Computer Science 330 Language Implementation

2001 Test Solution

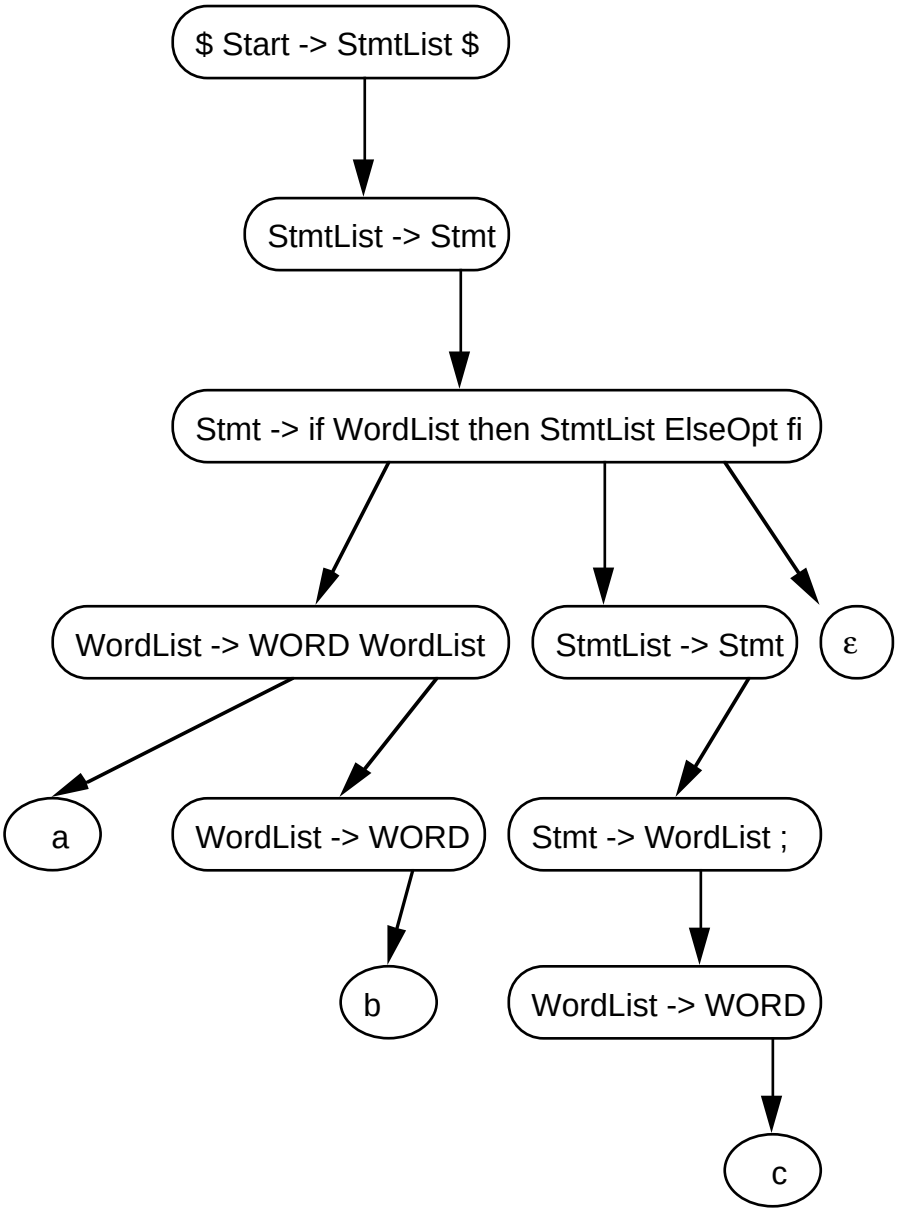
Question 1

(a) Using the information provided in the appendix, perform a shift-reduce LALR(1) parse of the input

```
if a b then c; fi
```

Stack								Input	Action
$ 0								if	Shift if 5
$ 0	if 5							WORD a	Shift WORD 1
$ 0	if 5	WD 1						WORD b	Shift WORD 1
$ 0	if 5	WD 1	WD 1					then	WL->WD
$ 0	if 5	WD 1	WL 26						WL->WD WL
$ 0	if 5	WL 9							Shift then 10
$ 0	if 5	WL 9	then 10					WORD c	Shift WORD 1
$ 0	if 5	WL 9	then 10	WD 1				;	WL->WD
$ 0	if 5	WL 9	then 10	WL 2					Shift ; 25
$ 0	if 5	WL 9	then 10	WL 2	; 25			fi	S->WL ;
$ 0	if 5	WL 9	then 10	S 3					SL->S
$ 0	if 5	WL 9	then 10	SL 11					EO->ε
$ 0	if 5	WL 9	then 10	SL 11	EO 14				Shift fi 15
$ 0	if 5	WL 9	then 10	SL 11	EO 14	fi 15		$	S->if...fi
$ 0	S 3								SL->S
$ 0	SL 6								Shift $ 8
$ 0	SL 6	$ 8						-	Reduce $S' -> SL $
$ 0	$S' -1								Accept

(b) Draw the full parse tree, showing all rules used in the above shift-reduce parse.



