1 Introduction

1.1 Basic definitions

- Feature
  - Command: No return value, but does modify objects. On the synctactical level, it is an instruction
  - Query: Returns a value, but does not modify any objects. The syntax equivalent is the expression.
    - Functions get their results through computation
    - Attributes are values directly stored in memory

    For queries, there is the uniform access principle, which states that it doesn't matter to the client whether a query is implemented as a function or attribute. Features should be accessible to clients the same way whether implemented by storage or by computation.
  - Creation Procedure: Commands to initiate objects, can be several. There is also a *default_create*, which is inherited by all classes, and does nothing by default.
- Feature Calls
  - Unqualified calls: Feature calls which apply to the current object
  - Qualified calls: Feature calls which apply to a certain object, causing this object to become the current object.
- Class clauses
  - Indexing
  - Inheritance
  - Creation
  - Feature
  - Invariant
- Specimen: A syntactic element, such as a class name or an instruction, but no delimiters. The type of a specimen is its construct. See Describing syntax Add reference
- Abstract syntax tree: Shows the syntax structure with all its specimens, but obviously without any delimiters, A tree has nodes, each one of the following kind:
  - Root: Node with no incoming branch.
  - Leaf: Node without outgoing branches
  - Internal node: Neither of the former
- Basic elements of a program text:
  - Terminals
    - Identifiers: Names chosen by the programmer

        * Keywords
        * Special symbols, such as a period

- Describing a program
    - Semantic rules: Define the effect of programming, satisfying the syntax rules
    - Syntax rules: Define how to make up specimens out of tokens satisfying the lexical rules
    - Lexical rules: Define how to make up tokens out of characters
- Syntax: The way you write a program; characters grouped into words, grouped into bigger structures.
- Semantics: The effect you expect from this program at runtime
- Identifier: Name chosen by the programmer to represent certain program elements, such as classes, features or runtime values. If it denotes a runtime value, it is called an identity or variable if it can change its value. During execution, an entity may become attached to an object.
- Executing a system consists of creating a root object, which in an instance of a designated class from the system, the root class, using a designated creation procedure, called its root procedure.

1.2 Variables

- Types
    - Reference types: Entities with a reference value
    - Expanded types: Entities with an object as a value
    - A type is one of:
        * A non-generic class
        * A generic derivation, i.e. the name of a class followed by a list of types, the actual generic parameters, in brackets
- Setters: It is possible to make assignments such as $x.att := val$, which is shorthand for $x.set\_att(val)$
- Effect of an assignment
    - Reference types: Reference assignment
    - Expanded types: Value copy
- Variable copy
    - Shallow object duplication (creates a new object): $b := a.twin$
    - Deep object duplication (creates a new object): $b := a.deep\_twin$
    - Shallow field-by-field copy (does not create an object): $b.copy(a)$

Maybe add the objet creation diagram??

1.3 Interface

- A client of a software mechanism is a system of any kind - such as a software element or a human - that uses it. For its client, the mechanism is a supplier
- Interface: The description of techniques enabling clients to use these mechanisms. For example: GUIs (Graphical User Interface), command line interfaces (shell, bash,...), or APIs
- An object can be an instance of a class, if the class is the generating class of the object

1.4 Information Hiding

- For its clients, an attribute may be:
  - Secret
  - Read-only
  - Read, but partially write restricted (only certain things are allowed to be written)
  - Writing one or more classes in curly brackets after the keyword *feature* exports these features only to these classes and its descendants. If no class is listed, the features are exported to *ANY*.

  Information hiding only applies to use by clients using dot or infix notation. Unqualified calls are not subject to information hiding.

