Programming Project 2: Syntax Analysis

The Goal

This time around you'll extend your Decaf compiler (at this point it's nothing more than a lexer) to handle the syntax analysis phase, the second task of the front-end, by using **`bison`** to create a parser. The parser will read Decaf source programs and construct a parse tree. If no syntax errors are encountered, your code will print the completed parse tree as flat text. At this stage, you aren't responsible for verifying meaning, just structure. The purpose of this project is to familiarize you with the tools and give you experience in solving typical problems one encounters when using them to generate a parser.

There are two challenges to this assignment. The first is all about **`bison`**—taking your parsing knowledge, coming up to speed on how the tools work, and generating a Decaf parser. The second challenge will come in familiarizing yourself with our provided code for building a parse tree. It's just a starting skeleton to get you going on, and your job will be to fully flesh it out over the course of the projects.
My sense is that in the long run you'll be glad to have had our help code, but you'll have to first invest the time to come up to speed on someone else's code, so be prepared for that in this project.

Due: Thursday, April 25 at 11:59 p.m.

Decaf Program Structure

The reference grammar given in the Decaf language handout defines the official grammar specification you must parse. The language supports global variables, functions, classes and interfaces, variables of various types including arrays and objects, arithmetic and Boolean expressions, constructs such as **`if`**, **`while`**, and so on. First, peruse the grammar specification carefully. Although the grammar is fairly large, most of it is not tricky to parse.

In addition to the language constructs already specified for you, you'll also extend the grammar to support C-style post-increment and decrement expressions along with **`switch`** statements.

Starter files

The starting files are in piazza.
As always, be sure to read through the files we give you to understand what we have provided. If you have questions, be sure to ask! The p2 project directory contains the following files:

`Makefile`	builds project
`main.cc`	**`main`** and its helper functions
`scanner.h`	declares scanner functions and types
`scanner.l`	our **`flex`** scanner for Decaf
`parser.h`	declares parser functions and types
`parser.y`	skeleton of a **`bison`** parser for Decaf
`ast.h/.cc`	interface/implementation of base AST node class
`ast_type.h/.cc`	interface/implementation of AST type classes
`ast_decl.h/.cc`	interface/implementation of AST declaration classes
`ast_expr.h/.cc`	interface/implementation of AST expression classes
`ast_stmt.h/.cc`	interface/implementation of AST statement classes
`errors.h/.cc`	error-reporting class to use in your compiler
`location.h`	utilities for handling locations, **`yylloc`**/**`yyltype`**
`utility.h/.cc`	interface/implementation of various utility functions
`samples/`	directory of test input files

Copy the entire directory to your home directory. Use **`make`** to build the project. The **`Makefile`** provided will produce a parser called **`dcc`**. As usual, **`dcc`** reads input from **`stdin`** , writes output to **`stdout`**, and writes errors to **`stderr`**. You can use standard UNIX file redirection to read from a file and/or save the output to a file:

```
% ./dcc < samples/program.decaf >& program.out
```

On Tuesday night, we will be releasing our own scanner code as **`scanner.l`**.
Take a look at the code to learn how we did things, as it's always interesting to see how different people solve the same problem. You can use our scanner or replace it with your own. Until then, you should use the scanner you wrote for the first assignment. This delay is mostly so that we don't give out a working solution to the last assignment before students using late days have necessarily turned it in.

If you do use your own scanner, you will need to implement a function with the following signature in your **`scanner.l`** file:

```
const char* GetLineNumbered(int num);
```

This function should take in a line number, then hand back the text of the line with that number. Our reference scanner provides this function for you, but it wasn't required in the first assignment and chances are you did not implement this functionality on your own. Consequently, if you want to start early, you can stub out this function by having it always return NULL:

```
const char* GetLineNumbered(int num) {
    return NULL;
}
```

If you want to use your scanner in your final submission, that's fine, but be sure that you provide a working implementation for this function. The error reporting routines will use it to print out really shiny error messages.