(c)

Nullable symbols (1 mark)

Append terminal symbols for first graph (4 marks)

Arrows for first graph (4 marks).

Compute first set from first graph (2 marks).

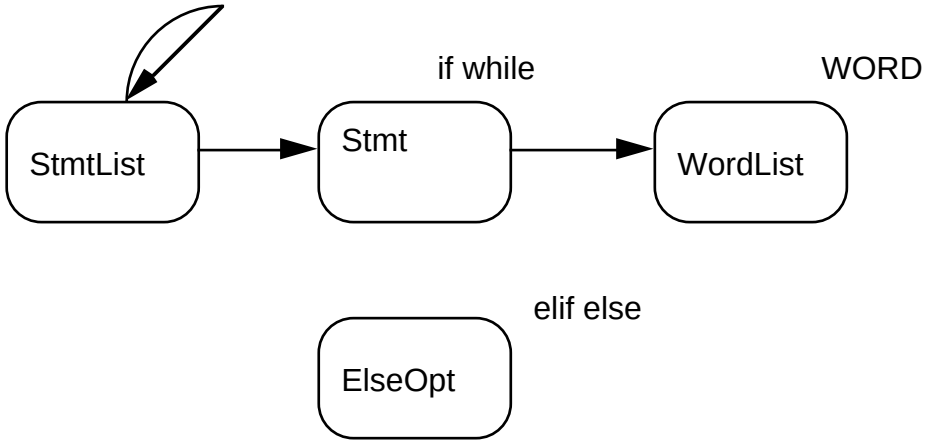
Append terminal symbols for follow graph (6 marks)

Arrows for follow graph (6 marks).

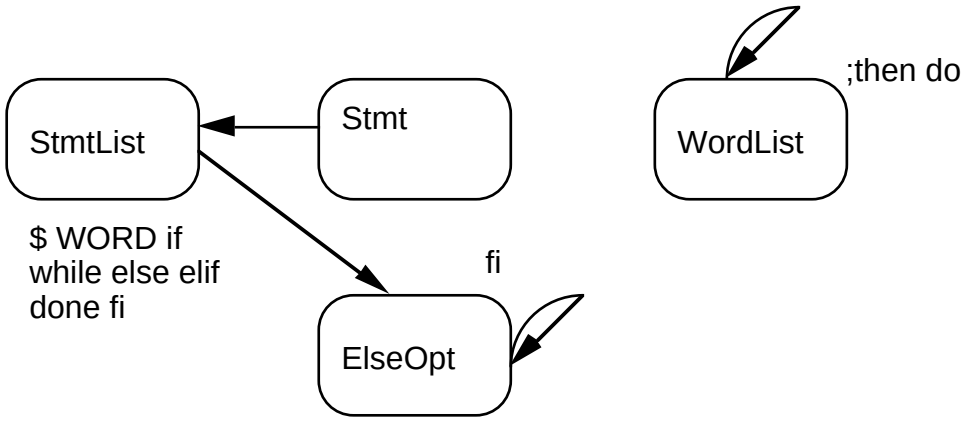
Compute follow set from follow graph (3 marks).

Base marking on whether they show understanding, not how many symbols/arrows are correct. Take into account the effects of previous errors. Take into account the extent to which their answer makes intuitive sense. For example, they should not have first or follow sets for a nonterminal that are empty.

Draw the first graph for this grammar.



Draw the follow graph for this grammar.



Indicate the which nonterminals are nullable, and the first and follow sets for the grammar

Symbol	Nullable	First Set	Follow Set
StmtList	false	if while WORD	$ WORD if while else elif done fi
Stmt	false	if while WORD	$ WORD if while else elif done fi
ElseOpt	true	elif else	fi
WordList	false	WORD	; then do

(d) Points to be considered, but not necessarily according to this marking scheme.

1 Match line breaks, and do not return anything.

1 Match blanks and tabs, and do not return anything.

2 Match comments, and do not return anything.

1 Match semicolons and return something.

1 Match words and return something.

2 Words are multi-character.

2 Include/exclude correct chars for word.

1 Return something for errors.

3 Overall program makes sense and does not include extraneous material, that looks like a rote learnt solution to the assignment or a different problem.

