Midterm Study Guide Solutions

Handout 15
CSCI 334: Spring 2022

Solutions

Q1. (10 points) ..... Terminology

- executable binary: An executable program written in binary machine code.
- assembly language: An intermediate form of machine code that makes binary executables easier for humans to read by way of instruction mnemonics.
- instruction mnemonic: A human-readable name for a machine code instruction.
- portability: The ability to easily move a program from one computer platform to another.
- porting: Rewriting portions of a program for a different computer platform.
- imperative language: A program that instructs a computer to perform operations step by step.
- function definition: A statement that defines a program subroutine in a programming language.
- program subroutine: A parameterized sequence of instructions.
- entry point: The initial point of execution in a program.
- return type: A program annotation denoting the type of the return value of a function.
- function body: A parameterized sequence of instructions represented by a function.
- function call: The process of applying a set of arguments to a function. This process replaces a function's parameters with the given values and executes the function body.
- program loader: The operating system facility responsible for loading a program into memory and executing it.
- program statement: A syntactic program construct that represents a single unit of work.
- compiler warning: A message emitted by a compiler identifying a program construct that has potentially unintended consequences. A compiler warning is a non-fatal message because the program is valid, but not recommended, thus the compiler succeeds in compiling it.
- make rule: A rule that relates a `make` target, dependency list, and command list.
- make target: The output of a `make` rule. The target should generally be the name of a file created by running the commands in the associated `make` rule.
- make dependency list: A set of make targets that must exist before executing the command list in a `make` rule.
- make command list: The sequence of instructions needed to generate the target.
- `make clean` rule: A rule conventionally added to a `Makefile` that restores a source code repository to its pre-build state.
- `make all` rule: A rule that builds all the targets in a `Makefile`.
- memory allocation: The mechanism by which a program obtains memory.
- variable assignment: The mechanism by which a program stores a value in a memory location (and optionally associates it with a variable name).
- primitive data type: The set of data types that are considered "atomic" for a language, meaning that their implementations are a part of the language definition itself. Primitive types form the basis for all user-defined types in a language.
- C string: A null-terminated character array.

- literal value: A value written exactly as it is intended to be interpreted. Literal values are often considered constant (i.e., unchanging) values in a programming language.
- segmentation fault: A fatal error that causes a program to abort prematurely. Segmentation faults always indicate that a program carried out an invalid memory operation.
- string interpolation: The process of substituting in values into a parameterized string.
- format string: A parameterized string used by the `printf` function to perform string interpolation before printing.
- format specifier: A special character sequence representing a string parameter that instructs the `printf` function how a stored value should be displayed during string interpolation.
- call stack: A runtime control structure that manages variable scope, function calls, and function returns. The call stack is a stack data structure consisting of stack frames, one frame for every function call.
- variable scope: The region of a program where a variable with local storage is valid.
- stack frame: The basic unit of organization in a call stack.
- storage duration: A C program annotation that denotes the "lifetime" of a block of memory.
- local storage, aka, automatic duration: An annotation that tells the C allocator that the lifetime of a storage location should be tied to the local scope. Since local scope is usually managed by the call stack in C, such memory is usally allocated within a stack frame.
- allocated duration: A C program annotation that tells the C allocator that storage lifetime is to manually managed by the programmer.
- heap memory: A region of memory managed by the memory allocator as opposed to memory managed by the runtime call stack. The heap stores values in memory with (typically) allocated duration.
- pointer: A primitive data type that represents a memory location.
- pointer dereference: A C operation that returns the value of the memory location pointed to by a pointer value.
- address-of: A C operation that returns the memory address of a variable.
- NULL dereference: An error caused by dereferencing a NULL value, which is never valid in C. On most operating systems, a NULL dereference will raise a segmentation fault. This is because `NULL` is represented by the address `0`. Address `0` is mapped to the zeroth page in memory, which is marked with a memory protection bit that raises a page protection fault (a kind of trap, usually a hardware trap from the MMU) whenever it is accessed.
- memory leak: A program error caused by a programmer's failure to correctly deallocate memory when it is no longer in use by the program.
- use-after-free: A program error that indicates a memory location is read after having been deallocated. This error is considered undefined behavior in C.
- double free: A program error caused by freeing already-freed memory.
- undefined behavior: Program behavior (semantics) that is not defined by a language specification for a given program construct. A compiler may handle undefined behavior any way the compiler implementors choose, including discarding the instructions generated by the user's code.
- dangling pointer: A pointer that points to freed or otherwise invalid memory.
- call-by-value: A form of parameter passing in which values are passed to functions by copying them into a new stack frame before transferring control to the called function.
- instruction pointer: A pointer to the current instruction being executed by the computer.
- function preamble: The routine that allocates storage for and initializes a stack frame before transferring control to the called function.

