7

1. **[8 points]** For each of the statements below indicate only whether it is true or false by circling TRUE or FALSE.

(a) [1 point] TRUE [✓ FALSE]

There is a one-to-one correspondence between instructions in assembly language and instructions in a high-level language.

(b) [1 point] TRUE [✓ FALSE]

The scanner analyzes the syntactic structure of a program, while the parser analyzes its lexical structure.

(c) [1 point] TRUE [✓ FALSE]

A local variable in C can be automatically initialized with a default value at compile time.

(d) [1 point] TRUE [✓ FALSE]

Subroutine closures are used in languages with shallow binding.

(e) [1 point] TRUE [✓ FALSE]

Interpreters offer better efficiency, while compilers offer better flexibility.

(f) [1 point] [✓ TRUE] FALSE −1

C belongs to the class of declarative programming languages.

(g) [1 point] [✓ TRUE] FALSE

Every `case` statement can also be written as one or more `if`...`then`...`else` statements.

(h) [1 point] TRUE [✓ FALSE]

Every logically-controlled loop can also be written as an enumeration-controlled loop.

6

2. **[8 points]** For each of the questions below indicate only one choice, that corresponds to the best answer.

(a) [2 points] Which of the following best characterizes the difference between declarative and imperative programming languages?

- ■ Declarative languages describe what to do; imperative languages describe how to do it.
- □ Declarative languages require that all variables be declared explicitly; imperative languages do not.
- □ Declarative languages are usually compiled; imperative languages are usually interpreted.
- □ Declarative languages emphasize problem solving through iteration; imperative languages emphasize problem solving through recursion.

(b) [2 points] Given a programming language and its compiler, what is the formal language accepted by the scanner?

- □ The set of all valid arithmetic expressions in the programming language
- □ The set of all valid tokens in the programming language
- ■ The set of all valid programs in the programming language −2
- □ The set of all valid variable names in the programming language

(c) [2 points] Which of the following is an error reported by the scanner in C?

- ■ A variable name that begins with a digit
- □ A missing }
- □ The use of a variable that has not been declared
- □ Dereferencing a NULL pointer

(d) [2 points] Why are tail-recursive functions useful?

- □ Because they are easier to read and write.
- □ Because they produce faster code, since their parameters are evaluated only when needed.
- ■ Because they produce faster code, since tail-recursive calls may reuse the same space on the stack, and hence do not involve *push* and *pop* operations.
- □ Because they compile faster.

8

3. [9 points] Assume two versions of the same program – one using macros, and the other using functions. In general, which one will run faster? Which one will produce a larger compiler-generated code? Why?

The program using macros will run faster, but and produce more compiler generated code because each place the macro is used it is replaced with the code that defined the macro. The program using functions will run slower and produce less compiler generated code. This is because functions are only evaluated at runtime.

−1

9

4. [9 points] Is short circuiting useful just because it is more efficient, or can it also change the program behavior (i.e., by computing different results or by avoiding runtime crashes)? If it can, write a short program whose behavior is different depending whether or not the language uses short-circuiting. If not, explain why not.

```
int x = 1;

if ((x > 0) || (3 / (x-1) == 1) {
        Some code
}
```

In a language with short circuiting this code can run with no errors and the code in the if statement will be executed. In a language without short circuiting the code crashes because the code tries to divide by zero.

9

5. **[9 points]** Write a regular expression that describes the following language: the set of strings that contain an odd number of a's, all of them adjacent, over alphabet $\{a, b\}$.

$(b^* a (aa)^* b^*)$

9

6. **[9 points]** Write a context-free grammar that describes simple function headers in C syntax. Assume that the return type and the type of formal parameters are either `int` or `float`. The following are examples of legal strings according to this grammar:

```
int f ();
float g (int x);
int h (float a, int b, float c);
```

You do not need to describe the identifiers for function names and parameter names, consider them given by the scanner as ID.

S → type ID (P) ;

