**CSE 450: Translation of Programming Languages**

**Sample Exam**

These questions, along with those asked on the quizzes, are representative of the questions you will be asked on the final exam. The details of the questions will be modified, but will remain conceptually identical.

**1.** Given the regular expression: 1\* 0+ 1? 0

Circle all of the following strings that can be generated:

0 10

00 01110

1010 1100

000010 1

**2.** Design a regular expression that will identify all literal integers where the digits are in numerical order. For example, “144568889” is allowed, but “31337” is not.

**3.** Construct a regular expression that can identify an integer value between 0 and 255. Your expression should identify all of these values with no extra leading zeros and no additional characters.

Good examples: 200 36 2 0 255

Bad examples: -1 3.14 450 007 256

**4.** Design a context free grammar to describe the set of:

1. All possible sets of matched parentheses
2. All strings consisting of a series of a’s followed by a series b’s where there are always *more* a’s than b’s.
3. All possible palindromes over the alphabet Σ = {a, b, c}
4. A series of ‘a’s, followed by ‘b’s, followed by ‘c’s, where the total number of c’s equals the number of a’s plus the number of b’s. (e.g., “aaabcccc”, “abcc”, and “aaaccc” are all legal, but “abc”, “abccc”, and “aabb” are not because of the wrong number of c’s and “bacc", “acbc” and “cccaaa” are not because the letters are in the wrong order.)

**5.** Prove that the following context free grammar is ambiguous by drawing two valid parse trees for the same string of your choosing.

S → a S b

S → SS

S → ε

**6.** Given the context free grammar:

S → Ab

A → aAa | b

Draw the parse tree associated with the string: aaabaaab

**7.** Given the AST below, write a legal expression that could have produced it. You may assume that all variables have been pre-defined as val.

=

result /

\* +

abc 5 def 2

**8. (a)** Write a program in our Tubular source language that will take two random inputs from 1 to 10 (lets call them *m* and *n*) and calculate:

(*m* – *n*)! if m > n, or

(*n* – *m*)! otherwise.

Basically, find the factorial of the difference between the two numbers, and then output it. You should use while-loops to calculate factorials. (remember, 0! = 1)

**(b)** Draw the abstract syntax tree for the Tubular program you created in part (a).

**(c)** Show the Intermediate Code that would be generated from this abstract syntax tree and annotate the tree with the variables that would be passed up through it.

**(d)** Show the TubeCode Assembly output that you would expect from compiling your program.

**9.** Suppose you wish to add a new meta-type to Tubular called “dictionary”, which is created with the same syntax as an array (i.e., dictionary(*type*) ), and has two new built-in methods associated with it:

*dictname.*add(string *key*, *type entry*) and

*type* *dictname.*find(string *key*)

**(a)** What changes would you need to make to your **lexer** to allow for this? Include any regular expressions you would create for new tokens.

**(b)** How would you modify the production rules in your **parser**? Be specific about what these new rules look like. You may include any tokens you created in part (a)

**(c)** What **semantic checking** would you need to do?

(**d**) What new **Intermediate Code** commands would you want to help you with this?

**(e)** How would you deal with translating this to **TubeCode**? [hard!]

For example:

dictionary(val) my\_dict;

my\_dict.add(“one”, 1);

my\_dict.add(“seven”, 7);

print(my\_dict.find(“seven”));

should output the number “7” when executed.

**10.** Write a program in Tubular that will create an array of numbers with a random size between 10 and 19, inclusive. Next, fill each entry in the array with a random digit (0-9). Finally, output the digit at a random position in the array.

**(a)** Write out this Tubular program.

**(b)** Draw is the Abstract Syntax Tree that would be generated for this program.

(**c**) Show the Intermediate Code you expect your compiler to produce for this program.

**(d)** Show the TubeCode Assembly you expect to output from compiling this program.

**(e)** Show an example of what the virtual machine’s memory might look like after this program finished executing. You may choose any results you like for the random numbers.

**11. (a)** Given a transition table for a Deterministic Finite Automaton, draw the graph representation. The starred (\*) states are accepting states. State 1 is the start state.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Input | | |
| State | 'a' | 'b' | 'c' |
| 1 | 2 | 3 | 4 |
| 2\* | 3 | 3 | 2 |
| 3 | 4 | 2 | 1 |
| 4\* | 1 | 1 | 1 |

(b) Which of the following strings does the DFA accept?

"c", "abbc", "bba", "aaa"

**Tube Intermediate Code Overview**

**Arguments**

VAL: This argument uses a value; it can be a literal integer, a literal char, a label, or a scalar variable.

SCL: This argument must be a scalar variable (which will be written to)

ARR: This argument must be an array variable

**Instructions**

**add**  [VAL:*num1*] [VAL:*num2*] [SCL:*result*] **sub** [VAL:*num1*] [VAL:*num2*] [SCL:*result*]

**mult** [VAL:*num1*] [VAL:*num2*] [SCL:*result*] **div** [VAL:*num1*] [VAL:*num2*] [SCL:*result*]

Apply the given math operation on *num1* and *num2*, and place the answer in *result*.

**val\_copy** [VAL:*from*] [SCL:*to*] Copy the value *from* into the scalar variable *to*.

**out\_val** [VAL:*num*] Output the number represented by the argument.

**out\_char** [VAL:*char*] Output the character represented by the argument.

**test\_\*** [VAL:*num1*] [VAL:*num2*] [SCL:*result*]

Options are: **test\_less**, **test\_gtr**, **test\_equ**, **test\_nequ**, **test\_lte**, or **test\_gte**

Compare *num1* and *num2*. Set *result* argument to 0 or 1 based on if condition is false or true.

**jump** [VAL:*line*] Move the instruction pointer to *line*.

**jump\_if\_0** [VAL:*test*] [VAL:*line*] Move the instruction pointer to *line* if *test* is equal to zero.

**jump\_if\_n0** [VAL:*test*] [VAL:*line*] Move the instruction pointer to *line* if *test* is NOT equal to zero.

**random** [VAL:*max*] [SCL:*result*] Generate random number 0 to (*max* - 1) and store in *result*.

**ar\_get\_idx** [ARR:*array*] [VAL:*index*] [SCL:*result*]

Look up *index* in *array* and store its value as *result*.

**ar\_set\_idx** [ARR:*array*] [VAL:*index*] [VAL:*value*]

Loop up *index* in *array* and set its value to *value*.

**ar\_get\_size** [ARR:*array*] [SCL:*result*]

Look up the size of *array* and store it in *result*.

**ar\_set\_size** [ARR:*array*] [VAL:*new\_size*]

Resize *array* to *new\_size*, copying over those elements in common.

**ar\_copy** [ARR:*array1*] [ARR:*array2*]

Copy the contents and size of *array1* into *array2*.

**Labels**

A label is a string of alphanumeric characters, beginning with a letter that is used to reference a line number elsewhere in the code. When a label is created, it must be placed at the beginning of a line of code and it must end with a colon (‘:’). A label will typically be used to indicate the end point in a jump command.

**TubeCode Assembly Overview**

TubeCode Assembly is very similar to the intermediate code, with a handful of changes

* The array\_\* instructions are not available.
* Scalar variables are not available, but eight registers (regA through regH) take their place.
* Three new instructions are available that allow you to interact with memory. They are:

**load** [VAL:from] [REG:to] Load the value in memory position *from* into register *to*.

**store** [REG:from] [VAL:to] Store the value in register *from* into memory position *to*.

**mem\_copy** [VAL:from] [VAL:to] Copy the value in memory position *from* into memory position *to*.