CS-GY 6413/CS-UY 3943 — Programming Assignment 3

Jeff Epstein

Introduction

We will write a compiler for the Cool language. Each of several programming assignments will cover one component of the compiler: scanner, parser, semantic analyzer, and code generator. When combined, these components will result in a complete compiler. This assignment covers the parser.

Before starting, please make sure that you're using a suitable programming environment, as discussed in previous assignments.

You should complete the assignment in the C++ language, using the `bison` parser generator. Please refer to the `bison` manual, available on Classes.

This assignment assumes that you are already familiar with the Cool language. Please refer to the Cool manual, available on Classes, for details on the language.

Collaboration policy Undergraduate students may work on this assignment either alone, or in a group of exactly two people. Both members of a group must be enrolled in the same section: that is, undergrads may not collaborate with grad students. If working in a group, make sure to submit it on Gradescope using the group submission option. Collaboration with anyone not in your group is prohibited. Graduate students must work alone.

Assignment

Your task is to implement a parser that can syntactically analyze a Cool program. The input to your parser will be a token stream produced by a Cool lexer. The output of your program will be an abstract syntax tree. You should use the language description given in the Cool manual, in particular the sections "Cool Syntax," and "Informal language description." You may also refer to the provided example programs.

You will write the parser using `bison`, a tool for generating parsers. Using `bison`, you can specify grammar rules and their associated semantic actions that will create an abstract syntax tree (AST).

Parser output

Your semantic actions should build an AST. The root (and only the root) of the AST should be of type `program`. For programs that parse successfully, the output of the 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 an error reporting function that prints error messages in an appropriate format. It will be invoked automatically by the parser generator: do not modify it or call it directly.

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

Tree package

You are provided with C++ classes and functions for constructing the Cool abstract syntax tree. You must use these tools in your parser.

These tools are documented in *A Tour of the Cool Support Code*, found in `cool-tour.pdf` on Classes. Please see section 6, "Abstract Syntax Tree."

Error handling

If there's a parse error, we would like our parser to recover and continue parsing, rather than just give up. We provide some suggestions for how to achieve this.

You should use the special `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. See the `bison` documentation for how best to use `error`.

In your `README` file, 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 a feature (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 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.

Other notes

- `bison` allows precedence declarations (using `%left`, `%right`, and `%noassoc`), which can be used to disambiguate the precedence of operators, thus removing conflicts. Please use these declarations judiciously: they should be used only within expressions. Most of the time, there are more elegant ways to remove a parser conflict.
- The Cool `let` construct introduces an ambiguity into the language. Consider the code:

```
class Main inherits IO {
  main() : Object {
    let x : Int <- let y : Int <- 5 in y + 1 in out_int(x + 1)
  };
};
```

The problem is that the parser may not know which `in` is associated with which `let`. The following parse would be wrong:

```
[[let x : Int <- let y : Int <- 5 in y + 1]] in x + 1
```

The ambiguity will show up in your parser as a shift-reduce conflict involving the productions for `let`.

The Cool manual clarifies the correct interpretation of such code: a `let` expression extends as far to the right as possible. But how to express that sentiment within `bison`? To solve this problem, use `bison`'s `%prec` construct, which lets you assign precedence to a particular grammar production. See the `bison` documentation for more info.

- Please make sure that your parser doesn't output anything other than the syntax tree itself, or else it won't work in the compiler pipeline. Don't use `printf`, `cout<<`, etc., in your final submitted version.
- Each grammar production in `bison` is expected to produce a value, which is assigned to the special symbol `$$`. In the case of our Cool parser, that value will be always be an AST node. Each kind of expression has its own type of AST node: for example, a Cool class is represented by the `class__class` type, a boolean constant is represented by the `bool_const_class` type, etc. These AST nodes are in turn subtypes of so-called *phylum* types, `Class_`, `Expression`, etc. This hierarchy of AST types is explained in more detail in `cool-tour.pdf`. In the provided `cool.y` file, you are given a `%union` declaration, where each possible AST phylum is mapped to a field. The union is defined as follows, and you should not change it:

```
%union {
  Boolean boolean;
  Symbol symbol;
  Program program;
  Class_ class_;
  Classes classes;
  Feature feature;
  Features features;
  Formal formal;
  Formals formals;
  Case case_;
  Cases cases;
  Expression expression;
  Expressions expressions;
  char *error_msg;
}
```

`bison` needs to know the type of each grammar production. We express that using `%type` declarations in the `bison` code, where a particular nonterminal is mapped to a field in the union. For example, in the skeleton `cool.y` are the declarations:

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

These say that the nonterminal `program` is associated with the union field `program` (which has type `Program`); the nonterminal `class_list` is associated with the union field `classes` (which has type `Classes`); and the nonterminal `class` is associated with the union field `class_` (which has type `Class_`).

