CPS 140 **Project 2** **Spring 2012**

Part I Due: Thursday, March 22, 11:59pm
Part II Due: Thursday, March 29, 11:59pm
30 points

PART 1

Write a valid MOUSECAT program with at least 20 statements. The room should be no larger than 80 by 80.

PART 2

The purpose of this assignment is to write a parser for the MOUSECAT programming language (see the project 1 handout for a description of the tokens in the MOUSECAT programming language). Your program will read in a data file containing a MOUSECAT program, and will determine if it is a syntactically correct MOUSECAT program using an SLR(1) parser. In project 3 you will continue to build on this project, writing an interpretor.

The MOUSECAT programming language has a program definition (shown first) and seven types of statements:

Statement	Meaning
size i j begin *stmts* halt	program definition - defines height i and width j of the room.
cat v i j d ;	draw a cat at position (i, j) in direction d
mouse v i j d ;	draw a mouse at position (i, j) in direction d
hole i j ;	draw a hole at position (i, j)
move v ;	move the critter one space in its current direction
move v i ;	move the critter i spaces in its current direction
clockwise v ;	move the critter v 90 degrees clockwise
repeat i *stmts* end ;	execute *stmts* i times

where v is a variable, i and j are integers, d is a direction (north, south, east or west) and *stmts* represents 1 or more valid statements.

CFG for the MOUSECAT Programming Language

(1)	<Program>	→	size int int begin <List> halt
(2)	<List>	→	<Statement> ;
(3)	<List>	→	<List> <Statement> ;
(4)	<Statement>	→	cat var int int <Direction>
(5)	<Statement>	→	mouse var int int <Direction>
(6)	<Statement>	→	hole int int
(7)	<Statement>	→	move var
(8)	<Statement>	→	move var int
(9)	<Statement>	→	clockwise var
(10)	<Statement>	→	repeat int <List> end
(11)	<Direction>	→	north
(12)	<Direction>	→	south
(13)	<Direction>	→	east
(14)	<Direction>	→	west

where "var" represents a variable and "int" represents an integer. The productions are numbered.

Grammar for MOUSECAT: In shorter notation (mostly use first symbol of each variable or terminal, except use size(z), move (o), clockwise(l), end(d), halt(t)), and adding a new start symbol (P'):

(0)	P'	→	P
(1)	P	→	z i i b L t
(2)	L	→	S ;
(3)	L	→	L S ;
(4)	S	→	c v i i D
(5)	S	→	m v i i D
(6)	S	→	h i i
(7)	S	→	o v
(8)	S	→	o v i
(9)	S	→	l v
(10)	S	→	r i L d
(11)	D	→	n
(12)	D	→	s
(13)	D	→	e
(14)	D	→	w

DESCRIPTION OF YOUR PROGRAM

Given a MOUSECAT program, your task is to 1) scan the program and identify all its *parts* (or *tokens*) and 2) parse the program using an SLR parser and identify if it is syntactically correct. If it is, then produce a list of rules starting with the start symbol that will produce a rightmost derivation of the program.

Part 1 - The Scanner

The purpose of the scanner is to find the next token in your program, enter its value into a symbol table (a data structure that handles searches and insertions), and return 1) the location to the tokens value in the symbol table, and 2) a unique symbol, called the *token type*, which indicates

the type of the token. If a value already exists in the symbol table, don't reenter it. This part was done in Project 1.

Part 2 - The Parser

You are to write an SLR(1) parser to produce rightmost derivations of MOUSECAT programs. The parser (called the driver in project 1) will call the scanner whenever it needs the next token in the MOUSECAT program. The token type will be shifted onto the SLR parsing stack. The pointer to the token's value in the symbol table will be ignored for now. It will be used in project 3.

An SLR(1) Parse Table for the MOUSECAT programming language and its associated transition diagram are attached to this handout. The columns are labeled by symbols in the grammar (both terminals and nonterminals) and an end-of-string marker ($) (in this case, the end-of-string marker represents the end of a MOUSECAT program or end-of-file marker). The rows represent the state numbers from the transition diagram. Each entry in the table represents one of four actions. There may be additional information stored in the entry.

Actions:

- ERROR - The MOUSECAT program is not syntactically correct.
- ACCEPT - The MOUSECAT program is syntactically correct.
- SHIFT - Shift the input symbol (lookahead) and state number onto the stack. A state number must be stored in this table entry.
- REDUCE - Replace the righthand side (rhs) of the rewrite rule that is on top of the stack with its lefthand side (lhs). You might want to store some representation of the rule in this table entry.

The file `parsedata` contains the data file to create the SLR(1) Parse Table in this handout. You should read in this file before a MOUSECAT program to create the data entries in the parse table. The format of the file is: 1 row of headers for the terminals, 38 rows of entries, 1 row of headers for the variables, and 38 rows of entries. Each row in the table first has the number of the row, except the header rows, which have no entry. Items are separated by "&". If there is no entry in a column, then there will be two adjacent &'s.

