EDAP05: Concepts of Programming Languages - Skills Sheet

Karl Hallsby

Last Edited: January 12, 2020

Contents

1 Language Critique

The book provides a convenient common framework for arguing about the benefits and disadvantages of language features and programming languages in general. Table 1.1 and, more generally, Section 1.3 in the book describe a number of axes along which we can try to understand the effect of language features and the qualities of programming languages.

1.1

You should be able to recognise and describe all the characteristics listed in Table 1.1, as well as the three basic criteria listed therein.

- Readability — How easy is it to read the program's source code and understand what's happening?
  - Simplicity — How "simple" is the language?
    - Is it like brainfuck and hard to read and write (leading to reliability issues), or like Python that is quite literate (in terms of how similar it can be to English)?
  - Orthogonality — How do the number of features present in a language overlap?
    - Operator overloading is one instance where we sacrifice orthogonality for a simpler to read and write (and more reliable) language.
    - For example, the addition operator is typically overloaded for integers and floating-point numbers, because in math we use the same operator for both, but have different meanings and required operations in computers.
  - Data Types — How closely do data types that we use in the program match our needs? Do these overlap with things that may be present in the real world (in the case of OOP languages)?
  - Syntax Design — Is the syntax, i.e. the way the language must be read and written easy or difficult?
    - If it is difficult, is there a reason?
    - This may influence the reliability of the language if it is hard to understand how the program is working.
- Writability — How easy is it to write a program's source code and understand what's happening? This does not just refer to the writer, but also the reader (who may be the same persion or someone else) later.
  - Support for Abstraction — Can we take repeated details of our program and make them more general than before?
    - For program operations (subprograms), instead of having to embed common operations everywhere in a program that may need to perform this calculation, can we make it a subprogram that can be called repeatedly and operate as before?
    - For data, how naturally can we represent more complicated things? For example, a binary tree in C is a `struct` with 2 pointers and a holding value. In older languages (FORTRAN), it was 3 arrays in parallel.
  - Expressivity — Using the features built into the language, how easy is it to **naturally** express the computation that we want to perform?
    - How much of the language's code do we need to describe some computation?
- Reliability — How sure are we (the writer) that our code is actually doing what we want, without error? If there are errors, how can we ensure that they are handled?
  - Type Checking — Using the data types from earlier, what kind of checking can we perform to ensure our program operates the way we intend it to?
    - Think of SML's type checking, or Rust's type checking for a strongly-typed and statically checked language.
    - C/C++ is weakly-typed because of pointers (which can be overwritten in any way, without regards to the type checker), although it is statically checked.
    - Python is strongly-typed, but dynamically checked. This is the reason any variable can be any type at any point of execution, but the operations that can be performed on them are limited.
    - JavaScript is weakly-typed and dynamically checked. Being weakly-typed means the language allows errors that could be caught to instead "fly under the radar", until the thing is used again. Being dynamically checked means that these are only found during runtime of the language.
  - Exception Handling — If an error happens during program execution (a file is not present, network connection lost, etc.) how well can we handle that? How easy is it to read and write a program where these can occur?
  - Restricting Aliasing — Can we restrict the number of things that point to a single memory cell? This is important for type checking and resource deallocation.

1.2

For some language feature, you should be able to recognise the characteristics and criteria (from Table 1.1 (Section 1.1)) affected by that feature's presence or absence.

1.3

For some language feature, you should be able to recognise how the feature's presence or absence affects the language, based on the three criteria from Table 1.1 (Section 1.1).

1.4

For two related language features, you should be able to compare them, based on the three criteria from Table 1.1 (Section 1.1).

1.5

For some language feature, you should be able to recognise and explain how it can affect the Cost considerations for training, compile time, execution time, and cost of poor reliability as outlined in Section 1.3.7.

- Training Time — How long does it take programmers to learn this language?
- Compile Time — How long does it take the compiler to compile the program?
  - A slower compiler may catch more errors or generate more optimized machine code
  - A faster compiler allows for quicker development and faster programmer feedback.
- Execution Time — When the program is running, how much time does it take to run?
  - This can be measured in many ways: CPU time, CPU cycles, human time, etc.
- Poor Reliability — If the program runs, but does so unreliably, it must be fixed. What is the cost of fixing the reliability issues with this/these program(s)?

2 Language Implementation

You should be familiar with the following concepts:

2.1

Language implementation via **pure interpretation**, **compilation**, and **hybrid implementation**.

- Pure Interpretation
  - Python
  - The program runs on an interpretation layer between the hardware and itself. (VM usually)
  - Quicker development with faster feedback on issues
  - No compilation (May be possible to compile down to VM-specific code, but not machine code.)
  - Slower execution time (Must interpret each command, break down to possible CPU actions, and write the machine code on-the-fly.)
- Compilation
  - C
  - Program runs ON the computer's hardware
  - Slower development, must compile for some issues to be found (Type Error). Must run for other issues to be found (Null pointer).
  - Compile down to machine code for quick runtime.
  - Fast execution. Human-readable code already in machine code, just need to load into memory and begin execution.
- Hybrid Implementation
  - Java
  - Program may start by being interpreted, but parts in use will start to be compiled while running.
  - Runs some parts through an interpreter, and some pieces are compiled.
  - Theoretically, same kind of fast development as interpreted languages, with the more frequently-used code being compiled for speed.
  - Hypothetically, Fast execution for the parts that are compiled.