1.5 Control structures

Requires a lot of work

- Sequence or compound
- Loop
  - Loop invariant
    - Satisfied **after** initialization, after the *from* clause
    - Preserved by every loop iteration executed with the exit condition not satisfied. So in the end, the loop invariant and the exit condition hold!
  - Loop variant
    - Non-negative (i.e. ≥ 0) integer expression, right after initialization
    - Decreases while remaining non-negative for every iteration of the body with exit condition not satisfied.
- Conditional

1.6 Contracts

- Contracts are made of assertions, each containing an assertion tag and a condition (a Boolean expression)
- Precondition
  - Property that a feature imposes on every client
  - If there is no *require* clause, is treated as one, with one only being true.
- Postcondition
  - Property that a feature guarantees every client
  - Can make use of keyword *old*
- Class invariant
  - The invariant expresses consistency requirements between queries of a class

Check for an explanation of this keyword

1.7 Miscellaneous

- Semistrict operators
  - Let us define the order of expression evaluation
  - **and then** is the semistrict version of **and**. Use it if a condition only makes sense when another is true.
  - **or else** is the semistrict version of **or**. Use it if a condition only makes sense when another is false
  - **implies** is always semstrict!

2 Describing syntax

2.1 BNF

Backus-Naur-Form (BNF): A metasyntax used to express context-free grammars. A formal way to describe formal languages. It consists of the following parts

- **Delimiters**: Fixed tokens of the languages vocabulary, such as keywords and special symbols
- **Constructs**: They represent structures of the language, for instance *Conditional*. A particular instance of a construct os known as a specimen of the construct. There are two kinds of constructs:
  - **Nonterminal construct**: They are defined by a production
  - **Terminal construct**: Terminal constructs such as *Identifier* or *Integer* are not defined by this grammar, they are described at the lexical level

- **Productions**: They are associated with a particular construct and specify their specimens

Each production defines the syntax of specimens of a particular construct, in terms of other constructs and delimiters. An example for a production (not BNF-E)

$$A = B|C[D]\{E\text{“;”}\}^*$$

Depending on the right side of a production, they can be separated into three kinds:

- **Concatenation**: This production lists zero or more constructs, some may enclosed in brackets and said to be **optional**
- **Choice**: Listing one or more constructs, separated by vertical bars. A choice specifies that every specimen of the construct on the left consists of exactly one specimen of one of the constructs on the right
- **Repetition**: A construct, enclosed in curly brackets, followed by a star. This indicates zero or more occurrences of the construct, Example: $A = \{B\}^*$. The star might be replaced by a plus, indicating one or more repetition

2.2 BNF-E

- Every non-terminal must appear on the left side of exactly one production, called its defining production.
- Every production must be of one kind: either concatenation, choice or repetition
- There is also a major change in the repetition production. Instead of

  $$A = [B\{\text{terminal } B\}^*]$$

  one may write

  $$A = \{B \text{ terminal } \dots\}^*$$

  The same is also true for the plus instead of a star.

2.3 Regular Grammar

The regular grammar is generally used to describe the terminal construct, which could be done using BNF, but can be achieved more easily using a regular grammar. The rules are quite similar, although different:

- The use of **choice** is no problem, possibly with character intervals
- There are also **concatenations**, although they do not assume breaks (spaces, new lines, ...) between elements. But you may define them explicitly using a lexical construct.
- **Repetitions** have a different, simpler form; A^* or A^+, following the same rules

- No **recursion** is allowed whatsoever. As a result you may write any language in a single regular expression
- Unlike BNF-E, you may mix different kinds of productions

3 Inheritance and Genericity

3.1 Inheritance

- Terminology

  Check for a better explanation?

  - A class is a **parent** to another, if the other inherits directly from it, i.e. class A is a parent to class B, if class B inherits from class A
  - The **descendants** of a class are the class itself and (recursively) the descendants of its heirs (parents). **Proper descendant** excludes the class itself.

