and to begin to learn about it.

1 Find a Compiler

The kinds of questions you will be asked to consider during the semester might be

- What is the high-level intermediate representation used by the compiler?
- How robust is the compiler in the face of errors? How good are the diagnostics?
- How faithful is the compiler to the source language definition?
- How good is the code quality with respect to optimization? Register usage?
- Is there a systematic approach to type checking; if so, what is it?
- What would be needed to add some new language feature (e.g. a control construct)?

Thus, you should look for a compiler with the following attributes:

- The source code is available.
- Documentation is available, either separately or embedded in the code.
- Someone might be available to answer questions (either because the compiler is widely used, e.g. gcc, or because someone is listed as a contact person).
- The compiler generates code, or at least low-level pseudo-code (i.e. not an interpreter or just a front end).
- You can execute the compiler (just reading code is easy to do remotely, but execution might require either a guest account, downloading binaries, or a local installation).

- It's possible for you to find out enough about the source language to be able to write some small programs.

The first place to look is in the software warehouse, since you might prefer to have something local. Check /usr/sww, /usr/sww/lang, /usr/sww/share/lang as a start, although some of that is not very current. Note that there may be some compilers available on the Berkeley campus under a site license that are not otherwise available on the WWW.

Or look at http://remarque.berkeley.edu/~muir/free-compilers

Use the class newsgroup to share what you learn with others, so that everyone doesn't follow the same blind alley or make the same choice.

2 Write A Brief Summary of the Attributes of Your Selection

Among the information you should provide is

1. What compiler you've chosen and where it's from
2. Why you chose this compiler
3. A summary of how well the compiler satisfies the criteria (source code availability, etc.) and what difficulties you anticipate you may have in using your chosen system for the course.

3 Evaluate the Lexical Analysis Done by Your Chosen Compiler

Study the way lexical analysis is done and evaluate it with respect to the issues discussed in class. Following are some examples of issues that would be interesting in evaluating the analysis. You should add any others that you discover or that you think are relevant.

- Does the analyzer use some sort of finite state automata, regular expression, etc. specification? Is there any formal specification of the tokens in the documentation or language definition?
- Are there non-finite state aspects of the lexical analysis? If so, what are they? Are they required to handle this language or are they used for the convenience of the compiler writer (e.g. for efficiency)? Explain.
- Is there a clean interface between scanner and parser? Between scanner and semantics? If so, how can you tell? If not, why not?
- Is there a sequential interface between the scanner and other phases (i.e. could the scanner be written with minimal change to cause all scanning to be completed and the token stream stored before beginning any other phase)
- What would be involved in changing the lexical definition of the language? How hard would it be? (Examples might be putting keywords in other natural languages besides English, or changing operator symbols).

4 Get some experience using Flex

A lexical language is described subsequently. It is designed solely as a scanner exercise – it is not a thing of beauty. Two characters of lookahead are necessary to recognize the tokens described below.

Write a scanner that translates a source program consistent with the syntax described below into a sequence of lexical units (tokens) and associated lexical values (if appropriate). Write the scanner using the **flex** scanner generator. The textbook contains a pretty good description of the predecessor tool LEX. The **flex** manual is available on-line.

The scanner should emit a token ERROR with a lexical value which is the sequence of characters in error. Minimal error recovery is to find the nearest character at which to continue scanning. You may add more if you wish. Evaluate the correctness and robustness of your scanner by providing as input pseudo-programs constructed using the lexical structure we have described. You may share test programs with other class members by posting them to the news group. (Be sure the comments explain the intended outcome). Also run some real program text (e.g. from C) through your scanner to test robustness.

Keep track of how much time you spend writing and debugging your scanner. Your write-up should discuss your experience (what the difficulties were and how you solved them, how long it took, whether you had used Flex before), as well as your solution and testing. I would also like some input/output examples of your scanner in action. In other words, convince me that you did the assignment and maybe that you learned something.

Lexical Structure of the Source Language

The source language is composed of four types of lexical units: identifiers, keywords, literals, and special symbols.

An identifier is a sequence of letters, digits, and the special character '_' (underscore). Identifiers must begin with a letter. Two or more '_'s may not appear together. Identifiers are case sensitive (e.g., *i* and *I* are different identifiers), and must be distinct from the keywords of the language. They will be represented in grammar rules by the generic token IDENT. Identifiers may be of arbitrary length, so the implementation should not impose length limits.

There is a list of keywords in ~cs264/assignments/assn1/keywords.

A keyword is a sequence of letters. Your scanner should be designed so that modifying the keyword list is easy. A keyword can be all upper or all lower case, but not mixed. Thus, *begin* and *BEGIN* are the same keyword, but *Begin* is just an identifier.

Literals are the primitive values manipulated by the language. There are five built-in primitive types: integer, float, character, text, and boolean, represented, respectively, by the generic tokens INTLIT, FLOATLIT, CHARLIT, TEXTLIT, and BOOLLIT. Integer literals are sequences of digits. Float literals are of the form *n.m*, *n.mEk*, or *n.mE+k*, where *n*, *m*, and *k* are integer constants (i.e., sequences of digits). Note that *.m*, *n.*, *nEk*, and *nE+k* are not legal float literals. The letter *E* can also be the lower case *e*; the + sign can also be a -. Character literals are single characters in single quotes: 'a', '%', etc. Strings are sequences of characters in double quotes: "abcdefg", "12&^#%$", etc. In character and text literals, the backslash character is used as an escape (as in C): '\'' is a single quote character, '\"' is a double quote, and '\\' is a single backslash. Finally, a boolean literal is one of the reserved words *TRUE*, *true*, *FALSE*, or *false*.

Special symbols comprise the operators and delimiters of the language. For the purpose of this assignment, your scanner should recognize these symbols:

(	)	[	]	{	}
,	;	.	:	:=	..
^	+	-	*	/	#
<	<=	>	>=	=	

The lexical units of the source program may be separated by the formatting characters (blank, tab, end-of-line, form-feed) and by comments. At least one formatting character is needed between two lexical units when neither is a special symbol.

Comments are arbitrary character sequences enclosed by (* and *). Comments may be nested and may occur between any two lexical units. They may contain embedded formatting characters.