Using `bison` to generate the parser

- The given **`parser.y`** input file contains a very incomplete skeleton which you must fill out to accept the correct grammar**.** Your first task is to add the rules for each of the Decaf grammar features. You do not need the actions to build the parse tree yet. In fact, it may help to first concentrate on getting the rules written and conflicts resolved before you add actions. This is precisely what I did when I wrote my own parser.
- Running **`bison`** on an incorrect or ambiguous grammar will report shift/reduce errors, useless rules, and reduce/reduce errors. To understand the conflicts being reported, scan the generated **`y.output`** file that identifies where the difficulties lie. Take care to investigate the underlying conflict and what is needed to resolve it rather than adding precedence rules like mad until the conflict goes away.
- Your parser should accept the grammar as given in the Decaf specification document, but you can rearrange the productions as needed to resolve conflicts. Some conflicts (**`if-else`**, overlap in function versus prototype) can be resolved in a multitude of ways (re-writing the productions, setting precedence, etc.) and you are free to take whatever approach appeals to you. All you need to ensure is that you end up with an equivalent grammar.
- All conflicts and errors should be eliminated, i.e. you **should not** make use of **`bison`**'s automatic resolution of conflicts or use **`%expect`** to mask them. You'll see messages in **`y.output`** like: "Conflict in state X between rule Y and token Z resolved as reduce." This is fine—it just means that your precedence directives were used to resolve a conflict.

The Parse Tree

- There are several files of support code (the generic list class, and the five AST files with various parse tree node classes). Before you get started on building the parse tree, read through these carefully. The code should be fairly self-explanatory. Each node has the ability to print itself and, where appropriate, manage its parent and lexical location (these will be of use in the later projects). Consider the starting code yours to modify and adapt in any way you like.

- We've included limited comments to give an overview of the functionality in our provided classes, but if you find that you need to know more details, don't be shy about opening up the .cc file and reading through the implementation to figure it out. You can learn a lot by just tracing through the code, but if you can't seem to make sense of it on your own, you can send us email or come to office hours.

- You could add actions for each rule as you go, but I recommend that you wait until all rules are debugged and then go back and add actions. The action for each rule will be to construct the section of the parse tree corresponding to the rule reduced for use in later reductions. For example, when reducing a variable declaration, you will combine the **`Type`** and **`Identifier`** nodes into a **`VarDecl`** node, to be gathered in a list of declarations within a class or statement block at a later reduction.

- Be sure you understand how to use symbol locations and attributes in **`bison`**: accessing locations using **`@n`**, getting/setting attributes using **`$`** notation, setting up the attributes union, how attribute types are set, the attribute stack, **`yylval`** and **`yylloc`**, so on. There is additional information on how to do these things (and much more) in the on-line references (though everything you see in the **`bison`** handout should be more than enough.)

- Keep on your toes when assigning and using results passed from other productions in **`bison`**. If the action of a production forgets to assign **`$$`** or inappropriately relies on the default result (**`$$ = $1`**), you don't get warnings or errors; instead, you're rewarded with entertaining runtime nastiness because you use some variable you've never initialized.

We expect you to match our output on the reference grammar, so be sure to use look at our output and make good use of **`diff -w`**. At the end of parsing, if no syntax errors have been reported, the entire parse tree is printed via an inorder traversal. Our parse classes are all configured to properly print themselves in the expected format, so there is nothing new you need to do here. If you have wired up the tree in the correct way, the printed version should match ours, line for line.

Beyond The Reference Grammar

Once you have the full grammar from the Decaf spec operational, you have three creative tasks to finish off your syntax analysis:

1. Postfix expressions. Add post-increment and decrement expressions:

```
i++;
if (i == arr[j]--) j++;
```

   Both of these are unary operators at the same precedence level as unary minus and logical not. You only need to support the postfix form, not the prefix. An increment or decrement can be applied to any assignable location (i.e. anything that has storage). You will need to modify the scanner and parser, and add new parse tree nodes for this new construct.

