Topics:

1. What is a parser?
2. What does bison do?
3. Show a shift-reduce parse listing the shift and reduce operations
4. What is a shift-reduce conflict and what typically causes them?
5. What is a reduce-reduce conflict and what typically causes them?
6. Be able to draw a parse tree for a given grammar and statement
7. Be familiar with the following C++ operators that were NOT included in lang:
   1. ++val
   2. val++
8. Be able to reason through grammar changes and what their impact would be on the language, and the AST.
9. Be prepared to answer questions that show your understanding of the material covered in the labs.
10. Know what is meant by a sentential form.
11. Know what an ambiguous grammar is
12. Know what handles are
13. Know what potential handles are and how an LR(1) parser makes use of them

Sample questions:

1) Show a shift-reduce parse of 3+4\*5 given the following grammar:

start: expr

expr: expr ‘+’ term

expr: expr ‘-‘ term

expr: term

term: term ‘\*’ NUM

term: term ‘/’ NUM

term: NUM

|  |  |  |
| --- | --- | --- |
| stack | input | action |
|  | 3+4\*5 | shift |
| 3 | "+4\*5" | reduce |
| term | "+4\*5" | reduce |
| expr | "+4\*5" | shift |
| expr + | 4\*5 | shift |
| expr + 4 | \*5 | reduce |
| expr + term | \*5 | shift |
| expr + term \* | 5 | shift |
| expr + term \* 5 |  | reduce |
| expr + term |  | reduce |
| expr |  | reduce |
| start |  | accept |

2) Given the grammar for #1, after parsing the string 3+4, (where “parsing” means that the three tokens have been shifted and any reductions have taken place) if the next token was a ‘+’, should the parser shift or reduce? Why?

|  |
| --- |
| Yes, because there is no line that allows a term then a ‘+’. |

3) Given the grammar for #1, after parsing the string 3+4 (where “parsing” has the same meaning as for #2), what handles and potential handles should the parser be considering?

|  |  |
| --- | --- |
| Handle: term  P handles: \* num  / num | Handle: expr  P handles: + term  - term |

4) Given the following AST node constructors:

cAssignNode(cLval \*lval, cExpr \*expr);

cBinaryExpr(cExpr \*left, cExpr \*right, cOpNode \*op);

cIntExpr(int value);

provide the code snippet for the following production using only the constructors given above:

assign: lval ‘=’ varref PLUS\_PLUS ‘;’ {}

You can assume that varref inherits from cExpr and that PLUS\_PLUS is the “++” operator with the normal meaning.

|  |
| --- |
| assign: lval ‘=’ varref PLUS\_PLUS ‘;’  {  $$ = new cAssignNode($1, $3);  $3 = new cBinaryExpr($3, new cIntExpr(1),  new cOpNode(‘+’));  } |

5) Explain why the offsets to a function’s parameters are at a negative offset to the frame pointer but the local variables are at a **non**-negative offset.

|  |
| --- |
| The parameters are defined and pushed onto the stack by the caller, but local variables are defined by the callee. |

6) What is the difference between a syntax error and a semantic error?

|  |
| --- |
| Syntax is |

7) In our compiler, the code below would align aa at offset 0 and bb at offset 4. The “extra” bytes would be placed in Block 2. An alternative implementation would place the extra bytes in Block 1. Our language does not allow declarations after Block 2. Assuming that it did, how might this affect the design decision on where to place the extra bytes?

{ // Block 1

char aa;

{ // Block 2

int bb;

...

}

// char cc; // not allowed here in our language

...

}

|  |
| --- |
| Aa -> bb -> cc |

8) Most C compilers push arguments onto the stack from right to left. Why?

|  |
| --- |
|  |

9) Given a programming language construct from any actual programming language, what is the likelihood that it can be expressed as an LR(1) grammar? Justify your answer.

|  |
| --- |
|  |

10) A scanner typically supplies a parser with both a token type (usually a number) and a semantic value. If you are using both flex and bison, explain how each of these are passed between the scanner and the parser.

|  |
| --- |
|  |

11) Give an expression grammar for the two operators ? and # where # has higher precedence and both are right associative.

|  |
| --- |
| Expr: var ? expr | var  Var: term # var | term |

12) An LR(1) parser can handle both left recursive and right recursive grammars. Why is left recursive preferable?

|  |
| --- |
|  |

13) How does an LR(1) parser decide that there is a syntax error without shifting all the input?

|  |
| --- |
|  |

14) Draw an object diagram for the following lang program

{

array int[5] a1;

a1 var;

a1[2] = 3;

}

|  |
| --- |
|  |