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

February 1, 2010
KSL

Exam Project in Compiler Construction, part 2

Kim Skak Larsen
Spring 2010

Introduction

In this note, we describe one part of the exam project which must be solved in connection with the project "A compiler for an imperative programming language", Spring 2010. 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

Friday, February 26, 2010, at noon.

A scanner and parser in C

Among other things, you must turn in a program which must be written in the programming language C. It must be the variant which is called ANSI-C. This excludes C++, in particular.

You must construct a scanner using the tool FLEX and a parser using the tool BISON. Using BISON, you must build an abstract syntax tree. Finally, you must write a prettyprinter, which should be used to document that the scanning, parsing, and building of the syntax tree have been carried out correctly. Here, a prettyprinter is a program which prints the abstract syntax tree with appropriately chosen indentation and/or sufficiently many parenthesis to make it possible, preferably easy, to verify the abstract syntax tree.

The language you will work with is called TIPU, and it is partially defined by the grammar in Fig. 1 and 2. The figure has been split in two for typographical reasons only.

The start symbol is ⟨body⟩, and all terminal symbols are written in bold face.

There is more information about the language below. It is part of the assignment to decide which of these could most conveniently be dealt with in these phases and which should be postponed until the phases weed, symbol, type checking, and code generation.

⟨function⟩ : ⟨head⟩ ⟨body⟩ ⟨tail⟩
⟨head⟩ : **func id (** ⟨par_decl_list⟩ **) :** ⟨type⟩
⟨tail⟩ : **end id**
⟨type⟩ : **id**
| **int**
| **bool**
| **array of** ⟨type⟩
| **record of** { ⟨var_decl_list⟩ }
⟨par_decl_list⟩ : ⟨var_decl_list⟩
| ε
⟨var_decl_list⟩ : ⟨var_decl_list⟩ **,** ⟨var_type⟩
| ⟨var_type⟩
⟨var_type⟩ : **id :** ⟨type⟩
⟨body⟩ : ⟨decl_list⟩ ⟨statement_list⟩
⟨decl_list⟩ : ⟨decl_list⟩ ⟨declaration⟩
| ε
⟨declaration⟩ : **type id =** ⟨type⟩ **;**
| ⟨function⟩
| **var** ⟨var_decl_list⟩ **;**
⟨statement_list⟩ : ⟨statement⟩
| ⟨statement_list⟩ ⟨statement⟩
⟨statement⟩ : **return** ⟨expression⟩ **;**
| **write** ⟨expression⟩ **;**
| **allocate** ⟨variable⟩ ⟨opt_length⟩ **;**
| ⟨variable⟩ **=** ⟨expression⟩ **;**
| **if** ⟨expression⟩ **then** ⟨statement⟩ ⟨opt_else⟩
| **while** ⟨expression⟩ **do** ⟨statement⟩
| { ⟨statement_list⟩ }
⟨opt_length⟩ : **of length** ⟨expression⟩
| ε
⟨opt_else⟩ : **else** ⟨statement⟩
| ε
⟨variable⟩ : **id**
| ⟨variable⟩ **[** ⟨expression⟩ **]**
| ⟨variable⟩ **. id**

Figure 1: Grammar for TIPU, part 1.

⟨expression⟩ : ⟨expression⟩ **op** ⟨expression⟩
| ⟨term⟩
⟨term⟩ : ⟨variable⟩
| **id (** ⟨act_list⟩ **)**
| **(** ⟨expression⟩ **)**
| **!** ⟨term⟩
| **|** ⟨expression⟩ **|**
| **num**
| **true**
| **false**
| **null**
⟨act_list⟩ : ⟨exp_list⟩
| ε
⟨exp_list⟩ : ⟨expression⟩
| ⟨exp_list⟩ **,** ⟨expression⟩

Figure 2: Grammar for TIPU, part 2.

The semantics of the various constructions are mostly obvious, based on usual computer scientific tradition. The few which are not are also discussed below.

Further requirements for Tipu programs

The list below is intentionally incomplete. Partly because not all information is relevant right now and partly because some of the decisions of this nature should be made as a part of answering the project as a whole. Some information is relevant for this part of the project, but most have been included to give a sufficient impression of the language.

- **id** are usual identifiers.
- **num** are usual integers.
- A function name is repeated after the **end** which terminates the function definition. Thus, the two **id**s after **func** and **end** must be identical.
- At a function call, parameters which are simple types are passed as values whereas composite types (arrays and records) are passed by reference.
- All invocations of a function must result in the execution of a **return** statement.
- **write** prints the value of ⟨expression⟩, which can be limited to being an integer, followed by a return.
- **allocate** ⟨variable⟩ **of length** ⟨expression⟩ allocates space in memory. This space is of size ⟨expression⟩ for an array with the name ⟨variable⟩, while **allocate** ⟨variable⟩ allocates space for a record of ⟨variable⟩'s type.

- A function definition introduces a new (nested) scope.
- { ⟨statement_list⟩ } is a "compund statement", which can be used for grouping statements such that more than one statement can be executed in a **while**-construction, for example.
- **op** can be **+**, −, *****, **/**, **==**, **!=**, >, <, >**=**, <**=**, **&&**, | |.
- **|** ⟨expression⟩ **|** can denote the size of an array or the absolute value of an integer expression.
- Array indices start with 0.
- **null** is the standard value for a reference variable (array and record).
- # is used as the beginning of a one-line comment. The comment is ended by newline.
- **(*** is used as the beginning of a multi-line comment and ***)** closes the comment. As the name indicates, such a comments may run over several lines, though it may also be closed on the same line it is started. These comments can also be nested.

The Abstract Syntax Tree

Note that the `tiny expressions` example from the home page gives you the basic structure of the various files which are involved. However, be sure to understand and rethink all parts of this little example. Not everything in this little example is appropriate for a real compiler as the one you will be making. In particular, you should structure your AST using a number of typedefs roughly corresponding to the number of different left-hand sides in your grammar.

Turning in

Electronically, you must turn in

- a FLEX file.
- a BISON file.
- a C-program, which uses the files produced by the definition files above to implement a prettyprinter of TIPU-programs from an AST, build via the BISON definition file.
- a makefile, connecting all of the above.

Additionally, you must hand in a report with program listings of all of the above, along with brief descriptions of the most important choices made in the process; among these, grammar rewriting or other actions taken to remove conflicts. You must include a sufficient and documented testing. See also the standard requirements.

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 which 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 which 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 -g --header="Printed by group NN"
```

where `NN` is your group number.

The report must contain a description of the most important and relevant decisions which 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 which 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. For the final part, you must turn in two copies.

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 2010
KSL

CC, Spring 2010
Exam Project, part 2

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.