2. Switch statements. Add productions for parsing a C-style switch statement.

```
switch (num) {
  case 1: i = 1;
  case 2: i = 10; break;
  default: Print("hello");
}
```

   The expression in the **`switch`** is allowed to be any expression (as part of later semantic analysis, we could verify that it is of integer type). Unlike in C, the curly braces after the expression are mandatory. The **`case`** labels must be compile-time integer constants. Unlike in C, it is required that there is at least one non-**`default`** case statement in any **`switch`** statement. If there is a **`default`** case it must be listed last. A case contains a sequence of statements, possibly empty. If empty, control just flows through to the next case. You will need to modify the scanner and parser, and add new parse tree nodes for this new construct.

3. Error handling. If the input is not syntactically valid, the default behavior of a **`bison`**-generated parser is to call **`yyerror`** to print a message and halt parsing. In the starter code, we have provided a replacement version of **`yyerror`** that attempts to print the text of the current line and mark the first troubling token using the **`yylloc`** information recorded by the scanner like this:

```
*** Error line 18.
class Cow extends Animal extends Object {
                         ^^^^^^^
*** syntax error
```

This makes for a little clearer error-reporting, but just giving up at the first error is not desirable behavior for a compiler, so your task is add in some error-handling.

Before you try implementing anything, we suggest your first experiment with your favorite compiler (**`javac`**, **`csc`**, **`g++`**, etc.) with a critical eye to see how parse errors are handled. Feed it some syntactically incorrect code and evaluate how good a job it does at clearly and accurately reporting the problem. Is the error wording clear and specific? Is it helpful? Is it obvious how the compiler continues from there? What kind of syntax errors can it recover from gracefully and what errors really trip it up?

Given you are now writing your own compiler, you have a chance to personalize it to handle your common mistakes! Which keywords do you often misspell, what particular differences are there between Decaf and C++/Java that trip you up, how would you like these to be handled?

Pick out some simple errors that you think you can tackle and incorporate the use of **`bison`**'s **`error`** pseudo-terminal to add some rudimentary error-handling capabilities into your own parser. In your **`README`**, describe which errors you attempt to catch and provide some sample instances that illustrate the errors from which your parser can recover.

To receive full credit, your parser must do some error recovery, but our expectations for the standard requirements are quite modest. Simple recovery, say, at the statement and declaration levels, is enough. We just want you to explore and experiment with **`bison`**'s error handling. Feel free to attempt more ambitious error handling and we will credit it as an extension if it's substantial.

Testing

In the starting project, there is a **`samples`** directory containing various input files and matching **`.out`** files which represent the expected output.

Although we've said it before, we'll say it again, **the provided test files do not test every possible case**! Examine the test files and think about what cases aren't covered. Make up lots of test cases of your own. Run your parser on several incorrect files to make sure it finds their errors. What formations look like valid programs but aren't? What sequences might confuse your processing of expressions or class definitions? How well does your error recovery strategy stand up to abuse?

To make life easier, we've provided a utility script **`test-all.sh`** that will automatically run your generated compiler on all of the files in the samples directory, **`diff`**ing the

output against our reference solution. If you add your own tests with the **`.frag`** or **`.decaf`** extensions, the testing script should pick them up automatically and run them for you.

Remember that syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of the Decaf grammar. Given that our grammar is somewhat loose, some apparently nonsensical constructions will parse correctly and, of course, we are not yet doing any of the work for verify semantic validity (type-checking, declare before use, etc.). The following program is valid according to the grammar, but is obviously not semantically valid. It should parse just fine, though.

```
string binky()
{
   neverdefined b;

   if (1.5 * "Stanford")
      b / 4;
}
```

Grading

We expect your programs to accept exactly the reference grammar with the added constructs, have no parser conflicts, and have some rudimentary error recovery. We will run your program through the given test files from the samples directory as well as others of our own. We will compare (**`diff -w`**) your output to the output of our solution. This project is worth ten percent of your overall grade in the course. Most points are allocated for correctness, with the remainder reserved for the write-up of your error handling explorations in your **`README`**.

Good luck, and have fun!