Computer Science 330 Language Implementation Test Solution

Question 1 25 Marks

Write JFlex regular expressions to match the following tokens. You may declare support regular expressions if you need them.

(a) An identifier, composed of an upper case letter, followed by 0 or more letters or digits. For example, `Happy2`, but not `sad1`.

```
[A-Z][A-Za-z0-9]*
```

(2 marks)

(b) An identifier, composed of a sequence of one or more letters, followed by 0 or more digits. For example, `base`, `base16`, but not `base16Number`.

```
[A-Za-z]+[0-9]*
```

(2 marks)

(c) An identifier, composed of a single alphabetic letter, followed by an optional single digit. For example, `I`, `U2`, but not `Eye`, `You2`, `U24`, `UB40`.

```
[A-Za-z][0-9]?
```

(2 marks)

(d) A hexadecimal integer, with a prefix of `0x` or `0X`, where all digits have to agree with the case of the prefix. For example, `0x123456789abcdef`, or `0X123456789ABCDEF`, but not `0x123456789AbCdEf`, or `0x123456789ABCDEF`.

```
0x[0-9a-f]+|0X[0-9A-F]+
```

(2 marks)

(e) An identifier, composed of one or more words, with "_" characters in between, where a word is an upper case letter, followed by 0 or more lower case letters. For example, `The_Cat_Is_Hungry`, but not `The_Cat_Is_`, or `The_cat`, or `TheCat_IsHungry`.

```
word = [A-Z][a-z]*
{word}(_{word})*
```

(6 marks)

(f) A decimal integer, between `0` and `255`, without unnecessary leading "`0`"s. For example, `0`, `1`, `98`, `128`, `254`, `255`, but not `01`, `256`, `330`.

```
[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]
```

(6 marks)

(g) Indicate five different kinds of errors in the following fragment of JFlex code. ("..." just means omitted code). Assume spaces and line breaks are not syntactically important.

```
...
newline     =     \r|\n|\r\n
space       =     [\ \t]
ident       =     ...
%state NORMAL, COMMENT
%%
<NORMAL> {
      {ident}     { return token( sym.IDENT ); }
      if          { return token( sym.IF ); }
      else        { return token( sym.ELSE ); }
      ...
      /*          { yybegin( NORMAL ); }
      {space}     { return token( sym.SPACE ); }
      {newline}   { linecount++; }
      .           { }
      }
<COMMENT> {
      */          { yybegin( NORMAL ); }
      .           { }
      {newline}   { linecount++; }
      }
```

1 {ident} should be after the keywords
2 /* and */ should be quoted.
3 "/*" should have action yybegin(COMMENT);
4 {space} should have no action
5 . in <COMMENT> should be the last rule.
6 . in <NORMAL> should return error.
7 Should return token for <<EOF>>.

(5 marks)

Question 2 **55 marks**

Consider the CUP grammar in the appendices. The terminal symbols LEFTBRACE, RIGHTBRACE, ASSIGN, SEMICOLON, COMMA correspond to "{", "}", "=", ";", ",", and IDENT corresponds to an identifier.

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

```
int a = { b, c }, d;
```