- function epilogue: The routine that deallocated storage for a stack frame and transfers control back to the calling function.
- computability: The study of what a computer is capable of doing in principle.
- function: A mathematical relation between two sets, conventionally called the domain (i.e., the set of inputs) and the codomain (i.e., the set of possible outputs).
- computable function: A function that can be expressed in a Turing complete language such as the lambda calculus. Such functions are said to be "computable in principle" even if they have undesirable computational complexity (i.e., run time).
- syntax: The lexical structure or "surface appearance" of a language, particularly a programming language.
- semantics: The logical structure or meaning of a language, particularly a programming language.
- grammar: A set of rules that define the lexical structure of a sentence for a language.
- sentence: An instance of an expression in a grammar. For example, a Java program can be thought of as a sentence in the Java language as defined by the Java grammar.
- Backus-Naur form: A metalanguage for formally expressing a language's grammar. BNF constructs grammars out of terminals and non-terminals, composing them into relations called production rules.
- terminal: A primitive value in Backus-Naur form that represents concrete syntax such as literal characters. A valid sentence in a given language is composed entirely of terminals.
- non-terminal: An abstract placeholder for a set of possible terminals. Since valid sentences in a given language cannot contain non-terminals, such placeholders must be expanded by replacing them with the appropriate terminals according to the language's grammar.
- production rule: A rule that states how a non-terminal is to be expanded in terms of terminals and, possibly, additional non-terminals. Non-terminals may be recursive.
- expansion: The process of substitution by which non-terminals are replaced with terminals in a sentence.
- parsing: The process of recognizing whether a given sentence is a member of a given language.
- parser: A program that recognizes whether a given sentence is a member of a language. Real-world parsers often produce a parse tree when a sentence is a member instead of simply returning a `true`/`false` value.
- parse tree: A tree data structure that reveals the structure, syntactically or semantically, of a given sentence with respect to a given language.
- derivation tree or syntax tree: A tree data structure that reveals the structure, by way of a grammar's production rules, of a given sentence with respect to a given language.
- abstract syntax tree: A tree data structure that reveals the semantics, or meaning, of a given sentence with respect to a given language.
- ambiguous grammar: A grammar that may be used to parse (recognize) a given sentence in more than one way. Ambiguous grammars are undesirable from the standpoint of programming languages because multiple interpretations sometimes imply that the same sentence may have multiple semantics (meanings).
- precedence: A rule that prioritizes choices in grammar productions. Precedence is often used to preserve conventional interpretations of the order of operations for a given expression in the resulting abstract syntax. Precedence is usually expressed numerically. Precedence is only necessary for ambiguous grammars.
- associativity: A tie-breaking rule that prioritizes choices in grammar productions for choices that have the same precedence. A given grammar construct is either left associative, meaning that terms group to the left, or right associative, meaning that terms group to the right. Associativity is not necessary in an unambiguous language, such as a language where all expressions are parenthesized (for example, Lisp).

