Midterm Study Guide Solutions

Handout 15
CSCI 334: Spring 2024

Solutions

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

This list is not complete.

- 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 $\lambda x.$`<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.
- normal order reduction: An algorithm that always chooses the outermost, leftmost β reduction. If a normal form exists for a given lambda expression, the normal order will find it.
- applicative order reduction: An algorithm that always choose the innermost, leftmost β reduction. If a normal form exists for a given lambda expression, the applicative order may find it (if it terminates).
- mutable variable: A storage location in a program that can be changed during the runtime of a program.
- immutable variable: A storage location in a program that may not be changed during the runtime of a program.
- side effect: A program state change not reflected but the output of a program procedure.
- pure function: A program procedure that has no side effects; in other words, a true mathematical function.
- first class value: Any program value that can be stored in a variable and passed as an argument to a function.
- first class function: A function definition that can be stored in a variable and passed as an argument to a function.
- staticly typed: A programming language that performs type checking before it runs; programs that do not satisfy the type-checker are considered malformed and therefore buggy.
- expression: A line of code that returns a value.
- statement: A line of code that does not return a value.
- strongly typed: The property of a programming language that types are enforced strictly. F# is strongly typed.
- weakly typed: The property of a programming language that types are not enforced strictly (e.g., values can be cast implicitly into other values). C is weakly typed.
- REPL: A "read-eval-print loop." In other words, an interactive programming environment. `fsharpi` is a REPL.
- algebraic data type: A data type defined by cases.
- pattern matching: A language facility that supports case-by-case program logic, often with concise variable binding facilities.
- higher-order function: Any function that can take a function as a parameter.
- curried function: A function whose arguments are defined by composition of one-parameter functions. Curried functions can be partially applied.
- partial application: What is meant when a curried function is called without an argument for all of its formal parameters.
- map: A higher order function that applies a given function f to every element in a sequence.
- fold: A higher order function that applies a given function f to every element in a sequence while also maintaining an accumulator value.

Q2. (0 points) ..... Practice Reductions

Reduce each of the following expressions to their normal forms.

(a) $(\lambda x.x)(\lambda x.xx)(\lambda x.xa)$ reduces to aa

(b) $(\lambda x.x)(\lambda y.yy)(\lambda z.za)$ reduces to aa

(c) $(\lambda x.\lambda y.xyy)(\lambda a.a)b$ reduces to bb

(d) $(\lambda x.xx)(\lambda y.yx)z$ reduces to xxz

(e) $(\lambda x.(\lambda y.(xy))y)z$ reduces to zy

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 the 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:

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

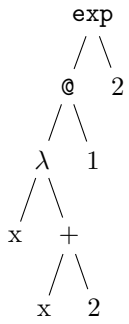
and correctly evaluate it to `9`.

We don't need to do much to add exponents. Adding an `<exponent>` 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.

Q5. (25 points) ..... Partial Application

```
let log (date: System.DateTime)(severity: string)(daemon: string)(message: string)(file: string) : unit =
    let datefmt = date.ToString("yyyy-MM-dd HH:mm")
    let msg = sprintf "[%s][%s] %s: %s" datefmt daemon severity message
    System.IO.File.AppendAllText(file, msg)

let lognow = log System.DateTime.Now

let info = lognow "INFO"

let warn = lognow "WARN"

let fail = lognow "FAIL"

let usage() =
    printfn "Usage:"
    printfn "\t$ dotnet run <severity> <daemon> <message> <file>"
    exit 1

[<EntryPoint>]
let main args =
    if Array.length args <> 4 then
        usage()

    let severity = args[0]
    let daemon = args[1]
    let message = args[2]
    let file = args[3]

    match severity with
    | "info" | "INFO" -> info daemon message file
    | "warn" | "WARN" -> warn daemon message file
    | "fail" | "FAIL" -> fail daemon message file
    | _ -> eprintfn "Unknown severity '%s'" severity

    0
```