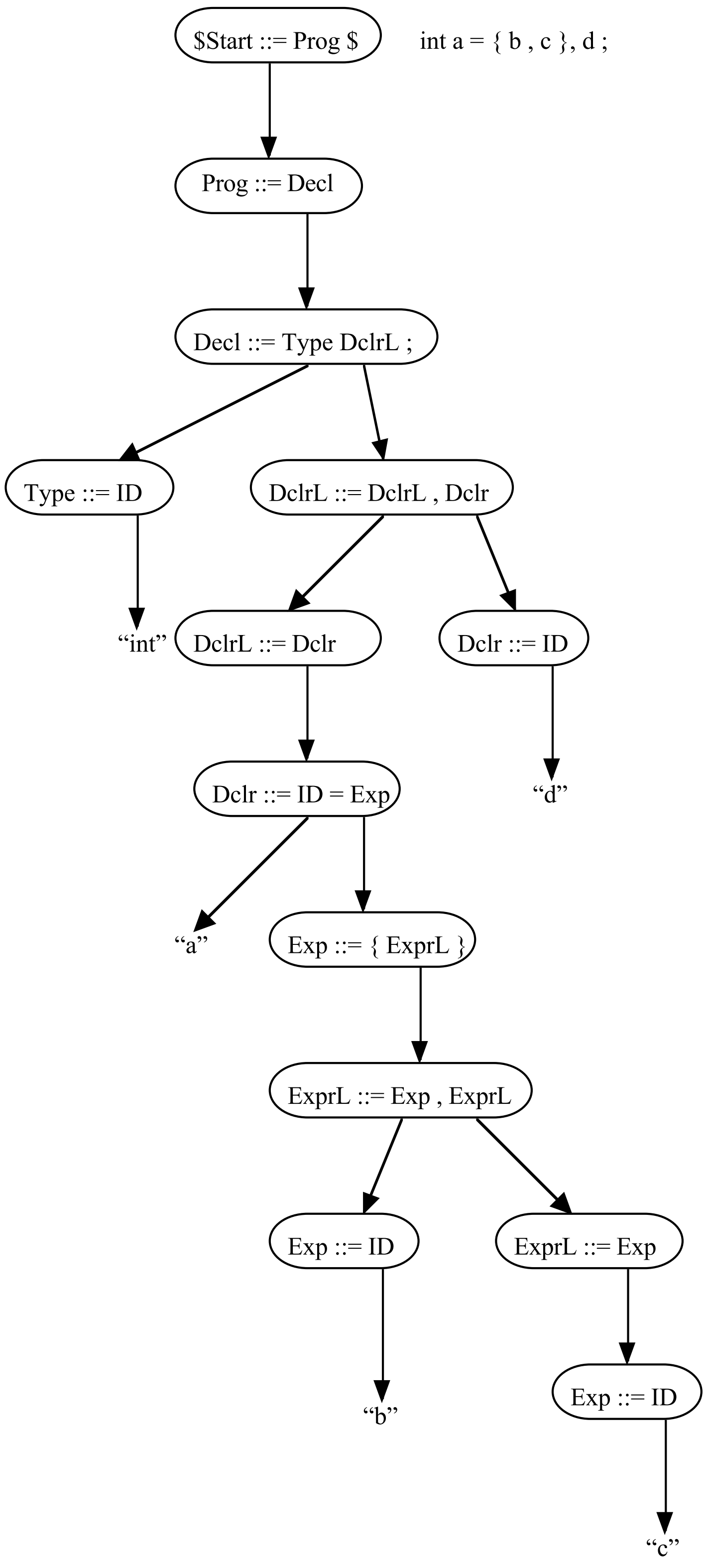
Show both the symbols and states on the stack, the current token, and the action performed at each stage. (Consider "int" to be an IDENT).

Note that some grammar rules are left recursive, while others are right recursive. (20 marks)

Stack								Input	Action	
$0								ID int	Shift	ID 4
$0	ID 4							ID a	Reduce	Type ::= ID
$0	Type 2								Shift	ID 8
$0	Type 2	ID 8						=	Shift	= 14
$0	Type 2	ID 8	= 14					{	Shift	{ 15
$0	Type 2	ID 8	= 14	{ 15				ID b	Shift	ID 17
$0	Type 2	ID 8	= 14	{ 15	ID 17			,	Reduce	Exp ::= ID
$0	Type 2	ID 8	= 14	{ 15	Exp 20				Shift	, 21
$0	Type 2	ID 8	= 14	{ 15	Exp 20	, 21		ID c	Shift	ID 17
$0	Type 2	ID 8	= 14	{ 15	Exp 20	, 21	ID 17	}	Reduce	Exp ::= ID
$0	Type 2	ID 8	= 14	{ 15	Exp 20	, 21	Exp 20		Reduce	ExpL ::= Exp
$0	Type 2	ID 8	= 14	{ 15	Exp 20	, 21	ExpL 22		Reduce	ExpL ::= Exp , ExpL
$0	Type 2	ID 8	= 14	{ 15	ExpL 18				Shift	} 24
$0	Type 2	ID 8	= 14	{ 15	ExpL 18	} 24		,	Reduce	Exp ::= { ExpL }
$0	Type 2	ID 8	= 14	Exp 16					Reduce	Dclr ::= ID = Exp
$0	Type 2	Dclr 10							Reduce	DclrL ::= Dclr
$0	Type 2	DclrL 9							Shift	, 11
$0	Type 2	DclrL 9	, 11					ID d	Shift	ID 8
$0	Type 2	DclrL 9	, 11	ID 8				;	Reduce	Dclr ::= ID
$0	Type 2	DclrL 9	, 11	Dclr 13					Reduce	DclrL ::= DclrL , Dclr
$0	Type 2	DclrL 9							Shift	; 12
$0	Type 2	DclrL 9	; 12					$	Reduce	Decl ::= Type DclrL ;
$0	Decl 5								Reduce	Prog ::= Decl
$0	Prog 1								Shift	$ 25
$0	Prog 1	$ 25						$	Reduce	$Start ::= Prog $
$0	$Start -1								Accept	

(b) Draw the full parse tree, showing all rules used in the above shift-reduce LALR(1) parse.