2.2

Language implementation with the help of a **just-in-time compiler**.

- A Just-In-Time (JIT) compiler starts by having the code run in an interpreted mode.
- After some time, the most frequently used portions of code are compiled, to improve performance, and the interpreted parts are replaced by the compiled parts.
- Java uses this.

2.3

The concept of language **run-time system**.

- These are the systems in place that were put in by the compiler/interpreter to help the program run.
- These can include: Error handlers, resource deallocators (Garbage collectors), and type coercers.

3 Syntax

Syntax describes the possible structure (or form) of programs of a given programming language. Backus-Naur Form (BNF) grammars have emerged as the standard mechanism for describing language syntax. BNF grammars used to describe languages when communicating with language adopters and compiler implementors. There are also many tools (particularly the yacc and antlr families of programs) for automatically generating parsers, programs that recognise whether an input program matches a grammar and, if it does, execute user-defined actions upon encountering certain language constructs.

3.1

You should be able to determine whether a given property of a language is part of the syntax, of the static semantics, or of the dynamic semantics.

- Syntax — This is done with the context-free grammar and parsing the input symbols
- Static Semantics — This is done during the implementation of the compiler/interpreter. These define some of the operations possible and how they should behave.
  - For example, the `+` operator must be defined to mean addition when given integers and concatenation when given strings.
  - It is required that the syntax be defined and relatively fixed for the semantics to be well-defined.
- Dynamic Semantics — This is done while the program is running. These determine the behaviors of the operators while the program is in-use.

3.2

You should be able to read a BNF grammar and understand the difference between terminals and nonterminals.

- Terminals — Symbols that represent a terminating point in the recursion.
  - Typically, these are symbols that mean we have reached the end of our recursion.
  - In a programming language, these are usually keywords.
  - In this course, they are typically symbols that lack the $\langle$ and $\rangle$.
- Nonterminals — Symbols that continue the recursion of a production to another production with the same symbol.
  - These are things used to construct a parse tree of what we have input to the parser/grammar.
  - In this course, they are typically denoted as $\langle A \rangle$.

3.3

Given a BNF grammar, you should be able to write down examples of programs that can be generated by the grammar.

- This means you must be able to run through and recurse through the grammar.

3.4

Given a BNF grammar, you should be able to tell whether a given program can be generated by the grammar. If the program is generated by the grammar, you should also be able to generate a parse tree for the program.

- Essentially, you must run the grammar on the given program and confirm whether the program could be accepted/generated by the grammar.

3.5

You should be able to determine whether a given (small) BNF grammar is ambiguous (the problem is undecidable in general, so this skill only pertains to practically relevant examples as covered in the textbook).

- There needs to be more than one way to generate a parse tree given a grammar.

3.6

Given a BNF grammar, you should be able to determine the associativity of any operator used therein.

- Left associative operators are left-recursive in their grammar, so the parentheses build up towards the left, with the first operation to be performed deepest in the parse tree on the left, with all subsequent operations higher in the tree and to the right.
- Right associative operators are right-recursive in their grammar, so the parentheses build towards the right, with the first operation to be performed deepest in the parse tree to the right, with all subsequent operation higher in the tree and to the left.

3.7

You should be able to describe the difference between an object language and a meta-language.

- Object language — A language that is used for something, for example, C. These can be meta-languages in some cases.
- Meta-language — A language that describes other languages, Context-free grammars, for example.

3.8

Understand **arity**, **fixity**, and **precedence**, and **associativity** of operators

- Arity
  1. Unary — An operator that takes a single argument. Logical NOT is an example of this, as well as `i++`.
  2. Binary — An operator that takes 2 arguments. Most mathematical and logical operations are like this.
  3. Ternary — An operator that takes 3 arguments. The ternary `if` operator is an example of this.
- Fixity
  - Prefix — Operator symbol before arguments, $+\,3\,2 = 5$
  - Infix — Operator symbol between arguments, $3 + 2 = 5$
  - Postfix — Operator symbol after arguments, $3\,2\,+ = 5$
- Precedence
  - Given several operators in a single line, which one should have the highest priority?
  - Only way to figure this out is to run tests which force certain operations to have mutually disjoint outputs. Meaning for some inputs, you will get different outputs for operations with different precedences.
- Associativity
  - Given several operators of the same precedence, what order should they be evaluated in?
  - Most mathematical and logical operators are left-associative, meaning they should be left-recursive in their grammar.
  - The exceptions to this are exponentiation and function types, which are right-associative, and thus must be right-recursive in their grammar.

4 Natural Semantics

There are many ways to describe semantics. In this course, we focus on natural semantics (also known as Big-Step Operational Semantics).

