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

February 26, 2012
KSL

Exam Project in Compiler Construction, part 4

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

Compiler	Wednesday, May 16, 2012 at 12:00 (noon)
Report	Thursday, May 24, 2012 at 12:00 (noon)

A Hungry Compiler

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 -m32
```

The primary new task of this part of the project is code generation, including optimization. These phases must then be combined with the front-end produced in part 3 of the project to form a complete compiler. The report must treat all the issues raised in the four project parts. The requirements for the compiler and report consist of all the requirements from the four project parts.

The report must be structured logically as one document, i.e., it cannot just be the reports from the various project parts with a “rubber band” around. A report draft is available via the course home page.[1]

Note that the deadlines do not imply that you can write a satisfactory report in one week.

[1]The draft is in Danish since this would be what almost all students would want. Please contact the lecturer if you have a translation problem.

Code Generation

Code generation must be handled in at least two subphases. In the following, these two phases are described, but more can be added in between the two.

The first phase generates abstract assembler code, which could be Pentium code, but which could be somewhat or significantly different. Some possibly differences could be that jump addresses are pointers to the linked list storing the (abstract) instructions. Another possibility is that temporary variables are used instead of explicit references to either stack or registers.

In the last phase, you must generate Intel Pentium Assembler AT&T style from the more or less abstract assembler code. You are allowed to use `printf` statements in your assembler code, as it has been done in the examples on the course home page. You are not allowed to use other functions from the C-library without explicit permission.

In between the two phases, you may place a number of optimization phases. If you have used temporary variables in your abstract assembler code, then such a phase could determine which temporary variables are placed in registers and which are placed on the stack.

Independent of the choice of abstract assembler code, peep-hole optimization is an obvious possibility for an optimization phase.

Execution Requirements

The makefile you turn in must be able to generate the complete compiler as an executable file using only your flex, bison, and C source files. It must of course be able to run on the department's computers, i.e., the ones in the terminal room. Example computers are `logon`⟨digit⟩`.imada.sdu.dk`, where ⟨digit⟩ can be any of the digits $1, \ldots, 9$. These are the computers you can connect to using ssh from outside IMADA.

The compiler (the executable) must be called `hungry` (all lower case) and it must (even though it can be generated by the makefile) already be generated in the directory you are turning in.

If you write `hungry < filename.hun` on command line and the HUNGRY program `filename.hun` is correct, then your compiler must produce Intel Pentium Assembler code AT&T style for the program. This must be sent to `stdout`. If the HUNGRY program contains errors, absolutely nothing should be written to `stdout`. Instead, an error message must be written to `stderr`. In either case, every time a new phase is started, the name of the phase should be written to `stderr`.

The compiler (`main` in the C program) must return 0 if the compilation is successful and another integer otherwise (1 if you do not have a reason to choose something else).

If, after a successful compilation, the output from your compiler has been placed on a file `filename.s` and you write `gcc filename.s` on command line, then a file `a.out` must be generated which executes with the correct result on the department's

computers. This requires that you strictly respect the requirements for `write` which must write its argument followed by a newline without any extra spaces or other characters to `stdout`. The only output that is permitted on `stdout` is what is written using `write`. If you would like to write nice error messages in case of a runtime error, you must write them to `stderr`.

The generated assembler code must return 0, i.e., as the last code you generate, 0 must be placed in %eax, followed by code to return to the operating system according to the conventions for this. There is one exception from this: If, at runtime, you catch one of the errors described under the heading "Runtime Safety Improvements", the assembler program should return the value indicated there.

Testing

The first code generation phase should be tested through a C function printing the more or less abstract assembler code to a file such that you can verify that the code produced is what you expect. This work can to a large extent be reused in the last phase of the code generation.

As the final testing, a sufficient collection of HUNGRY programs must be tested, and you must verify that the correct result is produced. This should be supplemented by well chosen internal tests of critical functionalities.

In the directory `/home/IMADA/courses/cc`, you will find a checking program, `check.py`. It is highly recommended that in addition to your own careful testing, you also test using this program, since this is the program which will be used by us in connection with an automatic testing of all compilers. In the beginning of this check program, you can see how to use it.

We emphasize that testing using only `check.py` is not considered a sufficient test of the compiler.

Extensions

A minimal core language, HUNGRY, has been chosen as the starting point. The purpose of only including the most necessary constructions in the language definition is to leave room for an individualization of the project by giving you the choice of which extensions to make. Thus, you are expected to add more features to your compiler.

In that context, there are the following requirements:

- You should not start work on extensions before having completed the basic work of implementing a compiler for the core language.
- It really should be extensions. You are not allowed to modify the core language. In particular, your compiler should be able to compile all the test programs.

- Any new facility should be motivated, described, and documented.

Below, we list some possibilities, but you are very welcome to introduce your own ideas. Some of the extensions are (much) harder than others. Your goal should be to implement at least (part of) one extension from each of the three collections: language extensions, runtime safety improvements, and advanced extensions. From the collection with advanced extensions, the peep-hole optimization is a task which is both interesting and can be limited to be quite manageable. Furthermore, it has the advantage that you can start with a simple version with few patterns and then gradually include more.

If you spend time considering extensions, but do not manage to complete the implementation, give a short account of your considerations and the status of your work implementing it.

Language Extensions

- Unary minus (-42 instead of $0 - 42$, for instance).
- Multi-dimensional arrays (this is different from arrays of arrays; you must have a layout such that for instance the address of A[i,j,k] can be computed directly and not via three pointer/offset operations as one would naturally do using A[i][j][k]).
- Array and record constants.
- Increment/decrement and assignment short-hands.
- For loops.
- Print of strings; possibly extended to strings as a type with various string operators.
- An input facility (here `scanf` from the C library may be used).
- Coercion from one type to another.
- More flexible assignment compatibility (including transfer of parameters).
- Possibility for structural assignment of records and arrays (making a copy instead of a reference to the same object).
- Extended loop control. Allow for the use of the keywords **continue** and **break** in **while**-constructions. The keyword **continue** starts the execution of the nearest enclosing while-loop from the beginning whereas **break** terminates the execution of the nearest enclosing while-loop. As an example, the following code adds positive numbers from an array A, stopping when a zero is encountered:

