

Assignment 3 (Due: Nov 10) (15 pts)

1 Introduction

In this assignment you will write a parser for Cool. The assignment makes use of two tools: the parser generator `bison` and a package for generating and manipulating trees. You will not use the generating part of the package, but only the manipulators of an already generated tree definition we will give you. The output of your parser will be an abstract syntax tree (AST). The AST will be constructed using `bison`'s semantic actions.

You certainly will need to refer to the syntactic structure of Cool, found in Figure 1 of the CoolAid manual, as well as other portions of the reference manual. An HTML manual for `bison` is available at the URL http://www.delorie.com/gnu/docs/bison/bison_toc.html. The tree package is described in the *Tour of Cool Support code* handout. You will need the tree package information for this and future assignments in CS781.

There is a lot of information in this handout and you need to know most of it to write a working parser. Please read the handout thoroughly.

You will be working in pairs on this assignment, similarly to the previous assignment. So, you are required to turn in only one copy of the code and the writeup, per group. Clearly identify the members of your group by name and email address in the README file.

2 Files and Directories

To get the assignment type

```
/usr/local/bin/make -f /usr/local/lib/cool/assignments/PA3/Makefile
```

in a directory named PA3. This command copies a number of files to your directory, some of them with read-only permission. As usual, you should not modify files that are read-only. If you insist on making your own versions of these files, you are on your own. If your code doesn't compile with the original files (including the original `Makefile`) you will receive little credit, because your program will fail all of the automatically run test cases. Please read and follow the directions in the README file.

The files that you need to modify are:

- `cool.y`
This file contains a start towards a parser description for Cool. You will need to add more rules. The section before the first `%%` is mostly complete; all you need to do is add `%type` declarations for new nonterminals. (We have given you names and `%type` declarations for the terminals.) The rule section is very incomplete.
- `good.cl` and `bad.cl`
These test a few features of the grammar. You should add tests to ensure that `good.cl` exercises every legal construction of the grammar and that `bad.cl` exercises as many types of parsing errors as possible in a single file. Explain your tests in your writeup.

The files that your code will link with and include, but that you should not modify, are:

- `parse-phase.cc`
This file contains a simple driver for the parser.
- `cool-tree.aps`
This file is a specification of the Cool tree language written in a special language. It defines a type system and describes the kinds of nodes used for the abstract syntax trees.
- `cool.h`, `cool-tree.h`, `cool-tree.handcode.h`, and `tree.h`
The header file `cool.h` defines basic types used by the compiler. The other files define the routines that let you create and examine abstract syntax trees and lists of trees. It may interest you that the second file was automatically generated from `cool-tree.aps`.

- `dumptype.cc`

This file implements a routine for printing out ASTs. The algorithm used illustrates the use of virtual functions to recursively traverse an AST; you will need to write algorithms in a similar style in future assignments.

Important: All software supplied with this assignment is supported on `gandalf.cs.wright.edu`. Recall that you should place `/usr/local/lib/cool/bin` at the beginning of your `PATH` variable if you want to run the appropriate binaries for `coolc` and `spim` without typing in the entire path.

The syntax for I/O redirection depends on the specifics of the UNIX shell you are using. The Makefile contains syntax suitable for use in `tcsh` and `bash`, with the default set for `tcsh` version. Copy and change the Makefile (by uncommenting appropriate commands and commenting the others) for the turnin to succeed. Alternatively, invoke `tcsh` and run the `turnin` command.

Ignore spurious messages of the form — “`file.o: ... weak symbol multiply defined ...`” — involving `*.o` files such as `dumptype.o`, `cool-parse.o`, `handle_flags.o`, etc, and warnings of the form — “[`cool incl dir`]/`file.h: ... control reaches end of non-void function`” or incompatibility warnings such as “[`gnu c++ incl dir`]/`...: This file includes at least one deprecated or antiquated header.`”, while building the parser.

3 Testing the Parser

You will need a working scanner to test the parser. You may use either your own scanner or the `coolc` scanner. By default, the `coolc` scanner is used. To change that, replace the `lexer` executable (which is a symbolic link in your project directory) with your own scanner. See also the `README` file for instructions. Don’t automatically assume that the scanner (whichever one you use!) is bug free—latent bugs in the scanner may cause mysterious problems in the parser.

You will run your parser using `myparser`, a shell script that “glues” together the parser with the scanner. Note that `myparser` has a `-p` flag for debugging the parser; using this flag causes lots of information about what the parser is doing to be printed on `stdout`. `Bison` produces human-readable dump of the LALR(1) parsing tables in the `cool.output` file. Examining this dump can be useful for debugging parser definition. In addition, the `-l` flag is available for generating debugging output from the scanner. Using these two flags together are sometimes useful for isolating strange behavior in the parser.