- **Features of classes**
  - They can be **inherited** if it is a feature of one of the parents of the class. They can also be **immediate** if it is declared in the class. In this case, the class is said to introduce the feature
  - Fully implemented features are called **effective**, otherwise one may call them **deferred**

- Contracts
  - The **invariant** of a class automatically includes the invariant clause from all its parents, and-ed
  - If no pre-/post condition is explicitly stated, features inherit the contracts from their parents
  - One can weaken the precondition with the keyword `require else`, resulting in a precondition `orig_pre` and `new_pre`
  - One can strengthen the postcondition with the keyword `ensure then`, resulting in a postcondition `orig_post` and `new_post`

3.1.1 Multiple Inheritance

- **Name clash**: If `C` inherits both from `A` and `B`, which both have a feature `f`, then we have a name clash. To resolve it, we can redefine one feature as `A_f`. A name clash must be resolved, unless it is:
  - Under repeated inheritance (the feature of `f` in `A` and `B` comes from a common ancestor, for instance `ANY`)
  - If at most one of the features `f` is effective, and all others are deferred. In that case (only one feature is effective), the features are said to be **merged**. Merging also works when one or more features are renamed. The merging happens after renaming!

  - If more than one feature are effective, merging can still help. We can **undefine** effective features, so that they are deferred again. Syntax: `undefine a,b,c end`. It is even possible to fist rename a feature, undefine it and then merge it.

- Repeated Inheritance
  - Features, not renamed along any of the inheritance paths, will be shared
  - Features, inherited under different names will be replicated
  - A potential ambiguity arises because of polymorphism and dynamic binding when class `C` inherits from `B` and `A`, but `A` redefines now copy (a feature of the common ancestor `ANY`). In this case, a simple rename will not be enough, we have to select (`select copy, f end`) the features from one parent, and rename the ones from the other.

3.2 Genericity

- Terminology:
  - A **formal generic paameter** is the parameter in the class, e.g. `LIST[G]` with `G` as formal generic parameter.
  - An **actual generic parameter** is the actual type passed as parameter in a type, e.g. `LIST[INTEGER]` with `INTEGER` being the actual generic parameter.
  - One can obtain a **generic derivation** of a generic class by passing a type

- Types
  - **Unconstrained** genericity: Any generic type is allowed. Example: `LIST[G]`, which is the same as `LIST[G->ANY]`
  - **Constrained** genericity: Only descendants are allowed as generic type. Example: `LIST[G->NUMERIC]`

3.3 Static Typing

- **Type-safe call** (during execution). A feature call `x.f` such that the object attached to `x` has a feature corresponding to `f`
- **Static type checker** is a program-processing tool (such as a compiler) that guarantees for any program that it accepts, that any call in any execution will be type-safe
- A programming language is called **statically typed language** if it is possible to write a static type checker

3.4 Polymorphism

- **Polymorphism** is the existence of the following possibilities:
  - An **attachment** (assignment or argument passing) is **polymorphic** if its target variable and source expression have different types.
  - An entity or expression is polymorphic if it may at runtime, as a result of polymorphic attachments, become attached to objects of different types.
  - A **container data structure** is polymorphic if it may contain references to objects of different types.
- The **static type** of an entity is the type used in its declaration in the class text. Similarly, the **dynamic type** of an entity is the type of the object, it is attached to. The type system ensures that the dynamic type of an entity will always conform to its static type.
- **Conformance**
  - A reference type `U` **conforms** to a reference type `T` if either:
    - They have no generic parameters, and `U` is a descendant of `T`
    - They both are generic derivations with the same number of actual generic parameters, the base class of `U` is a descendant of the base class of `T`, and every actual parameter of `U` (recursively) conforms to the corresponding actual parameter of `T`
  - An expanded type only conforms to itself
- **Object test**. Test the dynamic type of an object, e.g. `if {r:TYPE} obj then ... end`
  - `{r:TYPE}` is the object-test local, and only available in the `then`-part, not in the `else`-clause.
  - `obj` is the object to be tested
- **Assignment attempt**. Earlier mechanism for the object test. `a?=b` assigns `b` to `a` if and only if `b` is attached to an object whose type conforms to the type of `a`. Otherwise, `a` will be void

3.5 Dynamic Binding

- **Dynamic bonding** as a semantic rule is the property that any execution of a feature call will use the vesion of the feature best adapted to the type of the target object.

