Department of Mathematics and Computer Science
University of Southern Denmark, Odense

January 20, 2012
KSL

Exam Project in Compiler Construction, part 1

Kim Skak Larsen

Spring 2012

Introduction

In this note, we describe one part of the exam project that must be solved in connection with the project ”A compiler for an imperative programming language”, Spring 2012. It is important to read through the entire project description before starting the work on the project; also the sections on requirements and how to turn in your solution.

Deadline

Wednesday, February 8, 2012, at 12:00 (noon).

A Symbol Table in C

Among other things, you must turn in a program which must be written in the programming language C. It must be the c99 ANSI standard as specified by the options below. This excludes C++, in particular. Your programs should be compiled using

```
gcc -std=c99 -Wall -Wextra -pedantic
```

In this assignment, you must construct an advanced form of a symbol table where data is stored in a collection of connected hash tables. See Fig. 1. The entire construction illustrates the symbol table and each box illustrates a hash table.

You may use the file `symbol.h`, which is available via the course home page, as a starting point for this assignment (see Fig. 2).

Your task is to implement the six functions listed last. The implementation should be placed in a new file `symbol.c`. In addition, you must write and include test examples, commenting on what is tested, what is expected, and what is observed.

The elements in the symbol table are strings, `name`, with an associated value field, `value`. When such an element is inserted into the symbol table, it is stored into one of the hash tables. This will be described in detail below.

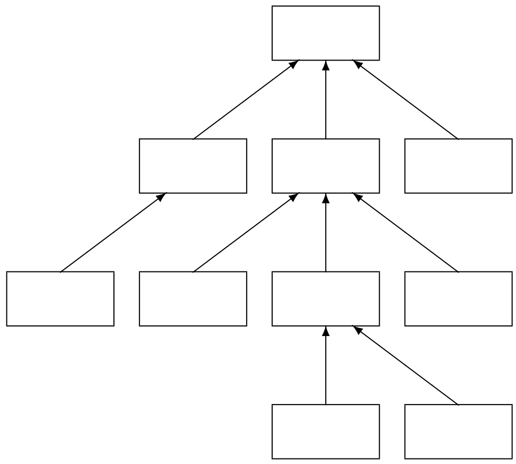

Figure 1: An example of connections between hash tables.

A pointer to a hash table can also be thought of as a pointer to (parts of) the symbol table which can be accessed through the pointer to the given hash table.

Conflicts during insertions into the hash tables are resolved using chaining. Thus, the entries in the hash table arrays are (possibly empty) linked lists of elements of the type `SYMBOL` (linked via `SYMBOL`'s `next` field). For each `name`, there is a value of type `SYMBOL` in which `name` is stored. To avoid any confusion, chaining is handled within each hash table and has nothing to do with the pointers seen in Fig. 1.

We now discuss the functionality of the six functions.

- `Hash` computes the hash values (see how below).
- `initSymbolTable` returns a pointer to a new initialized hash table (of type `SymbolTable`).
- `scopeSymbolTable` takes a pointer to a hash table `t` as argument and returns a new hash table with a pointer to `t` in its `next` field.
- `putSymbol` takes a hash table and a string, `name`, as arguments and inserts `name` into the hash table together with the associated value `value`. A pointer to the `SYMBOL` value which stores `name` is returned.
- `getSymbol` takes a hash table and a string `name` as arguments and searches for `name` in the following manner: First search for `name` in the hash table which is one of the arguments of the function call. If `name` is not there, continue the search in the next hash table. This process is repeatedly recursively. If `name` has not been found after the root of the tree (see Fig. 1) has been checked, the result `NULL` is returned. If `name` is found, return a pointer to the `SYMBOL` value in which `name` is stored.

```
#define HashSize 317
#define NEW(type) (type *)malloc(sizeof(type))
void *malloc(unsigned n);

/* SYMBOL will be extended later.
   Function calls will take more parameters later.
*/

typedef struct SYMBOL {
  char *name;
  int value;
  struct SYMBOL *next;
} SYMBOL;

typedef struct SymbolTable {
    SYMBOL *table[HashSize];
    struct SymbolTable *next;
} SymbolTable;

int Hash(char *str);

SymbolTable *initSymbolTable();

SymbolTable *scopeSymbolTable(SymbolTable *t);

SYMBOL *putSymbol(SymbolTable *t, char *name, int value);

SYMBOL *getSymbol(SymbolTable *t, char *name);

void dumpSymbolTable(SymbolTable *t);
```

Figure 2: The file `symbol.h`.

- `dumpSymbolTable` takes a pointer to a hash table `t` as argument and prints all the (`name`, `value`) pairs that are found in the hash tables from `t` up to the root. Hash tables are printed one at a time. The printing should be formatted in a nice way and is intended to be used for debugging (of other parts of the compiler).

The tests should, among other things, demonstrate that when a search for a given `name` is carried out, the one closest to the argument hash table is found. A given string, `name`, can be stored in many of the hash tables, but each hash table is only allowed to store a given `name` once. In the testing, the `value` field can be used to show which `name` value is found.

One of the tests must build a structure corresponding to the one illustrated in Fig. 1.

Computation of Hash Values

It is well known that for efficiency it is important that the entries inserted into a hash table are spread out fairly evenly over the hash table such that most of the linked lists end up relatively short.