Once you are confident that your parser is working, try running `mycoolc` to invoke the complete compiler that includes your parser. You should test this compiler on both good and bad inputs to see if everything is working. Remember, bugs in your parser may manifest themselves anywhere.

Your parser will be graded using our lexical analyzer. Thus, even if you do most of the work using your own scanner you should test your parser with the `coolc` scanner before turning in the assignment. As always, type `make turnin` to turn in your assignment; see the `README` file.

4 Parser Output

Your semantic actions should build an AST for the input. The root (and only the root) of the AST should be of type `Program`. The semantic action for the start symbol assigns the root of the AST to a global variable `ast_root` (also of type `Program`, of course). For programs that parse successfully, the output of `parser` is a listing of the AST.

For programs that have errors, the output is the error messages of the parser. We have supplied you with a function `yyerror` in `cool.y` that prints error messages in a standard format; please do not modify this function. You should not invoke `yyerror` directly in the semantic actions; `bison` automatically invokes `yyerror` when it detects a problem.

Your parser need only work for programs contained in a single file—don’t worry about compiling multiple files.

5 Error Handling

You should use `bison`'s `error` pseudo-nonterminal to add error handling capabilities in the parser. The purpose of `error` is to permit the parser to continue after some anticipated error. It is not a panacea and the parser may become completely confused. See the `bison` documentation for how best to use `error`. In your writeup, describe which errors you attempt to catch. Your test file `bad.cl` should have some instances that illustrate the errors from which your parser can recover. To receive full credit, your parser should recover in at least the following situations:

- If there is an error in a class definition but the class is terminated properly and the next class is syntactically correct, the parser should be able to restart at the next class definition.
- Similarly, the parser should recover from errors in features (going on to the next feature), a `let` binding (going on to the next variable), and an expression inside a `{...}` block.

Do not be overly concerned about the line numbers that appear in the error messages your parser generates. If your parser is working correctly, the line number will generally be the line where the error occurred. For erroneous constructs broken across multiple lines, the line number will probably be the last line of the construct.

6 The Tree Package

There is an extensive discussion of the tree package for Cool abstract syntax trees in the *Tour of COOL Support Code*. You will need most of that information to write a working parser.

7 Remarks

You may use `bison`'s precedence declarations, but only for expressions. Do not use precedence declarations blindly—do not respond to a shift-reduce conflict in your grammar by adding precedence rules until it goes away. If you find yourself making up rules for things other than operators in expressions, and for `let`, you are probably doing something wrong.

The Cool `let` construct introduces an ambiguity into the language. The manual resolves the ambiguity by saying that a `let` expression extends as far to the right as possible. The ambiguity will show up in your parser as a shift-reduce conflict involving the productions for `let`.

This problem has a simple, but slightly obscure, solution. Here is a hint. In `coolc`, we implemented the resolution of the `let` shift-reduce conflict using a `bison` feature that allows precedence to be associated with productions (not just operators).

You must declare `bison` “types” for your non-terminals and terminals that have attributes. For example, in the skeleton `cool.y`, there is the declaration:

```
%type <program> program
%type <class_> class
```

This declaration says that the non-terminals `program` and `class` have types `<program>` and `<class_>` respectively. The use of the word “type” is misleading here; what it really means is that the attribute for the non-terminal `class` is stored in the `class_` member of the `union` declaration in `cool.y`, which has type `Class_`. By specifying the type

```
%type <member_name> X Y Z ...
```

you instruct `bison` that the attributes of non-terminals (or terminals) `X`, `Y`, and `Z` have a type appropriate for the member `member_name` of the union.

All the union members and their types have similar names by design. It is a coincidence in the example above that the non-terminal `class` has the same name as a union member.

It is critical that you declare the correct types for the attributes of grammar symbols; failure to do so virtually guarantees that your parser won't work. You do not need to declare types for symbols of your grammar that do not have attributes.

The `g++` type checker complains if you use the tree constructors with the wrong type parameters. If you ignore the warnings, your program may crash when the constructor notices that it is being used incorrectly. Moreover, `bison` may complain if you make type errors. Heed any warnings. Don't be surprised if your program crashes when `bison` or `g++` give warning messages.

8 What to Turn in

When your group is ready to turn in the assignment, type `/usr/local/bin/make turnin` in the directory where you have prepared your assignment. This action will copy the three files of the assignment (`cool.y`, `good.cl`, `bad.cl`) and the two output files (`good.output`, `bad.output`) to the instructor's directory. The last turnin you do will be the one graded. Each turnin overwrites the previous one.

In your writeup: explain your design decisions, your test cases, and why you believe your program is correct and robust. Describe which errors you attempt to catch and how you recover from them. Also document any known bugs in your program.

It is part of the assignment to explain things in text as well as to comment your code. Take the extra time to clearly and concisely explain your program.