# Programming Assignment 2
# Due Apr 4, 2023 Midnight

For this assignment, you are going to **implement a syntax analyzer or parser** for *Cool*. You will use a parser generator called *bison* to generate a parser and build the AST. The output of your parser will be an abstract syntax tree (AST). You will construct this AST using semantic actions of the parser generator. Please make sure you get familiar with the syntactic structure of *Cool*, as described in Figure 1 of the *Cool* reference manual before you start writing a parser. The documentation for *bison* is available online (link provided on the course website). You will have to consult *A Tour of the Cool Support Code* as well to manipulate AST.

## 1 Files and Directories

To get started, download and unpack the *pa2* directory under Resources from the course website. This directory contains all the files that you will need for this PA. The files you will need to modify are:

- `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 type declarations for new non-terminals you introduce. We have give you names and type declarations for the terminals. The rule section is very incomplete. We have provided some parts of some rules. You should not need to modify this code to get a working solution, but you are welcome to if you like. However, do not assume that any particular rule is complete.

- `test_good.cl` and `test_bad.cl`
  These files test a few features of the grammar. You should add tests to ensure that `test_good.cl` exercises every legal construction of the grammar and that `test_bad.cl` exercises as many types of parsing errors as possible in a single file.

To build the parser, type

*make* or *make parser*

in the directory *pa2/src*. This will link more files of support code to your directory and compile your skeleton. Your parser needs as input the output of your completed lexer. Use `test_good.cl` or a working *Cool* program to test your skeleton parser by typing

*lexer test_good.cl* | *parser*

## 2 Parser Result

Your semantic actions should build an AST using the *Cool* support code tree package, whose interface is defined in *pa2/include/cool-tree.h*. *A Tour of the Cool Support Code* contains an extensive discussion of the tree package for *Cool* abstract syntax trees. You will need most of that information to write a working parser. Read the *Tour* documentation carefully: it contains explanations, caveats, and other details that will help you avoid a number of pitfalls in understanding and using the AST classes.

The root (and only the root) of the AST should be of type `program`. 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 an error reporting routine that prints error messages in a standard format; please do not modify it. You should not invoke this routine directly in the semantic actions; *bison* automatically invokes it when a problem is detected.

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

# 3 Error Handling

You should use the `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`. Your test file `test_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 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.

# 4 Notes

- Your parser must NOT generate any warning about having ANY shift-reduce or reduce-reduce conflicts. All conflicts must be resolved, either by redesigning the grammar rules or by using bison's precedence declarations.
- You may use precedence declarations, but only for expressions. Do not use precedence declarations blindly (i.e., do not respond to a shift-reduce conflict in your grammar by adding precedence rules until it goes away).
- The *Cool* `let` construct introduces an ambiguity into the language (try to construct an example if you are not convinced). The manual resolves the ambiguity by saying that a let expression extends as far to the right as possible. The ambiguity may show up in your parser as a shift-reduce conflict involving the productions for let. If you run into such a conflict, you may want to consider solving the problem by using a *bison* feature that allows precedence to be associated with productions (not just operators). See the *bison* manual for information on how to use this feature.
- You must declare *bison* "types" for your non-terminals and terminals that have attributes. For example, in the skeleton *cool.y* is the declaration:

  ```
  %type <program> program
  ```

  This declaration says that the non-terminal `program` has type `<program>`. The use of the word "type" is misleading here; what it really means is that the attribute for the non-terminal program is stored in the program member of the union declaration in *cool.y*, which has type `Program`. 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 `program` 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.
- Since your compiler uses pipes to communicate from one stage to the next, any extraneous characters produced by the parser can cause errors; in particular, the semantic analyzer may not be able to parse the AST your parser produces. **Please make sure to remove all debug prints from your parser before submitting your assignment as they will cause you to lose points.**

## 5 Testing the Parser

You will need a working scanner to test the parser. Do not automatically assume that the scanner is bug free - latent bugs in the scanner may cause mysterious problems in the parser. *bison* produces a human-readable dump of the LALR(1) parsing tables in the *cool.output* file (see the *bison* manual). Examining this dump may be useful for debugging the parser definition.

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.

## 6 What and How to Turn In

You have to turn in the *pa2* directory with your modified version of *cool.y*, *test_good.cl*, and *test_bad.cl* after compressing it with

*tar -cvf pa2_[your student id].tar pa2*

You can upload the compressed file to the board for assignment submission at the course website. Please do not copy or modify any part of the support code. The provided files are the ones that will be used in the grading process.