EECS665

Compiler Construction

Drew Davidson
Ruturaj Vaidya

Lecture: LEEP2 G415
MWF 3:00-3:50

Lab: Eaton 1005B

ANOUNCEMENTS | LAB | SCHEDULE | MATERIALS | ASSIGNMENTS

Project 3

Due on 10/17 11:59 PM

Accepted for 100% credit or 0 late days on 10/18 11:59 PM

Accepted for 90% credit or 1 late day on 10/19 11:59 PM

Accepted for 80% credit or 2 late days on 10/20 11:59 PM

Updates

1. The deadline has been extended to 10/17, in compensation for the project files being released a bit late
2. A free, non-portable late day credit has been given in compensation for the power outage. This effectively means that you can turn the project in on the 18th with no penalty, but that it will still not be accepted after the 20th.

Overview

For this assignment you will use the parser-generator **Bison** to write a parser for the language.

The parser will find syntax errors and, for syntactically correct programs, it will build an abstract-syntax tree (AST) representation of the program. You will also write methods to **unparse** the AST built by your parser and an input file to test your parser. A main program, P3.cpp, that calls the parser and then the unparser. You will be graded on the correctness of your parser and your unparse methods and on how thoroughly your input file tests the parser. In particular, you should write an input file that causes the action associated with every grammar rule in your Bison specification to be executed at least once.

Specifications

Getting Started

Skeleton files are provided for you below. A tarball of the files is also provided here: p3.tgz

The files are:

- lilc.l: A Flex specification for the language (similar to your submission to Project 2). Use this if there were problems with your lexer.
- lilc.yy: A Bison specification for a very small subset of the language (you will need to add to this file).
- lilc.grammar: A CFG for the language. Use this to guide the enhancements you make to .
- ast.cpp: Contains class declarations for some of the AST structure that the parser will build (you will need to add new class declarations for other AST node types).
- ast.hpp: Currently empty, but you are free to add your own auxiliary functions to ast nodes here.
- unparse.cpp: Contains example implemenetations for the unparse functions for a subset of

the AST nodes. you will need to add unparsing code to this file.

- P3.cpp: The main program that calls the parser, then, for a successful parse, calls the unparser (no changes needed). To compile P3 then run it, using test.lilc as the input, and sending the unparsed output to file test.out, type:

```
make
./P3 test.lilc test.out
```

- Makefile: A Makefile for program 3. You do not need to change this, but you may choose to do so to add test rules or to enhance your PATH. Just don't do anything that will break the build when we run it.
- test.lilc: Sample input file for the current version of the parser.
- lilc_compiler.hpp: Declaration for the manager class that serves as an interface between the lexer and parser.
- lilc_compiler.cpp: Implementation for the manager class that serves as an interface between the lexer and parser.
- lilc_scanner.hpp: As in project 2
- symbols.hpp: As in project 2

Operator Precedences and Associativities

The grammar in the file .grammar is ambiguous; it does not uniquely define the precedences and associativities of the arithmetic, relational, equality, and logical operators. You will need to add appropriate precedence and associativity declarations to your Bison specification.

- Assignment is right associative.
- The dot operator is left associative.
- The relational and equality operators (<, >, <=, >=, ==, and !=) are non-associative (i.e., expressions like a < b < c are not allowed and should cause a syntax error).
- All of the other binary operators are left associative.
- The unary minus and not (!) operators have the highest precedence, then multiplication and division, then addition and subtraction, then the relational and equality operators, then the logical *and* operator (&&), then the logical *or* operator (||), and finally the assignment operator (=).

Note that the same token (MINUS) is used for both the unary and binary minus operator, and that they have different precedences; however, the grammar has been written so that the unary minus operator has the correct (highest) precedence; therefore, you can declare MINUS to have

minus operator has the correct (highest) precedence, therefore, you can declare MINUS to have the precedence appropriate for the binary minus operator.

Bison will print a message telling you how many *conflicts* it found in your grammar. If the number is not zero, it means that your grammar is still ambiguous and the parser is unlikely to work correctly. **Do not ignore this!** Go back and fix your specification so that your grammar is not ambiguous.

Building an Abstract-Syntax Tree

To make your parser build an abstract-syntax tree, you must add new productions, declarations, and actions to .yy. You will need to decide, for each nonterminal that you add, what type its associated value should have. Then you must add the appropriate nonterminal declaration to the specification. For most nonterminals, the value will either be some kind of tree node (a subclass of ASTnode) or a LinkedList of some kind of node (use the information in ast.hpp to guide your decision). Note that you cannot use parameterized types for the types of nonterminals; so if the translation of a nonterminal is a LinkedList of some kind of node, you will have to declare its type as just plain LinkedList.

You must also add actions to each new grammar production that you add to .yy. Make sure that each action ends by assigning an appropriate value to RESULT. Note that the parser will return a Symbol whose value field contains the value assigned to RESULT in the production for the root nonterminal (nonterminal program).