By specifying the type declaration `%type <member_name> X Y Z ...`, you instruct `bison` that the production for terminals (or nonterminals) X, Y, and Z have a type appropriate for the member `member_name` of the union.

You will need to add `%type` declarations as you develop your grammar. It is critical that you declare the correct types for the attributes of grammar symbols. You do not need to declare types for symbols of your grammar that do not have attributes.

The C++ 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 such warnings.

- Your parser should not cause shift/reduce or reduce/reduce conflicts. If `bison` indicates that you have conflicts, but you don't know where, you can use `bison`'s `-v` flag to produce a `cool.output` file that will help you locate the source of conflicts.
- `bison` supports a GLR parser mode, however you may not use it.

Setup

Download two files: `pa3.zip`, which contains starter code specifically for this assignment; and `refimpl.zip`, which contains a reference implementation of the Cool compiler. Extract the content of both zip files into separate directories.

There are a few files that are of interest of us:

- `cool.y` — This file contains a start towards a parser description for Cool. The declaration section is mostly complete, but you will need to add additional parsing rules for new nonterminals you introduce. You have been given names and type declarations for the terminals. You might also need to add precedence declarations. The rule section, however, is rather incomplete.
- `good.cl` and `bad.cl` — These files test a few features of the grammar. You must add tests to these files 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 these files and put any overall comments in the `README` file.
- `README` — This file contains detailed instructions for the assignment as well as a number of useful tips. You should also edit this file to include the write-up for your project. You should explain design decisions, why your code is correct, and why your test cases are adequate. It is part of the assignment to clearly and concisely explain things in text as well as to comment your code.

You don't need to modify any other file.

Running the `make` command from that directory will compile your code. If compilation is successful, it will produce an executable file `parser`.

Testing

There are several ways you can test your parser. In all cases, it's up to you to design inputs that adequately test all features of your parser: don't rely only on the provided test files.

Direct testing You need a working scanner to test the parser. You can use either your scanner from the previous assignment, or the scanner from the reference implementation. The following `bash` command will call the reference compiler's `lexer` executable, giving it the provided file `good.cl` as input, and piping the resulting token stream as input to the reference compiler's parser:

```
../refimpl/lexer good.cl | ../refimpl/parser
```

This produces the following AST:

```
#1
_program
  #1
  _class
    A
```

```
    Object
    "good.cl"
    (
    )
  #4
  _class
    BB__
    A
    "good.cl"
    (
    )
```

This simple AST shows the structure of the minimal `good.cl` file: a `program` containing two classes, one named `A`, which inherits from `Object`; and one named `BB__`, which inherits from `A`.

You can compare the result with your own parser by substituting your compiled parser executable into the above command line, like this:

```
../refimpl/lexer good.cl | ./parser
```

The output of your parser is handled entirely by the provided AST classes. Your program simply constructs the AST. Your code never needs to output directly.

The parser will automatically produce a human-readable dump of the LALR(1) parsing tables in the `cool.output` file. Examining this dump is frequently useful for debugging the parser definition.

Testing as part of compiler The provided reference implementation of Cool contains complete versions of all four phases of the compiler. You can replace the reference version of the parser with your version, and then verify that the compiler still works correctly.

To replace the parser, just copy your compiled `parser` into the directory where you've extracted the reference implementation:

```
cp parser ../refimpl/parser
```

Now run the compiler `coolc` on your test program. If your parser is correct, the subsequent phases of compilation should run normally, and you can run the compiled test program.

Rules

Your code should compile cleanly with a 5.x or newer version of `gcc` and the provided `Makefile`. Your parser should be free of shift/reduce and reduce/reduce conflicts.

You may use any standard C++ header files, but you may not use any third-party libraries. You must use the provided AST library.

Do not add any additional files to the program. Modify only those files indicated in `README`.

Use good coding style, including informative variable names, consistent indentation, and clear structure. Your code should be robust and generally free of bugs. As a special case, please note that heap leaks are acceptable, as long as they don't impact the functioning of the program.

You are not obligated to use comments. However, confusing, unclear, and uncommented code may result in a reduced grade. The grader must be able to understand your code, and comments are one way to improve the readability of your code.

Submission

Test your code thoroughly before you submit.

In addition to your code, you must answer the questions at the end of the provided `README` file. Write your answers directly in the `README` file.

Run the command `make zip` and check the content of the resulting zip file to make sure that it contains all the code you wrote. Submit only the zip file on Gradescope. Do not add additional files to the zip file.

Acknowledgments

This sequence of assignments is based on work by Alex Aiken and Westley Weimer.