

Programming Project 2: Parser

Written by Julie Zelenski, Godmar Back, and updated by David Underhill.

Due: Wednesday, July 16th at 4:00 p.m.

The Goal

You will extend your Decaf compiler (well, lexer) to handle the syntax analysis phase 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 are not responsible for verifying meaning, just structure. This project will familiarize you with the tools and give you experience in solving difficulties that arise when using them.

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 easy 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 starter files can be checked out with `svn` by issuing the following command:

```
from myth: svn co file:///usr/class/cs143/assignments/svnrepos/pp2_base
```

```
Or remotely: svn co svn+ssh://USER@myth.stanford.edu/usr/class/cs143/assignments/svnrepos/pp2_base
```

```
Or just copy the files: cp -r /usr/class/cs143/assignments/pp2_base .
```

| File | Notes |
|-------------------------|--|
| New Files | |
| <code>parser.h</code> | declares parser functions and types |
| <code>parser.y</code> | skeleton of a <code>yacc</code> parser for Decaf |
| <code>ast.*</code> | Interfaces and implementations for the Abstract Syntax Tree (AST). |
| <code>ast_type.*</code> | |
| <code>ast_decl.*</code> | |
| <code>ast_expr.*</code> | |
| <code>ast_stmt.*</code> | |

| Modified Files from PP1 (with interesting changes) | |
|--|--|
| errors.cpp | Added yyerror() method for printing an error message when a parse does not succeed. |
| list.h | Added methods to set the parent of all items in the list, or print all items in the list (using AST-specific methods). |
| main.cpp | Now handles generating the parse tree output (as well as the lexer output from PP1). |
| Makefile | Updated a few rules so it builds the new project. <i>Make sure you do "make depend" when you start - otherwise when you type "make" it will just rebuild the changed files and not those that depend on it!</i> |
| options.h | The default output is now the parse tree. You still get the lexer's output by passing the option "lexer" to dcc. |
| scanner.* | Here is our solution for PP1. We extended it to recognize the new PP2 keywords, and added functionality which saves every line as it is scanned (so the parser can output more meaningful error messages). Scan through it to see new perspective about how the scanner could be written. Feel free to use your own scanner instead of ours (you'll have to implement the <code>getLineNumbered()</code> method though). |

Your first order of business is to read through all the files to learn the lay of the land as well as absorb the helpful hints contained in the files. Most of your work will be in `parser.y`, though you will also be updating `scanner.l`, `ast_expr.*`, and `ast_stmt.*`.

Using yacc to generate the parser

- The given `parser.y` input file contains a very incomplete skeleton which you must complete to accept the correct grammar. It may help to first concentrate on getting the rules written and conflicts resolved before you add actions.
- Running `yacc` 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 `y.output` file (which is generated when you run `yacc`) - it 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 that you need to ensure is that you end up with an equivalent grammar.

- All conflicts and errors must be eliminated, i.e. you should not make use of `yacc`'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 start building the parse tree, carefully read these. 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 (including completely replacing it with something of your own design).
- We've included 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 `.cpp` 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, then come to office hours.
- 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 `yacc/bison`: accessing locations using `@n`, getting/setting attributes using `$` notation, setting up the attributes union, how attribute types are set, the attribute stack, `yyval` and `yyloc`, so on.

If the action of a production does not assign `$$`, then the default result (`$$ = $1`), is used. Be careful not to do this on accident or you will probably end up with runtime nastiness because you use some variable you never initialized.

We expect you to match our output on the reference grammar, so be sure to use look at our output. 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 `yacc`-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 `yy1loc` 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 implement anything, experiment with your favorite compiler (`gcc`, `llvm`, etc.) and observe how parse errors are handled. Feed it some syntactically incorrect code and evaluate how clearly and accurately it reports the problem. What kind of syntax errors can it recover from gracefully and what trips it up?

Pick out some simple errors that you think you can tackle and incorporate the use of `yacc`'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 parse must do some error recovery. Simple recovery at the statement and declaration levels is enough - we just want you to explore `yacc`'s error handling.