Unparsing

To test your parser, you must write the unparse methods for the subclasses of ASTnode (in the file ast.cpp). When the unparse method of the root node of the program's abstract-syntax tree is called, it should print a nicely formatted version of the program (this is called *unparsing* the abstract-syntax tree). The output produced by calling unparse should be the same as the input to the parser except that:

1. There will be no comments in the output.
2. The output will be "pretty printed" (newlines and indentation will be used to make the program readable); and
3. Expressions will be fully parenthesized to reflect the order of evaluation.

For example, if the input program includes:

```
if (b == -1) { x = 4+3*5-y; while (c) { y = y*2+x; } } else { x = 0;
```

the output of unparse should be something like the following:

```
if ((b == (-1))) {
    x = ((4 + (3 * 5)) - y);
    while (c) {
        y = ((y * 2) + x);
    }
}
else {
    x = 0;
}
```

To make grading easier, put open curly braces on the *same* line as the preceding code and put closing curly braces on a line with no other code (as in the example above). Put the first statement in the body of an if or while on the line following the open curly brace. Whitespace within a line is up to you (as long as it looks reasonable).

Note: Trying to unparse a tree will help you determine whether you have built the tree correctly in the first place. Besides looking at the output of your unparser, you should try using it as the input to your parser; if it doesn't parse, you've made a mistake either in how you built your abstract-syntax tree or in how you've written your unparser.

Another good way to test your code is to try compiling the output of your unparser using the C++ compiler (g++). If your input program uses I/O (cin or cout), you will first need to add: #include <iostream> at the beginning of the file.

It is a good idea to work incrementally (see Suggestions for How to Work on This Assignment below for more detailed suggestions):

- Add a few grammar productions to .yy.
- Write the corresponding unparse operations.
- Write a test program that uses the new language constructs.
- Create a parser (using make) and run it on your test program.

Modifying ast.cpp

We will test your program by using our unparse methods on your abstract-syntax trees and by using your unparse methods on our abstract-syntax trees. To make this work, you will need to:

1. Modify ast.cpp by filling in the unparse methods and creating new node classes in ast.hpp
2. Make sure that no List field is null (i.e., when you call the constructor of a class with a LinkedList argument, that argument should never be null). Note that it is OK to make the

   ExpNode field of a ReturnStmtNode null (when no value is returned), likewise for the ExpListNode field of a CallExpNode (when the call has no arguments).
3. Follow the convention that the mySize field of a VarDeclNode has the value VarDeclNode.NOT_STRUCT if the type of the declared variable is a non-struct type and the value 0 is a struct type.

Testing

Part of your task will be to write an input file called test.lilc that thoroughly tests your parser and your unparser. You should be sure to include code that corresponds to every grammar rule in the file .grammar.

Note that since you are to provide only *one* input file, test.lilc should contain no syntax errors (you should also test your parser on some bad inputs, but don't hand those in).

You will probably find it helpful to use comments in test.lilc to explain what aspects of the parser are being tested, but your testing grade will depend only on how thoroughly the file tests the parser.

Suggestions for How to Work on This Assignment

Suggestions for How to Work on This Assignment

This assignment involves three main tasks:

1. Writing the parser specification (.yy).
2. Writing the unparse methods for the AST nodes (in ast.cpp).
3. Writing an input file (test.lilc) to test your implementation.

If you work with a partner, it is a good idea to share responsibility for all tasks to ensure that both partners understand all aspects of the assignment.

I suggest that you proceed as follows, testing your parser after each change (if you are working alone, I still suggest that you follow the basic steps outlined below, just do them all yourself):

- Working together, start by making a very small change to .yy. For example, add the rules and actions for:

  ```
  type ::= BOOL
  type ::= VOID
  ```

  Also update the appropriate unparse method in ast.cpp. Make sure that you can create and run the parser after making this small change. (To create the parser, just type make in the directory where you are working.)

- Next, add the rules needed to allow struct declarations.

- Next, add the rules needed to allow programs to include functions with no formal parameters and with empty statement lists only, and update the corresponding unparse methods.

- Still working together, add the rules (and unparse methods) for the simplest kind of expressions -- just plain identifiers.

- Now divide up the statement nonterminals into two parts, one part for each person.

- Each person should extend their own copy of .yy by adding rules for their half of the statements, and should extend their own copy of ast.cpp to define the unparse methods needed for those statements.

- Write test inputs for your statements and your partner's statements.

- After each person makes sure that their parser and unparser work on their own statements, combine the two by cutting and pasting one person's grammar rules into the other person's .yy (and similarly for ast.cpp).

- Now divide up the expression nonterminals into two parts and implement those using a similar approach. Note that you will also need to give the operators the right precedences and associativities during this step (see above).

- Divide up any remaining productions that need to be added, and add them.

- Talk about what needs to be tested and decide together what your final version of test.lilc should include.

- When working on your own, do *not* try to implement all of your nonterminals at once. Instead, add one new rule at a time to the Bison specification, make the corresponding changes to the unparse methods in ast.cpp, and test your work by augmenting your test.lilc or by writing a program that includes the new construct you added, and make sure that it is parsed and unparsed correctly.