P → type ID , P | type ID

type → float | int

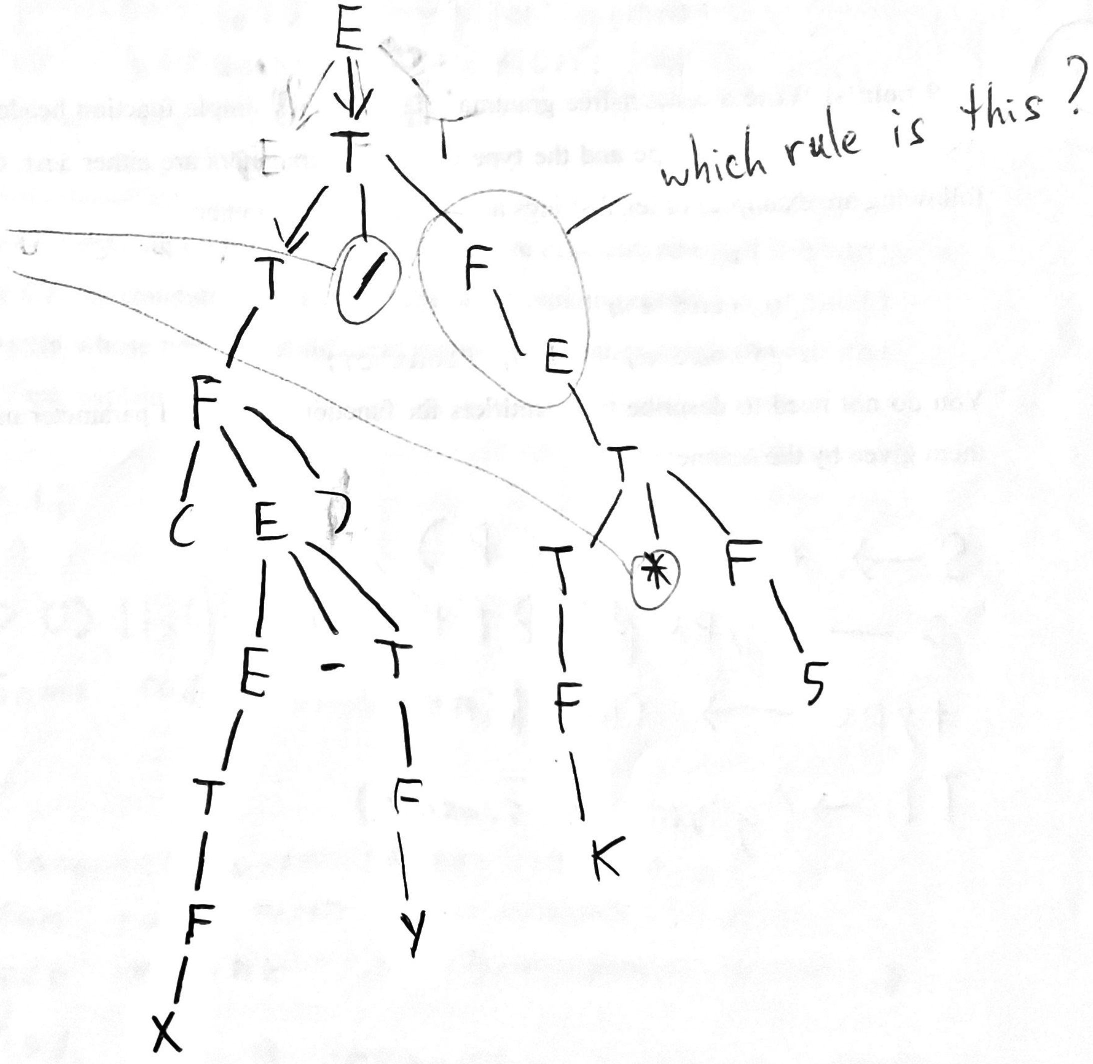
ID → (given by scanner)

8

7. **[9 points]** Consider the following unambiguous context-free grammar for arithmetic expressions:

E → E + T | E − T | T
T → T * F | T / F | F
F → identifier | number | − F | (E)

Using this grammar, show a parse tree for the following string: (x - y) / k * 5.



order

which rule is this?

−1

3

8. [9 points] Indicate the value returned by the following Scheme calls:

```
(car (cdr '(1 5 7)))              ⇒  5

(cons '(a b) '(c d))              ⇒  (a b c d)            -2

(list '(a b) '(c d))              ⇒  ( (ab) (c d))

(append '(a b) '(c d))            ⇒  (a b (c d))          -2

(define x '(a b c))
(set-cdr! x (cons 'm (cdr x)))
x                                 ⇒  (a b c d m)          -2
```

10

9. **[10 points]** Write a recursive function `(tally V L)`, which counts and returns the number of occurrences of the element `V` in the list `L`. Do not use any auxiliary variables, either global or local. The following example illustrates the use of this function:

```
> (tally 'a '(b a 7 c a a 3 a))
4
```

```
                                                        (cdr L)) 1)
( define   ( tally V L) ( cond
                          ( ( null? L) 0)
                          ( ( equal? ( car L) V) ( + ( tally V
                          ( else ( + ( tally V (cdr L)) 0)))
```

```
( define ( tally V L) ( cond
                    ( ( null? L) 0)
                    ( ( equal? ( car L) V) ( + ( tally V ( cdr L
```

7

10. [10 points] Consider the following Scheme function `f`:

```
(define (f V L)
    (cond
        ((null? L)              '())
        ((equal? V (car L))     (cdr L))
        (else                   (cons (car L)
                                      (f V (cdr L))))))
```

Indicate the value returned by the following calls:

`(f 'a '())` ⇒ ( )

`(f 'a '(c d c a f a))` ⇒ (c d c f) −1

`(f '3 '(5 (3 4) 3 7 3))` ⇒ (5 (3 4) 7) −1

Explain what this function does.

This function removes V from list L.

−1

Is function `f` tail-recursive? Justify your answer.

The function is not tail recursive because the result of the function must wait for the result of the function call to itself that it made.

10

11. [10 points] Consider the following program fragment, written in no particular language:

```
string tag = "b"                // global declaration

function print_formatted_string (string s)
     print "<", tag, ">", s, "</", tag, ">"

function emphasize (string s)
     string tag = "i"
     print_formatted_string(s)

begin                           // main program
     emphasize("hi, mom")
end
```

What does this program print if the language uses static scoping?

This prints

<b> hi, mom </b>

with static scoping.

What does it print if the language uses dynamic scoping?

This prints

<i> hi, mom </i>

with dynamic scoping

0

12. **[Extra Credit - 10 points]** Write a recursive Scheme function `(make-set L)`, which returns a set built from list L by removing duplicates, if any. You can use the predefined `member` function. Remember that the order of set elements does not matter. The following example illustrates the use of this function:

```
> (make-set '(4 3 5 3 4 1))
(5 3 4 1)
```

Do not use any imperative features of the Scheme programming language (such as `set-car!` or `set-cdr!`). Do not use any auxiliary variables, either global or local.

−10