Don't give marks for rubbish that looks vaguely like a solution, but shows no comprehension.

Write a JLex program suitable for lexically analysing input for the above language.

```
import java.io.*;
import java_cup.runtime.*;

%%

%public
%type       Symbol
%char

%{
      public Symbol token( int tokenType ) {
            return new Symbol( tokenType, yychar,
                  yychar + yytext().length(), yytext() );
            }

%}

%init{
      yybegin( NORMAL );
%init}

%eofval{
      return token( sym.EOF );
%eofval}

InputChar         =     ([^\n\r])
SpaceChar         =     ([\ \t])
LineChar          =     (\n|\r|\r\n)
Word              =     ([^\x00-\x20\x7f\;\#]+)

%state NORMAL, ERROR
%%

<NORMAL>if                    { return token( sym.IF ); }
<NORMAL>then                  { return token( sym.THEN ); }
<NORMAL>elif                  { return token( sym.ELIF ); }
<NORMAL>else                  { return token( sym.ELSE ); }
<NORMAL>fi                    { return token( sym.FI ); }
<NORMAL>while                 { return token( sym.WHILE ); }
<NORMAL>do                    { return token( sym.DO ); }
<NORMAL>done                  { return token( sym.DONE ); }

<NORMAL>";"                   { return token( sym.SEMICOLON ); }
```

```
<NORMAL>"#"{InputChar}*        { }

<NORMAL>{LineChar}             { }
<NORMAL>{SpaceChar}            { }

<NORMAL>{Word}                 { return token( sym.WORD ); }

<NORMAL>.                      { yybegin( ERROR ); return token( sym.ERROR ); }
<ERROR>{LineChar}              { yybegin( NORMAL ); }
<ERROR>.                       { }
```

(e) Indicate how a parse tree could be built, and a treewalk of the parse tree performed for the above grammar, by:

(i) Adding in actions, etc for the rules for ElseOpt. (4 marks)

Solution must relate to this specific question, and not something memorised for another problem.

1 Use of labels on nonterminals for children.

1 Use of RESULT = …

1 Use of constructor.

1 Passing of children as parameters.

```
ElseOpt::=
            /* Empty */
            {:
            RESULT = new EmptyElseOptNode();
            :}
      |
            ELIF WordList:wordList THEN StmtList:stmtList ElseOpt:elseOpt
            {:
            RESULT = new ElifNode( wordList, stmtList, elseOpt);
            :}
      |
            ELSE StmtList:stmtList
            {:
            RESULT = new ElseOptNode( stmtList );
            :}
      ;
```

(ii) Indicating the code in the main program to perform the parse, create the parse tree, then invoke a method to perform a treewalk of the parse tree. (4 marks)

Should show

1 creation of lexical analyser.

1 Creation of parser.

1 Invocation of parse(), and extraction of value.

1 Invocation of treewalking code.

Don't care about opening files, catching exceptions, etc.

Must be compatible with this grammar.

```
import java.io.*;
import java_cup.runtime.*;

public class Main {

      public static void main( String[] argv ) {
            try {
                  if ( argv.length != 1 )
                        throw new Error( "Usage: java Main filename" );
                  FileInputStream fileInputStream =
                        new FileInputStream( argv[ 0 ] );
                  RandomAccessFile sourceFile =
                        new RandomAccessFile( argv[ 0 ], "r" );
                  Yylex lexer = new Yylex( fileInputStream );
                  parser p = new parser( lexer, sourceFile );
                  StmtListNode stmtList = p.parse().value;
                  stmtList.treeWalk();
                  }
            catch ( Exception e ) {
                  System.out.println( "Exception" );
                  }
            }
```

```
}
```

Question 2

Write a CUP grammar to parse any sequence of variable declarations. You may assume that grammar rules have been provided for expressions. You do not have to write any actions.

(25 Marks)

```
terminal
    COMMA, SEMICOLON, EQ, LEFTSQ, RIGHTSQ;
terminal String
   IDENT;
non terminal
      DeclarationSequence, Declaration, Type, InitDeclrSeq, InitDeclr, Expr;

start with DeclarationSequence;

DeclarationSequence::=
            Declaration
      |
            DeclarationSequence Declaration
      ;
Declaration::=
            Type InitDeclrSeq SEMICOLON
      ;
Type::=
            IDENT
      |
            Type LEFTSQ RIGHTSQ
      ;
InitDeclrSeq::=
            InitDeclr
      |
            InitDeclrSeq COMMA InitDeclr
      ;
InitDeclr::=
            IDENT
      |
            IDENT EQ Expr
      ;
```