Lab 2 Written Report

Written by:

Eric Elmoznino

Abhinav Rajaseshan

Pranit Wadkar

Documentation

All lexical analysis code resides in the scanner.l file. No additional code has been added to the starter version, other than parsing the ‘,’ token, which was not there beforehand.

Syntax parsing code all resides in parser.y. We used bison to define the grammar and parsing rules, and no additional c helper functions. The sections are divided as follows:

- token declarations
- precedence and associativity declarations
- grammar rules

When bison reduces a grammar rule, we simply print out the matching rule to the trace file in order to verify the correct behavior.

Breakdown of work

Work was broken down as evenly as possible with the following responsibilities:

- Eric: Wrote test cases with predicted outputs (success or failure, and why).
- Abhinav: Went through the grammar definition from the language and removed any ambiguity. Wrote the precedence rules for operators.
- Pranit: Wrote the bison grammar rules.
- All: Everyone contributed to debugging and the writing of this document.

Overall approach

The general approach was as follows:

- Remove any ambiguity in the grammar so that it can be processed by the bison LR paraser.

- Write the bison grammar rules. The only action that we take upon matching a rule is to output that rule to the trace file. For instance, the rule “expression : expression + expression” would be handled with yTRACE(“expression -> expression + expression”). No other actions were taken, as we were not building an AST tree for this lab. The purpose of printing out the trace was to verify that the correct rules were being matched for our various test cases.
- Write any precedence and associativity rules (e.g. %left) in order to resolve ambiguity that the grammar rules do not consider. For instance, we used these directives to specify the precedence of addition vs. multiplication.
- Write short, manageable test cases that:
    - Verify all the grammar rules that we have
    - Test corner cases (e.g. precedence of multiplication vs. addition)
    - Should fail due to syntactic errors.
- Debug issues on a case-by-case basis.

Verification of parser

As mentioned above, we verified the parser with various test cases that covered all of our grammar rules, verified corner cases, and created syntactic errors. Once the grammar was complete, we ran through these test cases one by one and debugged any issues encountered.

Challenges faced

One challenge we faced was how to handle the dangling else problem, where if we had code such as if(true){;}if(true){;}else{;} we wanted the ‘else’ to pair with the last ‘if’. In other words, there is a shift-reduce conflict because the second if statement can either be reduced alone, or the else can be shifted onto the stack first and be paired with it. Our first attempt was to simply copy the grammar rules for ‘if’ and ‘else’ statements verbatim from the lab description, and then check to see what the parser output was in the trace file during a simple test case. Running this test case, however, it seemed as if there was no ambiguity problem and the ‘else’ was already properly being paired with the last ‘if’ statement. We spent some time trying to figure out why there was no issue before realizing that the %nonassoc ELSE directive provided

in the starter file was resolving the issue because it forces the parser to resolve the shift-reduce conflict by giving priority to shifting the ELSE token rather than reducing the IF statement alone.

We had trouble with the unary minus and its precedence. Statements such as -5*3 were being resolved as -(5*3) instead of (-5)*3. We figured out that this was because our operator precedence directives specified that '*' took precedence over '-', however we only wanted this to be the case during the binary '-'. To fix the issue, we used a fake token (%nonassoc UMINUS) with high precedence and applied it to the expression rule that matches the unary minus with the %prec directive (expression : '-' expression %prec UMINUS). This forced the grammar to give precedence to the unary minus reduction over multiplication, division, etc.

The grammar defined in the lab description sheet defined "expression : expression binary_op expression", and then defined all the binary_op rules matching all operators. We figured that this could work for the current lab, but would cause issues in later ones when we had to build the AST. In our actions, we would not be able to store the resulting value of the expression because "binary_op" would not be resolved yet (e.g. we don't know if we have to do $$=$1+$3 or $$=$1*$3). We decided to address this future issue right away. To do so, we got rid of the "binary_op" nonterminal and simply created a rule in expression for every binary operator (i.e. expression : expression '*' expression | expression '+' expression | etc.). This allows us in the future to store resulting values in the AST based on the operatoration being reduced.
One of the biggest challenges was coming up with appropriate test cases. We initially tried writing large programs, but figuring out why they failed and also trying to understand the trace output was too overwhelming. Our solution was to instead write many small test cases that mostly only checked individual rules and corner cases. This simplified the task of following the output trace in relation to the program, and also made it easier quickly write and check new corner cases as we thought of them.