```
i = −1;
sum = 0;
while i+1 < |A| {
    i = i + 1;
    if A[i] == 0 then break;
    if A[i] < 0 then continue;
    sum = sum + A[i];
}
```

Runtime Safety Improvements

- Run-time check for array index values (return value 2).
- Run-time check for division by zero (return value 3).
- Run-time check for positive argument for array allocation (return value 4).
- Run-time check for use of uninitialized variables, including indexing and dereferencing of null pointers (return value 5).
- Run-time check for out-of-memory (return value 6).

Advanced Extensions

- Peep-hole optimization.
- Introduction of a `free` command to free previously allocated array and record space. For this to be at all useful, your system should of course allow reuse of this space.
- Full (automatic) garbage collection of (unused) arrays and records.
- Advanced register allocation.
- Reuse of stack space for local variables and spilled temporaries not used simultaneously.
- Adding class definitions, class hierarchy, and objects to the language.

In addition to these three collections of extensions, where each group should make at least one from each, there are many further possibilities. For instance, the following:

Extra Extensions

- Constant folding.
- Algebraic simplification.

Turning In

Electronically, you must turn in

- all relevant files from this and previous parts of the project.
- a makefile, connecting all of the above.
- The compiler `hungry` as an executable file.

In addition, you must turn in two identical print-outs of your report and code files. There should be a reasonable and documented test of all phases. See also the standard requirements.

Evaluation

In order to pass, the compiler must work on a reasonable subset of HUNGRY. A compiler which does not generate working code for even the smallest and simplest HUNGRY programs will not be accepted.

Additionally, your compiler will be judged on structure, correctness, elegance, and extent.

The report should not be a textbook. Thus, in general you may assume what all participants in the course know. However, do keep the censor in mind and it is nice with a brief description of the setting in each section as a reference point for your own work.

Most importantly, the report should contain description and documentation for the most important choices made. A report is not good just because it is long! Think carefully about what to include and try to make it "to the point", but do not exclude interesting choices and considerations.

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 (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.

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 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. 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 4

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.