Department of Mathematics and Computer Science
University of Southern Denmark, Odense
January 21, 2016
KSL

Exam Project in Compiler Construction, part 1

Kim Skak Larsen
Spring 2016

Introduction

In this note, we describe one part of the exam project that must be solved in connection with the compiler project, Spring 2016. 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

Tuesday, February 9, 2016, 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.

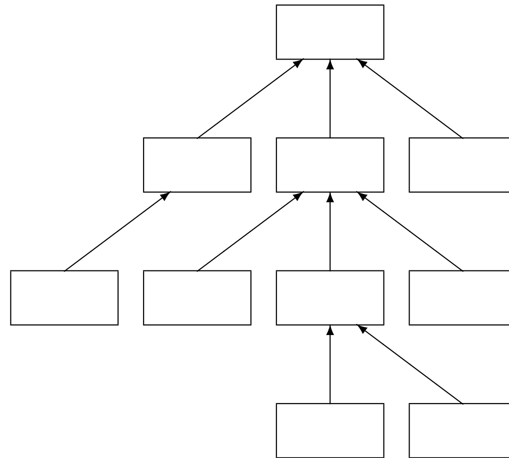

Figure 1: An example of connections between hash tables.

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))

/* 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.

For deciding on an entry in the hash table, you use the value, as computed in the above, modulo the size of the hash table. 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.

General Requirements and Rules

Here we list general requirement, procedures for turning in, and exam rules.

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 solving the project. Still, we strongly encourage you to plan your work such that you will finish some days before the deadline.

Solutions 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.

The solution

The solution consists of a program, test material, and a report. Thus, we use the term "report" to mean your description of the solution to the project without the program listing and listing of test examples and results (other than what may have been merged into the report as examples, etc.).

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, i.e., it must contain a description of the most important and relevant decisions that have been made in the process of developing the solution and reasons must be given where this is appropriate.

You must also explain how the program has been tested. Test examples or references to test examples and test runs can and should be included to the extent that this is meaningful.

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.

Programs

Files and directories should be named and organized logically. Programs must be well-structured with appropriately chosen names and indentation and tested sufficiently. The numbers of characters (including blanks and 4 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. In particular, they should 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 particular, no architecture-dependent option should be added, such as, for instance, `-m32` or `-m64`.

On the contrary, when you compile assembler code, then you *must* use `-m32`.

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.

Execution

This section on execution does not apply to part 1, but starts applying gradually through the project parts until it applies fully at the end. It is included in every project description, so you are not surprised at the end.

In the following, we list execution requirements regarding your compiler as well as the code your compiler produces. In most cases, this is just to conform to default standards or to choose one among alternatives:

- Your compiler (executable) must be called `compiler`.
- Behavior of your compiler:
  - Your compiler must read from `stdin`.
  - In the final part, only correct assembler code may be written to `stdout`.
  - If the compilation succeeds, the compiler must return zero.
  - If an error occurs during compilation, then
    - *nothing* should be written to `stdout`,
    - an error message should be written to `stderr`, and
    - a value different from zero must be returned.
  - It is recommended that the beginning of each phase of the compilation is announced on `stderr`.
- Behavior of the code your compiler produces:
  - Only **write** statements may write to `stdout` and it should write its integer or boolean argument followed by a newline.
  - If no error occurs, the code must return zero.
  - If an error occurs (that you catch), the code must return a value different from zero. If you write an error message, it must go to `stderr`.

Turning In

You must turn in on paper *and* electronically. The details are given below. All material that is turned in both on paper and electronically must be identical.

On Paper

You must turn in your

- report,
- a complete program listing,
- representative tests,
- the official front page.

You may omit very large test files and results and only turn these in electronically. The official front page that you find at the end of this document must be filled in, dated, and signed by the members of the group.

One reasonable way of producing your program listing is to print all your programs using the following (all on one line):

```
a2ps -Pd3 --line-numbers=1 --tabsize=4 -g
  --header="Printed by group NN" file.c
```

where `NN` is your group number. However, there are also ways to include your program listings as an appendix in your (LaTeX) report; see the CC home page.

Procedure for turning in on paper: The material on paper 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.

Electronically

Electronically, you must turn in

- the report as `report.pdf`,
- all relevant program and test files,
- a makefile, connecting the program files,
- the compiler as `compiler`, which should be an executable file.

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

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

However, it might be good to know already now that you should avoid Danish (and other non-ascii) characters (such as æ, ø, and å) in your directory and file names (Blackboard does not handle this well). To be safe, also avoid other special characters not normally occurring in file names.

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

Department of Mathematics and Computer Science | Spring 2016
University of Southern Denmark, Odense | KSL

CC, Spring 2016
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 student (`student.sdu.dk`) login. If you have an IMADA login that is different from your student login, give that in parenthesis.