(5 marks)



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

```
        int a { b, c, d };
```

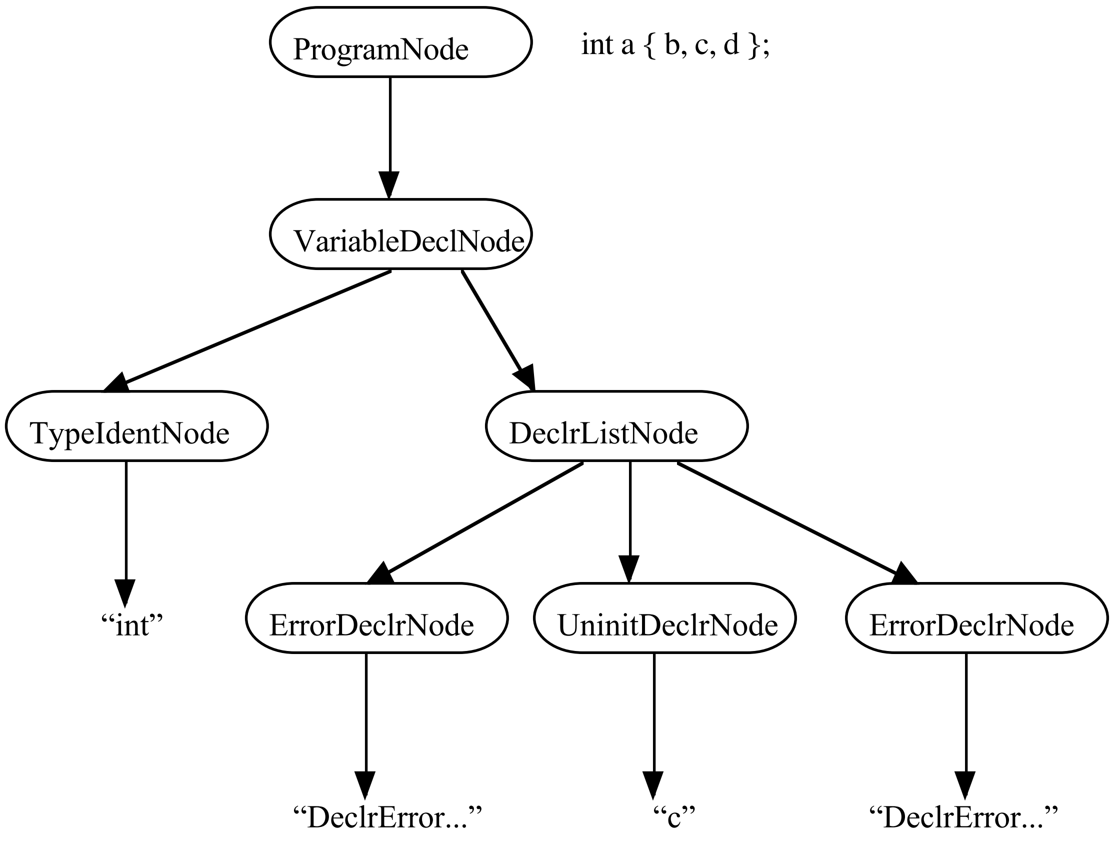
giving a clear indication of how CUP processes the input in the event of an error. (20 marks)

Stack				
$0				
$0	ID 4			
$0	Type 2			
$0	Type 2	ID 8		
$0	Type 2			
$0	Type 2	error 7		
$0	Type 2	error 7		
$0	Type 2	error 7		
$0	Type 2	Dclr 10		
$0	Type 2	DclrL 9		
$0	Type 2	DclrL 9	, 11	
$0	Type 2	DclrL 9	, 11	ID 8
$0	Type 2	DclrL 9	, 11	Dclr 13
$0	Type 2	DclrL 9		
$0	Type 2	DclrL 9	, 11	
$0	Type 2	DclrL 9	, 11	ID 8
$0	Type 2	DclrL 9	, 11	
$0	Type 2	DclrL 9	, 11	error 7
$0	Type 2	DclrL 9	, 11	error 7
$0	Type 2	DclrL 9	, 11	Dclr 13
$0	Type 2	DclrL 9		
$0	Type 2	DclrL 9	; 12	
$0	Decl 5			
$0	Prog 1			
$0	Prog 1	$ 25		
$0	$Start -1			