- lambda calculus: A small formal language designed to explore theoretical ideas about computability. The lambda calculus can be thought of as a minimal programming language.
- variable: A placeholder for a value. In the lambda calculus grammar, variables are represented by the production `<var>`.
- abstraction: A primitive form representing a mathematical function, particularly in the lambda calculus. Abstractions are written λ`<var>.<expr>` in the lambda calculus grammar.
- application: A primitive form representing a function call using an argument, as in"the application of an argument to a function". Applications are written as `<expr><expr>` in the lambda calculus grammar.
- prefix form: A form in which operations are written to the left of operands, for example, the expression `+ a b`. Prefix form is not ambiguous, however it is conventionally enclosed by parentheses to make it clearer for humans to read, such as the expression `(+ a b)`.
- evaluation: The process of following rules that convert an expression (e.g., a sentence or program in a language) into a value.
- interpreter: A program that evaluates an expression (e.g., a sentence or program in a language).
- term rewriting: A form of evaluation based on substitution rules. The lambda calculus is a term rewriting system.
- equivalence: A logical relation between two entities that states that they are equal with respect to a given property or properties.
- lexical equivalence: An equivalence relation that states that two sentences are equal if and only if they contain exactly the same sequence of characters.
- substitution operator: A lexical convention representing a substitution operation. Substitution is conventionally written as $[y/x]$`<expr>` where x and y are abstract variables representing the given variables. The notation is read as "y replaces x in `<expr>`."
- alpha equivalence: An equivalence relation that states that two sentences are equal if and only if they are lexically equivalent after applying an alpha reduction. Alpha equivalence is written as `<expr>` $=_\alpha$ `<expr>`.
- alpha reduction: A substitution rule that renames bound variables. Formally, the rule states λx.`<expr>` $=_\alpha$ $\lambda y.[y/x]$`<expr>` where x and y are abstract variables representing the given variables. Alpha reduction is a recursive process that terminates either when the given variable x is itself redefined inside `<expr>` or when no instances of x remain.
- beta equivalence: An equivalence relation that states that two sentences are equal if and only if they are lexically equivalence after applying a beta redution. Beta equivalence is written as `<expr>` $=_\beta$ `<expr>`.
- beta reduction: A substitution rule that carries out function application. At a high level, beta reduction consists of replacing a bound variable with that of an argument. The process is carried out recursively until either the bound variable is redefined by a nested abstraction or no bound variables remain. Formally, the rule states $(\lambda x.$`<expr>`$)y =_\beta [y/x]$`<expr>`.
- bound variable: A variable as defined by an abstraction. Unlike a free variable, whose value is not defined by an expression, the value of a bound variable is equivalent to the given value in an application.
- free variable: A variable whose meaning is not defined by an expression. Unlike bound variables, free variables cannot be α-reduced.
- normal form: An expression that contains no reducible expressions, or redexes.
- reducible expression or redex: An expression that is β-equivalent to a simpler expression.
- confluence: The property of term rewriting system such that multiple different rewritings yield the same normal form. The lambda calculus is confluent.

- reduction order: An algorithm for applying reductions to a lambda expression. Since the lambda calculus is confluent, reduction is non-deterministic. Reduction orders make reduction deterministic in the lambda calculus (in other words, they remove choice).
- normal order reduction: An algorithm that always chooses the leftmost, outermost β reduction.
- applicative order reduction: An algorithm that always choose the rightmost, innermost β reduction.

Q2. (10 points) ..... Memory Leak

Does the following program leak memory? Why or why not?

