CSCI 5161: Introduction to Compilers
Spring 2023, University of Minnesota
Homework 3

Posted: Feb 15, 2023
Due: March 3, 2023

Note: This homework involves writing more code than the previous one. This code is still not all that much---you will mainly be writing a grammar description---and the real action will come *after* this point when you won't have tools like *ML-Lex* and *ML-Yacc* to generate the code for you from high-level descriptions.

The Objective in This Homework

In this homework, we will build a parser for the *Tiger* language that generates an abstract syntax representation for programs in that language. This combines the programming problems at the end of Chapters 3 and 4. I have placed a working scanner in the starting code in case your scanner is still not working satisfactorily. Use this while you continue to fix your scanner. *Note that you must not change the terminal declarations in the beginning of the file tiger.grm given to you as the starting grammar if you want to use my lexer*. Note also that if you are using your own lexer then you would have to generate it from something extra added to the *tiger.lex* file for it to be able to talk properly with the parser. What has to be added is described in detail below.

To do this homework, you will need to familiarize yourself with ML-Yacc and with the abstract syntax we want to generate for *Tiger* programs. The latter is explained on page 98 of the book.

You may wonder why we have combined the two programming projects at the end of Chapters 3 and 4. The reason for this is that without also doing the work at the end of Chapter 4, it is not possible to see the results of your parser; in particular, it does not output anything that you can look at to check if it has worked correctly. However, in actually doing the homework, you may want to work in stages, first getting your grammar working and only then including semantic actions that generate abstract syntax. To help you do this, I have included the code and the testing scaffold for the two phases below in the directories called pre-hw3 and hw3, the tarred, gzipped versions of which are available in pre-hw3.tar.gz and hw3.tar.gz, respectively. Use both as you wish. *However, do not turn in a copy of pre-hw3 unless you have not been able to finish the full hw3 to your satisfaction*.

The Specific Task

Do the programming problems at the end of Chapters 3 and 4 in the text, i.e. those starting on pages 81 and 100 respectively; at the end of this, you should have a parser for Tiger programs that will produce abstract syntax representations of the kind discussed in Chapter 4. You really only need to change the contents of the file tiger.grm. As explained previously, you should turn in a complete version of your hw3 directory so that our job of testing your code is not too cumbersome.

In addition to the files corresponding to the program, include in your submission a file called `hw3.txt` that corresponds to the written part of your homework. **This file should be in plain text format---not in Word and also not in pdf form.** In this file, you should document all the shift-reduce conflicts that arise out of your grammar and you should argue why these conflicts are harmless. One further thing that I want you to discuss in this file is why the shift-reduce conflict that is mentioned on page 82 of the book is problematic and what you have done in your grammar to avoid it. Finally, include in this file any relevant discussion of nifty error correction ideas that are embedded in your `ml-yacc` description file.

If you are using your own *tiger.lex* file from homework 2, you will have to add something to it in the beginning to allow it to talk properly with the parser that is generated from the *tiger.grm* file. This is needed because the parser and the lexer communicate via tokens and so should have a common description of these. Essentially, the parser will be producing the token type from the terminal declarations early in the *tiger.grm* file and the declarations below will set things up so that the lexer also gets to use these declarations:

```
type svalue = Tokens.svalue
type pos = int
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult = (svalue,pos) Tokens.token
```

You will also need to add the following line to the middle section of the file, i.e., the section where you define states, macros, etc.

```
%header  (functor TigerLexFun(structure Tokens: Tiger_TOKENS));
```

This line changes the name of the functor/structure that is generated from the lexer description by ml-lex. The code in parse.sml assumes that a functor of this name and functionality is the one that is generated.

Make sure to test your grammar using various relevant test cases present in the syntax-and-sem-analysis subdirectory of the testcases directory that is available in tarred and gzipped form here.

Some Further (Helpful) Comments

Here are a few comments that might help you in doing this assignment:

- Looking at examples is usually the best way to start the learning process. I found the calculator example in the ML-Yacc Manual quite helpful.

- You may wonder about the contents of the files table.sml and symbol.sml in the directory. For the moment, think of these as black boxes that eventually simply yield the type *symbol* that is used to represent identifiers of different kinds in the abstract syntax; to get an object of this type from a string, you would use the function *Symbol.symbol* on the string. In due course we will understand the structure of the symbol representation better.

- As suggested above, you may find it easier to do the assignment in stages: first write a grammar with the null semantic action being associated with each rule, sort out all the conflicts and generally debug the grammar (perhaps testing your error correction ideas), and only then add the semantic actions needed for building abstract syntax representations. One thing to keep in mind, though is that your grammar rules should be structured so as to allow the right abstract syntax to be extracted. This is not an issue in most cases. However, it is important to bear the following comment in mind with regard to the treatment of function and type declarations. As explained on pages 97-99 of the book, a sequence of such declarations should be *combined* into one to treat mutually recursive declarations properly; this will require anticipating a sequence in your grammar rules.

- You will discover in your treatment of `control flow' expressions like `while`, `for` and even `if` that there will be a conflict with operators such as `+`. For example, how is

  ```
  while 1 do i + j
  ```

  to be treated, as a top-level while statement or an addition? I argue for the former. If you disagree with this, let us have a Piazza discussion about it. Otherwise implement this approach.

- Sometimes the semantic action to be carried out at a particular rule may have to depend on the semantic action at the parent; from a terminological perspective, what you are using in parsing and translating into an internal form is an attribute grammar, and when information flows from the parent to the child in a parse tree, you have the case of an *inherited attribute*. In my treatment of the `conflict' described on page 82, I had to use such an attribute. As we would have discussed in class, this is really easy to do in ML, given its support for higher-order functions. Think about using this approach in your code.

- The Tiger directory for Chapter 4 has a pretty printer for abstract syntax that you can use to determine if your parser is working right. The version of the driver parse.sml that I have placed in the code directory defines a function called `parseandprint` that will allow you to parse a program in a named input file and pretty print the resulting abstract syntax to a named output file. To run it, after you have built the system you should type

  ```
  Parse.parseandprint "foo.tig" "foo.abs"
  ```

  There is a similar driver in parse.sml in the `pre-hw3` directory except, of course, this parse function has only one argument.

- There is a automated testing scaffold set up for this homework along the lines of the one for homework 2. Read the documentation in the directory to understand how to use this. You can also download the directory (using the tarred and gzipped version here) and use it to test your parser.

Last updated on Feb 14, 2023 by ngopalan atsign umn dot edu

The views and opinions expressed in this page are strictly those of the page author(s). The contents of this page have not been reviewed or approved by the University of Minnesota.