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

February 4, 2019
KSL

Exam Project in Compiler Construction, part 2

Kim Skak Larsen
Spring 2019

Introduction

In this note, we describe one part of the exam project that must be solved in connection with the compiler project, Spring 2019. 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 22, 2019, at 12:00 (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 c11 ANSI standard as specified by the options below. This excludes C++, in particular. Your programs should be compiled using

```
gcc -std=c11 -Wall -Wextra -pedantic
```

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 KITTY, 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.

⟨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_type⟩ **,** ⟨var_decl_list⟩
| ⟨var_type⟩
⟨var_type⟩ : **id :** ⟨type⟩
⟨body⟩ : ⟨decl_list⟩ ⟨statement_list⟩
⟨decl_list⟩ : ⟨declaration⟩ ⟨decl_list⟩
| ε
⟨declaration⟩ : **type id =** ⟨type⟩ **;**
| ⟨function⟩
| **var** ⟨var_decl_list⟩ **;**
⟨statement_list⟩ : ⟨statement⟩
| ⟨statement⟩ ⟨statement_list⟩
⟨statement⟩ : **return** ⟨expression⟩ **;**
| **write** ⟨expression⟩ **;**
| **allocate** ⟨variable⟩ **;**
| **allocate** ⟨variable⟩ **of length** ⟨expression⟩ **;**
| ⟨variable⟩ **=** ⟨expression⟩ **;**
| **if** ⟨expression⟩ **then** ⟨statement⟩
| **if** ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩
| **while** ⟨expression⟩ **do** ⟨statement⟩
| { ⟨statement_list⟩ }
⟨variable⟩ : **id**
| ⟨variable⟩ **[** ⟨expression⟩ **]**
| ⟨variable⟩ **. id**

Figure 1: Grammar for KITTY, 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⟩
| ⟨expression⟩ **,** ⟨exp_list⟩

Figure 2: Grammar for KITTY, part 2.

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.

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 Kitty 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 or a boolean, followed by a return. Booleans are printed as the constants, i.e., either the four lower-case letter "true" or the five lower-case letters "false".
- **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 "compound 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 KITTY-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.

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 c11 ANSI standard as specified by the options below. This excludes C++, in particular. Your programs should be compiled using

```
gcc -std=c11 -Wall -Wextra -pedantic
```

In particular, no architecture-dependent option should be added, such as, for instance, `-m32` or `-m64`.

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, Mac, 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:
  - The code produced must be X86 Assembly/GAS Syntax, compilable with

    ```
    gcc -no-pie -m64
    ```

  - 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 front page of your report should contain your group number and the full names and student logins of all group members.

You may omit very large test files and results and only turn these in electronically.

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

```
a2ps --line-numbers=1 --tabsize=4 -g --header="Group
    ⟨Group number⟩" -o ⟨Filename⟩.ps ⟨Filename⟩.c
```

where `NN` is your group number. You can print `ps` directly or convert `ps` to `pdf` using `ps2pdf`. However, there are also other 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 or this direct link:

`https://imada.sdu.dk/~kslarsen/CC/elafl.php`

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 whitespace and 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.