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

int main() {
    int *i = malloc(sizeof(int));
    if (!i) {
        printf("Out of memory!\n");
        exit(1);
    }
    *i = 0;

    while(true) {
        printf("i: %d\n", (*i)++);
    }

    return 0;
}
```

The program does not leak memory because the allocated memory is always in use: the `while` loop never terminates. If it were possible for the program to exit the loop (it is not), then yes, technically, the program would leak memory. However, at that point a leak is not really an issue, because the program immediately terminates. Cleanup is then handled by the operating system.

Q3. (20 points) ..... Church Numerals

Subtraction by one can be achieved using the pred function.

$$\texttt{pred} \equiv \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u)$$

Prove that 1 - 1 = 0.

The following is a normal order reduction.

$(\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))(\lambda u.x)(\lambda u.u))(\lambda f.\lambda x.f\ x)$	means `(pred 1)`
$(\lambda n.\lambda a.\lambda b.n(\lambda g.\lambda h.h(g\ a))(\lambda u.b)(\lambda u.u))(\lambda f.\lambda x.f\ x)$	α-reduce a for f and b for x
$(\lambda a.\lambda b.(\lambda f.\lambda x.f\ x)(\lambda g.\lambda h.h(g\ a))(\lambda u.b)(\lambda u.u))$	β-reduce $(\lambda f.\lambda x.fx)$ for n
$\lambda a.\lambda b.(\lambda f.\lambda x.f\ x)(\lambda g.\lambda h.h(g\ a))(\lambda u.b)(\lambda u.u)$	remove parens
$\lambda a.\lambda b.((\lambda f.\lambda x.f\ x)(\lambda g.\lambda h.h(g\ a)))(\lambda u.b)(\lambda u.u)$	add parens to make available redex easier to see
$\lambda a.\lambda b.((\lambda x.(\lambda g.\lambda h.h(g\ a))x))(\lambda u.b)(\lambda u.u)$	β-reduce $(\lambda g.\lambda h.h(ga))$ for f
$\lambda a.\lambda b.(\lambda x.(\lambda g.\lambda h.h(g\ a))x)(\lambda u.b)(\lambda u.u)$	remove parens
$\lambda a.\lambda b.((\lambda g.\lambda h.h(g\ a))(\lambda u.b))(\lambda u.u)$	β-reduce $(\lambda u.b)$ for x
$\lambda a.\lambda b.((\lambda h.h((\lambda u.b)a)))(\lambda u.u)$	β-reduce $(\lambda u.b)$ for g
$\lambda a.\lambda b.(\lambda h.h((\lambda u.b)a))(\lambda u.u)$	remove parens
$\lambda a.\lambda b.((\lambda u.u)((\lambda u.b)a))$	β-reduce $(\lambda u.u)$ for h
$\lambda a.\lambda b.(\lambda u.u)((\lambda u.b)a)$	remove parens
$\lambda a.\lambda b.((\lambda u.b)\ a)$	β-reduce $((\lambda u.b)a)$ for u
$\lambda a.\lambda b.b$	β-reduce a for u
$\lambda f.\lambda x.x$	α-reduce f for a and x for b
	$\lambda f.\lambda x.x$ means 0, done

Q4. (10 points) ..... Backus-Naur Form

Add grammar support for exponentiation. For example, we should be able to parse the following expression:

$$((\lambda x.(+\ x\ 2))1)^2$$

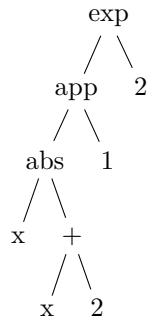
and correctly evaluate it to `9`.

We don't need to do much to add exponents. Adding the following case to `<expression>` suffices.

`<exponent> ::= <expression>`$^{\texttt{<expression>}}$

Composing `<exponent>` with the pre-existing `<expression>` non-terminal allows us to use expressions as both bases and exponents. The danger of such an expressive grammar is that we might somehow end up with an exponential expression that does not make sense. However, since the only values in our system are numbers, variables, and functions, these are all reasonable values for exponential expressions.

Out new grammar allows us to parse the above expression into the following abstract syntax tree.



Although we have not yet discussed in detail how to reduce an expression that contains non-pure lambda calculus terms (these kinds of special lambda calculus reductions are sometimes called δ-reduction), this tree should make intuitive sense: the base and the exponent are both operands. The fact that we put the base on the left and the exponent on the right is arbitrary—the correct arrangement depends on how you decide to define the `exp` abstract syntax.