Testing

Testing works in much the same way as PP1. After you have acquired PP2's files, you should notice the `tests` directory has a new suite of tests: `samples-pp2`. Make sure you go to your `tests` directory and type "make up" to get the latest shared tests too (the rather long example test now tests your parser too). As before, type "make test" from your root project folder to run the tests.

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. 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 tco;
    if (20.07 * "Submarines")
        b / 11;
}
```

Extensions

Keep in mind that your primary goal (and what counts the most) is satisfying the base assignment requirements. *Make sure you have perfected all the required features and thoroughly tested before you work on any extensions.*

If you add something fancy, explain in your **README** file and tell us what we need to do to try it out. The default behavior of your compiler **must not** interfere with our testing of the base requirements – instead, you should provide some mechanism to turn on extensions which would otherwise interfere. This is very important as you don't want to lose credit for tests which fail simply because of interference from your extensions.

These are listed in the order in which we suggest (though not insist) you do them.

- 1) *Lexical Locations*: Carry lexical locations with your AST nodes. Like other attributes, lexical location is created from the symbol location of the terminals at the leaves. Each AST node combines location information from its children.
- 2) *The Decaf-to-C++™ compiler*: Provide an option to output C++. Decaf and C++ are syntactically similar so outputting C++ code from a Decaf AST amounts to little more than traversing the tree in appropriate order and printing the lexemes for the leaves of the tree!

Map Decaf's basic types to their same-named C++ counterparts. We provide a C++ boilerplate (`include/dcc.h`) that provides some primitives you can use (it should be included by your outputted C++ code with `"#include <dcc.h>"` ... thus your C++ code should compile with `"g++ -I/path/to/pp_base/include file.cpp"`). `dcc.h` contains macros to handle `ReadLine` and `ReadInteger` and a template class for arrays. The DECAF-TO-C++™ compiler is activated by the command-line flag `cpp`.

Generate all of the C++ output into a single `.cpp` file (this makes it easier for you, and it also makes it easier to automatically compile the output for our test cases).

Super C++ Extension:

- If you're feeling ambitious, then support the translation of any valid Decaf program. Some of the tricky cases are described below.
- Note that functions, methods, or variables must be declared before they can be used in C++ - a restriction not present in Decaf.
- Interfaces do not translate directly to C++, but you can imitate them nonetheless.
- Ok, so you have a functional translator. But can anyone read the C++ output? Can you keep the comments in appropriate spots? Can the C++ output meet our coding guidelines?

The sample file `ChessMeister.cpp` demonstrates one possible conversion from the `ChessMeister.decaf` program included in `shared_student_tests`.

We will grade this extension by running syntactically and semantically well-formed Decaf programs through your parser, compiling the output and actually running the executables.

- 3) *Improved error handling*: Designing and implementing a comprehensive error recovery strategy can be tricky, but very worthwhile in ensuring your mastery of `yacc` and its inner workings. As a pathological example, strip all the semicolons out of a valid program and run it through your parser. What does it take to make you parser able to successfully cope with this kind of input.
- 4) *Other features*. What features from other languages would you like to add to the grammar? Once you get the hang of `yacc`, adding new constructs is pretty easy. (It's the semantic checking and code generation that's the work!) You must support the command-line option `std` which tells the compiler to disable any non-standard language constructs. Because it is difficult to turn parts of your parser on and off, it is sufficient to simply report an error with `ReportError::NonStandardDecaf()` if someone tries to compile source code which utilizes our non-standard language constructs but passes the `std` flag on the command-line. Here are some ideas:
 - the C ternary operator
 - the C# `foreach` construct for arrays
 - Java's `break` and `continue` statements

Grading

Most of the grade will be allocated for correctness (85%). There will also be some consideration (15%) for design and implementation (see the “Coding Guidelines” page on the website). We will run your program with the given tests as well as others of our own. We will use `diff -w` to compare your output to the reference solutions. Credit for optional extensions is will be tracked separately and added as part of the final course grade determination.