Programming Project 2

Due date: Tue 10/7 11:00 pm

Introduction

In this project you will write a program to resolve static scope references using the LeBlanc-Cook algorithm discussed in class. Your program will use the lineparser() that we provide to read a “program” in a simplified language as in the following example:

```c
def x
def y
{
def x
ref x
ref y
}
ref x
```

Your program will process each line in the input, updating the symbol table and the scope stack as necessary. We interpret each line as in static scope in C/C++: '{' starts a scope, '}' ends a scope, 'def' is a variable declaration, and 'ref' is a reference to a variable. Our input files are simple and include exactly one of these on each line. Your program will echo a version of the input, as well as tracking the change in scopes and resolving the references to variables, as illustrated by the following output:

```
STARTING SCOPE 0
def x
SCOPE 0 DEFINES x
def y
SCOPE 0 DEFINES y
{
STARTING SCOPE 1
def x
SCOPE 1 DEFINES x
ref x
REF TO x OF SCOPE 1
ref y
REF TO y OF SCOPE 0
}
EXIT SCOPE 1
ref x
REF TO x OF SCOPE 0
ENDINP
EXIT SCOPE 0
```

Notice that all scopes are open, and we have one global scope, numbered 0, which is accessible from all scopes in the file.

A Simple Line Parser

We are providing you with /comp/80/files/pp1/pp2seed.cpp which includes a simple line parser that reads one line at a time from the input and returns the input in processed form. The relevant part to repeat here is the following:
#define ENDINP 0
#define DEF 1
#define REF 2
#define C_OPEN 3
#define C_CLOSE 4

struct token {
    int type;
    string val;
};

token lineparser();

You can call lineparser() to get the next line in processed form. For example, the first call on the input above will return a token with type DEF and val of “x”. A call made, after the last line of code is read, returns a token of type ENDINP.

Your Task

Write a program to process “programs” as illustrated above. Your program must use the LeBlanc-Cook algorithm as described in class. In our case, as in class, all scopes are open so there is no need to accommodate closed scopes (or imported/exported entries). You may also simply push scope 0 on the scope stack and ignore the “pervasive” shortcut but can also use it as in the text. In order to complete this task you need to build two data structures as follows:

The first is the scope stack. This stack can simply keep the scope numbers accessible from the current point in the program. These need to be updated every time the program enters or exits a scope. Notice that implementing only the standard push() pop() operations is not sufficient since in the LeBlanc-Cook algorithm we also need to iterate over elements in the stack from top to bottom.

The second data structure is the hash table. Table entries include variable names and their scope number. Entries are hashed using the name only. The table should support inserting new entries, retrieving entries and resolving a variable reference using the LeBlanc-Cook algorithm.

The algorithm is described in the text. We give a brief description here. Given a variable name, the algorithm searches for this name in the hash table. For each entry found, it searches the scope stack from top to bottom to see if the entry is accessible. We do not always need to go all the way to the bottom of the stack: if we find the scope, we update our “best match”. Also if we hit our previous “best match” there is no point searching below it. Since the global scope (numbered 0) is accessible from all points in the program there is no need to search for it in the scope stack. This the the “pervasive” variable in the algorithm.

After building these you need to write the main program that reads the input (via lineparser()) and processes it. You should check for two kinds of errors. First, we cannot define the same variable more than once in the same scope. So, for example

def x
def x

should print an error message

ERROR: REDECLARING VARIABLE x IN THE SAME SCOPE

and exit. The last line printed should be in this format. You may print anything before this line.

The second, is checking that we do not close more scopes than we open. So, for example

def x
{
def x
}
}
should print an error message

ERROR: CLOSING NON-EXISTING SCOPE

and exit. The last line printed should be in this format. You may print anything before this line. Other
than these two errors you may assume that the input file is in the right format and legal.
You must write your own classes/data structures. No STL or other libraries please. You should package the
data structures in one or two classes and use header and implementation files for each class.

Files

Initial code including the line parser is given in pp2seed.cpp. Several example input/output files are also
provided. All files are available in the directory /comp/80/files/pp2/.

Submitting your program

Assuming your code is in files pp2.cpp LBC.h LBC.cpp in the current directory on our sun machines, you
should submit by typing:
provide comp80 pp2 pp2.cpp LBC.h LBC.cpp
You can modify file names and in fact you may submit more files (up to 7). But please make sure the g++
compiles your code when using all cpp files on the command line.