An SLR(1) parser when applied to a string in the language it represents will produce a rightmost derivation (**in reverse order**) of the string. In order to list the rules in the order they would be used in a rightmost derivation (starting with the start symbol), the rules must be stacked. Whenever a REDUCE action is encountered in the parse table, store the rule on a rule stack (this is a different stack than the parsing stack). When the starting rule is encountered, print all the rules on the stack. Thus, for each MOUSECAT program that is syntactically correct, print the production rules that would derive the MOUSECAT program. Whenever there is an error, you do not have to dump the stack.

The parsing routine accesses a parser stack. Terminals and variables from the grammar, and state numbers can appear on this stack.

Consider the following MOUSECAT program.

```
// program 1
```

```
size 30 40
begin
      cat char 20 21 east ;
      move char 4 ;
halt
```

This MOUSECAT program can be derived by applying the following production rules (using the first letter of each variable):

RULES	DERIVATION
P → size int int begin L halt	size 30 40 begin L halt
L → L S ;	size 30 40 begin L S ; halt
S → move var int	size 30 40 begin L move char 4 ; halt
L → S ;	size 30 40 begin S ; move char 4 ; halt
S → cat var int int D	size 30 40 begin cat char 20 21 D ; move char 4 ; halt
D → east	size 30 40 begin cat char 20 21 east ; move char 4 ; halt

Note: An SLR parser will generate these rules in reverse order.

INPUT:

The format of the data file is the same as it was in project 1.

A data file consists of one MOUSECAT program. You may assume that MOUSECAT programs contain valid tokens. Sample data files are available on the CompSci 140 web page. These are not necessarily the data files that your program will be tested on. To ensure your program runs correctly, you should also create your own data files for testing.

OUTPUT:

Indicate whether or not the MOUSECAT program is syntactically correct. For each syntactically correct MOUSECAT program, you should list the production rules that form a rightmost derivation of the program, starting with the start symbol. (starting with the P rule, show the last rule found first, i.e. show the rules in the order they appear in the previous example (just the rules, you do not need to show the derivation)). If a program is not syntactically correct, then do not show the rules.

THE PROGRAM AND ITS SUBMISSION

Your program should be written in Java and compile in Eclipse. You should start by making a copy of your program from project 1. The name of the file with main for this part should be called mousecatpart2.java

Your program will be graded on style as well as content. Style will count for 20% of your grade.

Appropriate style for this course includes:

- *Modularity* - Your program should be divided into classes. Comments should describe each part of the classes.
- *Liberal use of comments* - In addition to the comment for each part of a class, each nontrivial section of code (for example a loop) should have a comment describing its purpose. Comments should not merely echo the code.

- *Readability* - Your program should use the indentation and spacing appropriately to make it easily readable. Your comments should be clearly distinguishable from the code.
- *Appropriate variable names* - Give appropriate names that describe their function for variables, methods, and classes.
- *Understandable output* - Your program should indicate its input as well as its output in a clear and readable manner. Remember, the output from your program is the only indication that it works!

The remaining of your grade is based on meeting the specifications of the assignment. If you do not get your program correctly running, for partial credit you may generate output that identifies which part of your program is correctly working. This output must also be clearly understandable or no credit will be given!

You should create a file called README that contains your name, the amount of time the project took, and anyone you received help from.

Submit Part I using Eclipse and Ambient under **project2part1** and submit part II using Eclipse and Ambient under **project2part2**.

Programs should be submitted by the due date. You should read your mail regularly after submitting your project in case the grader cannot compile your program.

LATE POLICY

Programs not submitted by the due date are penalized 10% up to three days late and 20% if four or more days late (Sunday does not count as a late day). You must meet with Prof. Rodger if your program is not turned in one week after the deadline.

LR(1) Parse Table - this is the table you will use, in file parsedata

	z	i	b	t	;	c	v	m	h	o	l	r	d	n	s	e	w	$
0	s2																	
1																		acc
2		s3																
3		s4																
4			s5															
5						s8		s9	s10	s11	s12	s13						
6				s14		s8		s9	s10	s11	s12	s13						
7					s16													
8							s17											
9							s18											
10		s19																
11							s20											
12							s21											
13		s22																
14																		r1
15					s23													
16				r2		r2		r2	r2	r2	r2	r2	r2					
17		s24																
18		s25																
19		s26																
20		s27			r7													
21					r9													
22						s8		s9	s10	s11	s12	s13						
23				r3		r3		r3	r3	r3	r3	r3	r3					
24		s29																
25		s30																
26					r6													
27					r8													
28						s8		s9	s10	s11	s12	s13	s31					
29														s33	s34	s35	s36	
30														s33	s34	s35	s36	
31					r10													
32					r4													
33					r11													
34					r12													
35					r13													
36					r14													
37					r5													

LR(1) Parse Table (continued)

	P	L	S	D
0	1			
1				
2				
3				
4				
5		6	7	
6			15	
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22		28	7	
23				
24				
25				
26				
27				
28			15	
29				32
30				37
31				
32				
33				
34				
35				
36				
37				

The next page shows a DFA and corresponding parse table for this grammar generated with JFLAP. The rows and columns in this table are organized in different order then the previous table that you will use.