4 Recursion

- **Definition**: A definition is recursive if it involves one or more instances of the concept itself. Recursion is the use of a recursive definition.
- Recursion can be either direct (routine `r` calls `r`) or indirect (routine `r_1` calls `r_2` .. calls `r_n` calls `r_1`)

- To be useful, a recursive definition should ensure that:

  R1: There is at least one non-recursive branch

  R2: Every recursive branch occurs in a context that differs from the original

  R3: For every recursive branch, the change of context R2 brings it closer to at least one of the non-recursive cases R1

- Recursive calls cause (without further optimization) a run-time penalty: the stack of preserved calls needs to be maintained. Various optimizations are possible:
  - Recursive schemes can sometimes be replaced by a loop (**recursive elimination**)
  - **Tail recursion** (last instruction of a routine is a recursive call) can usually be eliminated
- **Recursive variant**: Every recursive routine should use a recursion variant, an integer quantity associated with any call, such that
  - The variant is always ≥ 0
  - If a routine execution starts with variant value v, the value v' for any call satisfies $0 \leq v' \leq v$

5 Data Structures

5.1 Trees

5.1.1 Binary Trees

- A binary tree `G`, for an arbitrary data type `G`, is a finite set of items called nodes, each containing a value of type `G`, such that the nodes, if any, are divided into three disjoint parts:
  - A single node, called the root of the binary tree
  - (Recursively) two binary trees over `G`, called the left and right subtree
- **Theorem:** For any node of a binary tree, there is a single downward path connecting the root to the node through successive applications of `left` and `right` links.
- **Traversals**
  - **In-order**: traverse left sub-tree, visit root, traverse right sub-tree
  - **Pre-order**: visit root, traverse left sub-tree, traverse right sub-tree
  - **Post-order**: traverse left sub-tree, traverse right sub-tree, visit root
- **Binary search tree**: This is a tree over a sorted set `G` if for every node `n`:
  - For every node `x` of the left sub-tree of `n`: `x.item` $\leq$ `n.item`

  - For every node `x` of the right sub-tree of `n`: `x.item` $\geq$ `n.item`

- In a binary search tree, average behavior for insertion, deletion and search is $O(\log(n))$, only worst case is $O(n)$

Add binary search tree image

5.2 Container data structures

- Containers contain other objects, and store them in numerous ways. They differ in various properties, like the available operations, the speed of these operations and storage requirements. Some fundamental operations on a container:

Insertion	Add an item
Removal	Remove an occurence (if any) of an item
Wipeout	Remove remove all occurences of an item
Search	Find out if a given item is present
Iteration (or trasversal)	Apply a given operation to every item

- The `EiffelBase` classes use standard names for basic operations:
  - Queries:
    * `is_empty: BOOLEAN`
    * `has(v:G): BOOLEAN`
    * `count: INTEGER`
    * `item: G`
  - Commands
    * `make`
    * `put(v:G)`
    * `remove(v:G)`
    * `wipe_out`
    * `start, finish`
    * `forth, back`

- The **cursor** is present in many containers. It ranges from 0 to $count + 1$, and `before` and `after` hold if the cursor is not on an item. In an empty list, the cursor is at position 0

- **Alias notation**. A feature may be declared as follows: `item(i: INTEGER)alias "[]": G assign put`. It is then possible to do `a[i]` for `a.item(i)` and `a.item(i):= x` or `a[i] := x` for `a.put(x,i)`

5.2.1 Lists

- A list is a sequence of elements of a certain type. List is a general concept and has various implementations, including `LINKED_LIST`, `TWO_WAY_LIST`, `ARRAYED_LIST`, etc.
- Lists have a **cursor**. The current cursor position can be obtained by the query `index`. The element at this position is generally obtained by `item` and there are many other queries about the cursor position: `after`, `before`, `off`, `is_first`, `is_last`
- There are also various commands for **cursor movement**: `start`, `finish`, `forth`, `back`, `go_i_th`
- **Adding and removing** elements is done using: `put_front`, `put_left`, `put_right`, `extend`, `remove` (item at cursor position)

5.2.2 Arrays