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

March 3, 2016
KSL

Exam Project in Compiler Construction, part 4

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

Compiler	Friday, May 13, 2016 at 12:00 (noon)
Report	Monday, May 23, 2016 at 12:00 (noon)

A Tony Compiler

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.

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

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.

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 TONY 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, TONY, 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.
- Structural equivalence of composite types.
- 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.
- Structural equivalence on records and arrays.

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.

Evaluation

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

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 -P⟨Printer name⟩ --line-numbers=1 --tabsize=4 -g
--header="Printed by group ⟨Group number⟩" ⟨Filename⟩.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 by placing it in the lecturer's letterbox. You are also welcome to give it directly to the lecturer.

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
University of Southern Denmark, Odense

Spring 2016
KSL

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