Compiler Theory: Assignment 2

Marc Moreno Maza
University of Western Ontario
CS447b – Winter 2004

Posted: Tuesday, February 17, 2004
Due: Tuesday, March 16, 2004 (extended)

1 Guidelines

Date of postage. Tuesday, February 17, 2004.

Normal date for submission. Tuesday, March 16, 2004 (no penalties).

Last date for submission. Sunday, March 21, 2004 (late penalties apply from Monday 15).

Due format. The assignment should be returned to the instructor by email. Your submission will have the form of a UNIX *archive* created with the UNIX command `tar`. This archive will have a structure similar to those of the example archives available on the course web site. More precisely, a `src` directory will contain your `flex/lex` and `yacc` source programs, a `test` directory will contain your test files, a `Makefile` will define the targets and a `README` will explain these targets. Moreover, your answers to the assignment questions (that are not computer programs) will be made in a file

- which may be a text file, or a `postscript` file or a `pdf` file (No other formats are allowed. Moreover, I recommend that you generate your postscript and pdf files under a recent UNIX/Linux distribution.)

- and which will be placed in a `doc` directory.

In addition, you could have a `bin` directory for binaries (as in the course example archives) and a `ps` directory for postscript versions of computer programs (this is convenient to print them!). The computer programs of this assignment are all input files for the `flex/lex` and `yacc` program, together with C program or header files. The preferred version of `flex/lex` is `flex version 2.5.4`.

Your `Makefile` must work under Linux (with a recent distribution) or under a UNIX system of the GAUL network. You can reuse the Makefile of the archive `yacc.tar.gz` from the course web site. Note that this Makefile was developed with `bash` under Red Hat 7.3 (Linux) and using GCC version 2.96.

Your `README` must specify which UNIX system you are using (Linux or SunOS) and which shell (command language interpreter) you are using (`tcsh` or `bash`). For the compilation options of `flex/lex` and `yacc` please refer to the Makefile of the archive `yacc.tar.gz`.

Late penalties. See the course outlines.

Collaboration. You are expected to do this assignment on your own without assistance from anyone else in the class.

Marking. This assignment will be marked out of 40. The allocation of marks is shared equally between the four activities described below.

2 Preliminaries

You will also learn a lot by studying last year's project (the *MOOL* language) which was proposed as Assignment 3. Observe that the archive `yacc.tar.gz` contains samples of different steps in the development of a *MOOL* compiler.

`mool-0.y` generates a parser for MOOL programs restricted to classes declarations, themselves restricted to attributes declarations.

`mool-1.y` generates a parser similar to `mool-0.y` but provides animation to figure out in which order YACC actions are performed.

`mool-2.y` generates a parser similar to `mool-1.y`. In addition, stores the names of the defined classes. Hence, this is a preliminary stage for symbol table computations.

`mool-3.y` generates a parser similar to `mool-2.y`. In addition, stores the names of the defined attributes. It performs also some little checks.

`mool-4.y` is similar to `mool-3.y` except that it uses an external C file `mool-aux.c` for C code.

You are strongly suggested to proceed in the same way, carefully step by step:

1. Realize a first parser that will parse only category definitions (i.e. no domain definitions, no `main` definitions).

2. Realize a second parser from the previous one that will parse category and domain definitions with very limited statements, say just assignments (i.e. no alternatives, no loops, no arithmetic/boolean expressions, ...).

3. Your third parser will recognize arithmetic/boolean expressions.

4. The fourth will add statements,

5. Finally, enhance this parser with the construction of symbol tables. Here again, you can proceed step by step:

   (a) a symbol table for the whole program that will only store category and domain names,

   (b) a symbol table for each category that will store function declarations,

   (c) a symbol table for each domain that will store its category and its attributes,

   (d) a symbol table for each function definition that will store the domain it belongs to, its local variables, ...

3 The Assignment

The objective is to write a parser for the ALLCOT language. Please, refer to the document titled *The* ALLCOT *Language* and available on the course web site.

3.1 Status of your ALLCOT

The grammar, and also the language, you designed in Assignment 1 may still evolve! During the realization of the parser, you may discover new inconsistencies or conflicts that `yacc` cannot resolve. Indicate what is the current language that your parser recognizes. To do so,explain what are the modifications your version of ALLCOT contains w.r.t. to that of the document on the course web site. Here are some features you may discuss: arrays, errors, comments, arithmetic operators ...

Remember that the only additional requirement for your ALLCOT w.r.t. the original one is to have arrays. The other features are optional. In addition, your `array` type constructor does **not** need to be more sophisticated than the one of the PASCAL language (where arrays have fixed size).

3.2 A parser for ALLCOT

Using `flex/lex` and `yacc`, write a first parser for your version of ALLCOT without creation of symbol tables. The only goal of this parser is only to check whether an input text file is a valid ALLCOT program or not. Your parser should report on errors. But, *error recovery* is not required. So, your parser may halt the parsing after the first error is met.

3.3 Symbol Tables for ALLCOT

Enhance your parser such that it creates symbol tables. Again, please refer to the Assignment 3 of the year 2003-2004 together with the `yacc.tar.gz` archive.

To make things simple, one could avoid linked lists and use only records and arrays instead. You may have a look at the file `mool-3.h` in the `yacc.tar.gz` archive.

At the end of the parsing, your parser should report on the contents of the symbol-tables. Here's an example using `mool.y`, `mool.l`, `mool.h` linked to mool-3.y, mool-3.l, mool-3.h.

```
[moreno@iguanodon yacc]$ make mool.out
...................................

[moreno@iguanodon yacc]$ ./bin/mool < test/p0.mool
Parsing program ...
   Parsing class Pair ...
   Class parsed.
Program parsed.
Class[0] is Pair
      Attribute 0 : i of type -1
      Attribute 1 : j of type -1
```

3.4 Test Programs for ALLCOT

Update your three ALLCOT test programs, such they correspond to your current version of ALLCOT and such that they are valid ALLCOT programs.

Moreover, add short programs (say about 10) each of them in the order of 10 lines and generating a **syntax error** (not semantic error).

Example 1

```
MyCat: Category == add {
  foo: Integer -> %;
}
MyDom: MyCat == with {
  attrib(n: Integer);
  foo(n: Integer): % == attrib(n);
}
main(): () == {
}
main();
```

Observe that the following is a valid ALLCOT program (but semantically incorrect).

Example 2

```
MyCat: Category == with {
   foo: Integer -> %;
}
MyCat: Integer == Boolean add {
   attrib(b: Boolean);
   bar(p: Program): Compiler == compiler(p);
   ooops(p: Program): Error == crash(p);
}
main(): () == {
   Boolean: String :=  MyCat;
}
main();
```