Input	Action	
ID int	Shift	ID 4
ID a	Reduce	Type ::= ID
	Shift	ID 8
{	Error	Pop Stack
	Error	Shift error
	Error	Consume {
ID b	Error	Consume ID b
,	Reduce	Dclr ::= error
	Reduce	DclrL ::= Dclr
	Shift	, 11
ID c	Shift	ID 8
,	Reduce	Dclr ::= ID
	Reduce	DeclrL ::= DeclrL , Declr
	Shift	, 11
ID d	Shift	ID 8
}	Error	Pop Stack
	Error	Shift error
	Error	Consume }
;	Reduce	Dclr ::= error
	Reduce	DeclrL ::= DeclrL , Declr
	Shift	; 12
$	Reduce	Decl ::= Type DclrL ;
	Reduce	Prog ::= Decl
	Shift	$ 25
$	Reduce	$Start ::= Prog $
	Accept	

(d) Draw the abstract syntax tree, as specified by the actions associated with the rules. Assume DeclrListNode is composed of a vector, containing all the children.

(5 marks)



(e) Write Java code to implement an ExprListNode, with a toString() method to reprint the list, taking into account that the tail might be null.

Assume ExprListNode is used to create a right recursive linked list, with the same structure as the grammar rule.

```
package node.exprNode;
import node.*;

public class ExprListNode extends Node {

      private ExprNode head;
      private ExprListNode tail;

      public ExprListNode( ExprNode head, ExprListNode tail ) {
            this.head = head;
            this.tail = tail;
            }

      public String toString() {
            if ( tail == null )
                  return "" + head;
            else
                  return head + ", " + tail;
            }
      }
```

Question 3 **20 marks**

Suppose we have a language for writing down expressions involving integer constants, simple variables and function invocations (in a Java style, with 0 or more comma separated actual parameters, enclosed in parentheses, even if there are 0 parameters). For example, we can write expressions of the form:

```
a( 3, b(), c, d( 4, 5 ), e( f, g( 6 ) ) )
```

Moreover, we can follow the expression by an optional “where declaration list”, which defines the meaning of some of the variables and functions, in a comma separated sequence of declarations. For example, we may define:

```
a( 3, b(), c, d( 4, 5 ), e( f, g( 6 ) ) )
      where
            b() = 4,
            c = 7,
            d( x, y ) = x,
            e( x, y ) = d( y, x ),
            g( x ) = h( x, 1, d( x, x ) )
```

The left hand side of a declaration represents the variable or function being defined, together with its formal parameters, in the case of a function, and the right hand side represents its value, in terms of other variables and functions, and the formal parameters of the function being declared. Note that the variables and formal parameters are identifiers, not expressions.

Presumably a “compiler” will “simplify” the above “program” into:

```
a( 3, 4, 7, 4, h( 6, 1, 6 ) )
```

by performing as many substitutions as possible.

Write a CUP grammar definition to parse input for the above language. You do not have to write any actions.

```
terminal
    IDENT, LEFT, RIGHT, EQUALS, COMMA, INTCONST, WHERE, ERROR;
nonterminal
    Program, Expr, ExprSeq, ExprSeqOpt, DefnSeq, Defn, IdentSeq, IdentSeqOpt;

start with Program;

Program::=
        Expr WhereOpt
    ;

Expr::=
        IDENT
    |
        INTCONST
    |
        IDENT LEFT ExprSeqOpt RIGHT
    ;

ExprSeq::=
        Expr
    |
        ExprSeq COMMA Expr
    ;

ExprSeqOpt::=
        ExprSeq
    |
        /* Empty */
    ;

DefnSeq::=
        Defn
        |
        DefnSeq COMMA Defn
    ;

Defn::=
        IDENT EQUALS Expr
    |
        IDENT LEFT IdentSeqOpt RIGHT EQUALS Expr
    ;

IdentSeq::=
        IDENT
    |
        IdentSeq COMMA IDENT
    ;
IdentSeqOpt::=
        IdentSeq
    |
        /* Empty */
    ;

WhereOpt::=
        WHERE DefnSeq
    |
        /* Empty */
    ;
```