Compiler Theory: Assignment 2

Marc Moreno Maza
University of Western Ontario
CS447a – Fall 2004

Posted: Tuesday, October 19, 2004
Due: Tuesday, November 9, 2004

1 Guidelines

Date of postage. Tuesday, October 19, 2004.

Normal date for submission. Tuesday, November 9, 2004 (no penalties).

Last date for submission. Sunday, November 14, 2004 (no penalties).

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 given below.

2 Preliminaries

You will also learn a lot by studying the (the *MOOL* language) project given in the Winter 2003 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, it 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 objectives are:

- To write a parser for the ALLCOT language. Please, refer to the document titled *The* ALLCOT *Language* and available on the course web site.
- To define an intermediate language (to be used later on as the target language) for your ALLCOT compiler. Please refer to section on *Three-Address Code* in the chapter on *Intermediate Code*.
- To initiate the development of an optimizer for this intermediate language. Please refer to the first two sections of the chapter *Code Optimization*.

3.1 A parser for ALLCOT (15 marks)

Using `flex/lex` and `yacc`, write a first parser for the version of ALLCOT without creation of symbol tables. Use the definition of the ALLCOT as stated in Section 3 of *The* ALLCOT *Language* document. Extending the language with lists (or arrays) is optional.

The only goal of this parser is 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.

3.2 Symbol Tables for ALLCOT (10 marks)

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.

Alternatively, you can use data-structures provided by the `glibc` library.

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
```

3.3 Test Programs for ALLCOT (5 points)

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 5) 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();
```

3.4 An Intermediate Language for ALLCOT (5 marks)

Define an intermediate language for ALLCOT with the following constraints

- It is a subset of the standard C language
- It is a *Three-Address Code* language.

3.5 Building Control Flow Graphs for ALLCOT (5 marks)

Explain how you would realize a program that would construct the control flow graph of the intermediate representation of an ALLCOT program.