1. You should be able to read a specification of natural semantics.
2. You should be able to undestand how **expressions** and **values** relate to each other.
3. Given a natural semantics and a parse tree (or unambiguous expression), you should be able to compute the semantics of the given program.
4. Given a natural semantics and a BNF grammar, you should be able to tell whether any parts of the semantics are undefined.
5. Given an understanding of what a state-free expression language is supposed to do, you should be able to write down a simple natural semantics for it.
6. You should be able to understand the concepts of Environments in the context of natural semantics and be able to utilise it when reading and reasoning about natural semantics.

5 Bindings and Lifetimes

1. You should understand the difference between static and dynamic type binding and be able to take advantage of either property in your programming.
2. Given a syntax, a compiler and a run-time system for a language, you should be able to determine whether the language is using static or dynamic type binding.
3. Concepts: static binding, stack-dynamic binding, explicit heap-dynamic binding, and implicit heap-dynamic binding.
4. Given a syntax, a compiler and a run-time system for a language, you should be able to determine which storage binding(s) the language is using.
5. Concept: lifetime of a variable
6. Concepts: allocation and deallocation of a heap-dynamic variable
7. Concept: binding time, especially the difference between static and dynamic binding times

6 Scoping

1. You should understand the difference between static scoping and dynamic scoping and be able to exploit either in your programming.
2. Given a language implementation, you should be able to write a program to determine whether the language uses static or dynamic scoping.
3. Concept: referencing Environment

7 Types

1. Concepts: the types of integers, floating-point numbers (floats), and decimal numbers (decimals)
2. Concept: the type of booleans
3. Concept: enumeration type
4. Concepts: character and string types
5. Concept: subrange types
6. Concept: record types
7. Concept: tuple types
8. Concept: list types
9. Concept: associative array types
10. Concept: union types, both free and discriminated
11. Concept: operator overloading
12. Concepts: strong typing and weak typing
13. Concepts: type checking and the differences between dynamic type checking, static type checking.
14. Concepts: type equivalence, including the difference between nominal and structural type equivalence
15. Concept: type constructors
16. Concept: typing rules as part of type systems, and how to read such rules and utilise them in your reasoning about program semantics
17. Concept: the type preservation property, also known as subject reduction, of type systems
18. Concepts: type parameters and parametric polymorphism
19. Concept: function types for subroutines

20. Concept: type classes
21. Concept: subtyping
22. Concept: covariance and contravariance of type parameters, and the arrow rule
23. Concept: bounded parametric polymorphism
24. Concept: definition-site variance and use-site variance of type parameters
25. Concept: Algebraic Datatypes as in Standard ML and their use in pattern-matching
26. Concept: Automatic Type Inference

8 Expressions

1. Concept: arithmetic expressions
2. Concepts: boolean expressions and relational expressions
3. Concepts: Different forms of object equality, including reference equality and structural equality
4. Concept: operand evaluation order and how it affects the outcome of programs
5. Concept: short-circuit evaluation and how it affects the outcome of programs
6. Concept: referential transparency and side effects
7. Concept: list comprehensions and their semantics
8. Concept: type coercion expressions, both explicit and implicit, including narrowing conversions and widening conversions
9. Concept: conditional expressions

9 Statements and Control Structures

1. Concept: assignment statements, including compound assignment
2. Concept: two-way selection statements
3. Concept: multiple-selection statements
4. Concept: counter-controlled loops
5. Concept: logically controlled loops
6. Concept: datastructure-controlled loops

10 Subprograms and Parameter Passing

1. Concepts: subprograms, including formal arguments and actual arguments
2. Concept: local variables in subprograms
3. Concept: nested subprograms
4. Concepts: parameter passing modes
   (a) by value
   (b) by result
   (c) by value-result
   (d) by reference
   (e) by name
   (f) by need
5. Concepts: subprograms as parameters, subprograms as return values, Closures
6. Concept: activation records

11 Pointers, References, and Arrays

1. Concepts: pointers and references
2. Concept: the dangling pointer problem
3. Concepts: garbage collection in the forms of reference counting and mark-and-sweep collection
4. Concept: arrays in the forms of
   (a) static arrays
   (b) fixed stack-dynamic arrays
   (c) fixed heap-dynamic arrays
   (d) heap-dynamic arrays

12 Abstract Datatypes

1. Concepts: information hiding and encapsulation
2. Concept: abstract datatypes
3. Concept: generic abstract datatypes, that is, abstract datatypes that take one or more type parameters

13 Object-Oriented Programming

1. Concept: object-oriented language
2. Concept: dynamic dispatch, also known as dynamic binding of methods
3. Concept: inheritance
4. Concept: method overriding
5. Concept: the combined use of static types and dynamic types in statically typed object-oriented languages

14 Functional Programming

1. Concept: local let bindings of variables
2. Concept: anonymous functions, also known as lambda expressions
3. Concept: first-class functions
4. Concepts: pattern matching, including wildcards
5. Concepts: exhaustiveness and redundancy in pattern matching
6. Concept: lists as algebraic datatypes
7. Concept: the map function
8. Concept: currying
9. Functional Programming in Standard ML

15 Exceptions and Continuations

1. Concepts: exceptions and exception handlers