Experience shows that the following works well: One considers the ASCII values of each character. Thus, each character is considered an integer; or bit string. The characters are treated one at a time. Each treatment of a character results in an adjustment of a partial result which is zero initially. To be precise, for each character, the partial result is shifted one position to the left and then the ASCII value of the character in question is added to the partial result.

Consider the example in Fig. 3 where this has been done for the string `kitty`.

```
k = 107 = 0000000001101011
shift     0000000011010110
i = 105 = 0000000001101001
sum       0000000100111111
shift     0000001001111110
t = 116 = 0000000001110100
sum       0000001011110010
shift     0000010111100100
t = 116 = 0000000001110100
sum       0000011001011000
shift     0000110010110000
y = 121 = 0000000001111001
sum       0000110100101001
=         3369
```

Figure 3: Example computation of a hash value.

This is the method you must use in your implementation.

The Symbol Table in Your Compiler

The symbol table will later be used to store variable names, function names, etc. The value field will be used to store type information etc. Each hash table will be used to store the names within one function. The reason for the tree structure in Fig. 1 is that the language for which a compiler must be produced can have nested functions and the tree reflects this nesting structure.

The symbol table you implement in this assignment will likely have to be adjusted slightly to fit your concrete needs later.

Requirements

All material should be turned in on paper (referred to as *the report*) and electronically (a few exceptions are mentioned below). In addition, since this is an exam project, there are a number of important rules that will be detailed below.

Exam Rules

This is an exam project. Cooperation beyond what is explicitly permitted will be considered cheating and will be treated as such. You have a duty to keep your notes private and protect your files against reading and copying by others. Both parties involved in a possible plagiarism can be held responsible.

There will be given what we judge to be more than sufficient time for each assignment and you are strongly encouraged to plan your work such that you will finish some days before the deadline.

Assignments that are turned in after the deadline will not be accepted. Downtime on the system or the printers will not automatically result in an extension; not even if it is the last hours before the deadline. Neither will own or children's illness without a statement from your physician, etc.

Solutions

All specific requirements posed in the project description must of course be fulfilled.

The Report

The report should in the best possible manner account for the entire solution. Possible omissions, known errors, etc. should be described in the report. It is often a good idea to do this in a separate section instead of mixing it in with the rest of the report.

You must include the page at the end of this document as the front page of your report or attached in some way such that it is easily located. The report must be dated and signed by the members of the group.

For programs turned in as part of your solution, you must take care of the following:

The report must contain (possibly as an appendix) a printing of the entire program. This printing must be identical to the program that is turned in electronically. All the pages of your program print-out must contain your group number. One way of obtaining this is to use

```
a2ps -Pd3 --line-numbers=1 --tabsize=3 -g
  --header="Printed by group NN" file.c
```

where NN is your group number.

The report must contain a description of the most important and relevant decisions that have been made in the process of answering the assignment and reasons must be given where this is appropriate.

You must also explain how the program has been tested. Test examples and test runs can and should be included to the extent that this is meaningful (really large test files can just be turned in electronically).

Programs

Programs must be well-structured with appropriately chosen names and indentation and tested sufficiently. The numbers of characters (including blanks and 8 times the number of tabs) on a program line is limited to 79. This is important for various tools used for inspecting, evaluating, and viewing your programs, and it is important for the print-out of parts of your own program that you will see at the exam.

Programs will often be tested automatically. This makes it extremely important to respect all interface-like demands, e.g., input/output formats.

Programs that are turned in must compile and run on IMADA's machines. You are very welcome to develop your programs at home, but it is your responsibility. This includes technical problems at home, lack of access to relevant software, moving data to IMADA via e-mail, USB keys, etc. and converting to the correct format, e.g., between Windows and Linux.

Turning In

The report should be turned in at IMADA's secretaries' office. The office may be closed for very short periods of time. If, for some unexpected reason, the office must be closed for longer periods of time close to the deadline, an announcement will be made outside the office, giving instructions as to where you turn in your report.

For the first parts of the projects, you only need to turn in one copy of the report. For the final part, you must turn in two copies. For all parts, you must turn in all the material electronically.

Programs, test files, etc. should be turned in electronically. Your report should also be turned in electronically as a pdf file. As opposed to the paper version of your report, this version does not necessarily have to include programs and test files, since they are turned in separately. Also, signatures and the front page from the end of this document are not required in the pdf file.

The procedure for turning in electronically can be found via the project home page:

`http://www.imada.sdu.dk/~kslarsen/CC/Projekt/`

Avoid Danish (and other non-ascii) letters (such as æ, ø, and å) in your directory and file names (Blackboard does not handle this well).

You may upload your files individually or collect your files into one (archive) file before uploading. If you choose to do the latter, you must use either `tar` or `zip` for this.

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

Spring 2012
KSL

CC, Spring 2012
Exam Project, part 1

Group	

Date	

Name	
Birthday	
Logins	
Signature	

Name	
Birthday	
Logins	
Signature	

Name	
Birthday	
Logins	
Signature	

This report contains a total of .... pages.

Please write *very* clearly. Under Logins, give your IMADA followed by your student (student.sdu.dk) login.