

P2: Syntax Analysis

Due at Midnight 2/08/2023

Goal

In this programming project, you will extend your Decaf compiler 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 were found, your compiler will print the completed parse tree at the end of parsing. At this stage, you aren't responsible for verifying semantic rules, just the syntactic structure. The purpose of this project is to familiarize you with the tools and give you experience in solving typical difficulties that arise when using them to generate a parser.

There are two challenges to this assignment: the first is all about `yacc/bison`: taking your LALR 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; your job will be to fully flesh it out over the course of the semester. Our sense is that in the long run you'll be glad to have had our help, 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.

Syntactical Structure of Decaf

The reference grammar given in the Decaf language handout defines the official grammar specification you must parse. The language supports global variables and functions, classes and interfaces, variables of various types including arrays and objects, arithmetic and boolean expressions, constructs such as `if`, `while`, etc. First, read 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, you will also extend the grammar to support C-style post-increment and decrement expressions along with `switch` statements.

Starter Files

As always, be sure to read through the files we give you to understand what we have

provided. Following P2 files have been given to you (the boldface entries are the ones you will definitely modify, you may modify others as needed):

- Makefile builds project
- main.cc main() and some helper functions
- scanner.h declares scanner functions and types
- lex.yy.o object file of our lex scanner for Decaf
- parser.h declares parser functions and types
- **parser.y** skeleton of a yacc 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 list.h simple list template class
- location.h utilities for handling locations, yylloc/yyltype
- utility.h/.cc interface/implementation of various utility functions
- samples/ directory of test input files

You can use `make` to build the project. The Makefile provided will produce a parser called `dcc`. It 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& stderr
```

Since you've implemented your own `scanner.l` file in P1, this time we have given a scanner in `lex.yy.o` so you don't need to implement the scanner again. The scanner processes the comments (thus there is no need for a preprocessor in this project) and provides some additional functionalities for error handling, so please do not modify the object file. There are two objective files prepared (i.e., `lex.yy.ubuntu.o`, `lex.yy.mac.o`). Please pick the right objective file for your environment and change its name to `lex.yy.o` before `make`.

Tokens Returned from the Scanner

The given `parser.y` file already defines a set of terminal tokens to be returned from the given scanner. Some of them are the same as the tokens in P1, but there are also some new tokens. The following is a list of the new keywords and operators introduced in P2 and their corresponding token types. Items in *italic* are tokens not appeared in the reference grammar but used in this project.

Print	T_Print
ReadInteger	T_ReadInteger
ReadLine	T_ReadLine
[]	T_Dims
:	':'
++	T_Increm
--	T-Decrem
<i>switch</i>	T_Switch
<i>case</i>	T_Case
<i>default</i>	T_Default

The other keywords and operators appeared in the grammar return exactly the same token types as in P1.

Using yacc

- The given `parser.y` input file contains a very incomplete skeleton which you must complete 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; 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 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-then- 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 should be eliminated, i.e. you should not make use of yacc's automatic resolution of conflicts or use `%expect` to mask them. Note: you might 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.

Building the Abstract Syntax 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 files 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 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 can add actions to the rule as you go or 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 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 yylloc, so on. There is information on how to do these things (and much more) in the on-line references.
- Keep on your toes when assigning and using results passed from other productions in yacc. If the action of a production forgets to assign \$\$ or inappropriately relies on the default result (\$\$=\$1), you usually don't get warnings or errors, instead you are rewarded with entertaining runtime nastiness from using an unassigned variable.
- We expect you to match our output on the reference grammar, so be sure to 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 using an in-order walk. 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 two creative tasks to finish off your syntax analysis:

1) *Postfix expressions*. Add the post-increment and decrement operators similar to those in C:

```
i++;  
  
if (i == arr[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 expression (i.e. it can apply to any expression that could appear at the left-hand-side of an assignment statement). The result of a postfix expression is a normal expression so it can be placed anywhere an expression can appear.

2) *Switch statements*. Add your own production(s) 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). The case labels must be compile-time integer constants. It is required that there is at least one non-default case statement in any switch statement and 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.

To implement the above syntaxes, you will need to modify the parser and add new AST nodes for these new constructs. You can find examples for the names of these new AST nodes in the provided sample test cases.

Testing

In the starting project, there is a samples directory containing various input files and matching .out files which represent the expected output that you should match. `diff` is your friend here!

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 various incorrect files to

make sure it finds the 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?

Remember that syntax analysis is only responsible for verifying that the sequence of tokens form 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, declaring before using, etc.). The following program is valid according to the grammar, but is obviously not semantically valid. It should parse correctly.

```
string binky() {  
    neverdefined b;  
    if (1.5 * "Stanford")  
        b / 4;  
}
```

Grading

This project is worth 60 points. Most of the points will be allocated for correctness. We will run your program through the given test files from the samples directory as well as other tests of our own, using `diff -w` to compare your output to that of our solution.

Submission

Please go to <http://autograder.io> and submit your codes.