CSC324

Pierre's Guide

Fall 2022

Contents

Chapter 1

Basics of Programming Languages

1.1 Syntax

The syntax of a programming language are the rules on what expressions of the programming language can look like. The *grammar* is used to specify the syntax of a programming language. These grammars are specified using *terminal symbols* and **non-terminal symbols**. Terminal symbols are the most basic symbols.

Definition 1.1.1: Terminal Symbol

Some literal symbol or collection of symbols that it not recursively defined.

Example 1.1.1 (Terminal Symbol)

Imagine a programming language with a **terminal symbol** `NUMBER`. This represents any numeric literal. Such as $3, -1.5$ or 0.

Non-terminal symbols are often used to express symbols which can be constructed from other terminal symbols and non-terminal symbols, including itself.

Definition 1.1.2: Non-Terminal Symbols

These are symbols which generate valid expressions by substituting them with other grammar rules.

Example 1.1.2 (Non-Terminal Symbol)

```
<expr> = NUMBER | '(' <expr> <op> <expr> ')'
```

In this example we define grammar for expressions which can be a number or nested expressions connected with an operation surrounded by parenthesis.

1.2 Abstract Syntax Trees

A way to represent parsing in programming languages is to use **abstract syntax trees**. This is because syntax in language is usually hierarchical. We often have basic subexpressions and expressions which recursively uses other expressions. This means that a tree structure is appropriate for representing the order of how code should be parsed:

Definition 1.2.1: Abstract Syntax Tree

ASTs use the following structure to parse code:

1. A *leaf* represents an expressions that has no subexpressions. An example being a terminal symbol or literal.
2. *Internal nodes* represent a compound expression. This means this expression is built from smaller subexpressions.
3. Nodes are often categorized by the kind of expression they represent. This can be with a *tag*, which uses an object oriented approach. Also, this can use a foreign concept, algebraic data types.
4. Node types often correspond to the grammar structure of a language. Meaning, if we expect a node type to contain subexpressions we should expect the grammar expressing that node type to include inputs for those subexpressions.

1.3 Semantics

1.3.1 Imperative Programming Languages

Languages such as Python, Java, and C, are fundamentally *imperative*. This means they are inspired by the Turing machine model of computation. Often defined as having programs organized around *statements*. This model of computation tells the computer what to execute as a collection of instructions.

1.3.2 Semantics

The **semantics** of a languages refer to the rules governing the meaning of programs written in that language. So if **syntax** defines how a program should look, **semantics** define what the program does, given that the program follows the syntax. In python this would be equivalent to the meaning behind what `return` and `for` do (keywords). So, if a program is simplified to evaluating an expression, where the result is a value, the semantics define what the *value* of the programs are.

This brings us to the concept of **denotational** vs **operational** semantics. Denotational semantics refers to the value of some program. This means two expressions, such as `10` and `3+7`, have the same denotational semantics if they have the same value, even if the process to get to that value is different. The *evaluation* steps to achieve some value is denoted as the operational semantics. Operational semantics can be very complex in imperative languages, yet simple in functional.

Note:-

There is another type of semantic called **axiomatic semantics**, these semantics focus on invariants similar to algorithmic analysis. So the axiomatic semantics of an expression focus on what is true about a piece of code

1.4 Lambda Calculus

Lambda calculus is a model of computation in which expressions are the fundamental unit of computation. This means that programs in lambda calculus are a single expression, often containing subexpressions.

Definition 1.4.1: The Lambda Calculus Expression

An expressions can be one of three things:

1. An **identifier**, such as x. Similar to variables in mathematics.
2. A **function expression**. Such as $\lambda x.x$, this expression takes one input, expressed after the λ, and one output, expressed after the ..
3. A **functional application**. Also known as function call. Example being f *expr*, which would apply the function f to the expression $expr$

Note:-

A strong concept of functional programming (lambda calculus), is that the only possible way to substitute programs is to substitute function calls with their value, as oppossed to programming languages where there are often many ways to substitute programs.

Chapter 2

Racket

2.1 Racket Basics

2.1.1 Read-Eval-Print: Evaluation by substitution

An expression is reduced to a value by repeatedly evaluating the outermost expression, and substituting its value.

2.1.2 Name Bindings in Racket are immutable

This follows the concept of **referential transparency**:

Definition 2.1.1: Referential Transparency

An identifier is to be **referentially transparent** if the identifier can always be substituted for its value without changing the program's behavior.

Example 2.1.1 (Non Referentially Transparent Identifier)

The python function, `localDate.time`, is not referentially transparent since it constantly changes.

Note:-

This is constant with all functions in pure functional languages. This means that a function, `f`, should always have the same output if it has the same arguments. Since variables are also immutable it means the loop statements are impossible. Instead we use recursion.

2.1.3 Local Name Bindings with the `let` Expression

Using `let` in Racket as its own block creates an expression which evaluates each line at different undefined points. Reassigning values during the `let` occurs after all of the statements have finished. Note the following statement:

Example 2.1.2 (Using let)

(define a 100) (define b 100)
(define c 100)
(let (

$$a5$$

$$b(+aa)$$

$$c(+ab)$$

)
(list a b c))
The result to this is: (5 200 200)

Note:-

This means that trying to use variables that are defined in the let, within other statements of let will be unable to determine the value and raise an error.