lazy-dog minimal documentation

Kerstin Hoffmann

March 23, 2024

Contents

1 This document

This is a WIP minimal documentation for the programming language lazy-dog. The contents of this document are subject to change as development goes on. It is furthermore not guaranteed that this document is fully up to date.

Furthermore, this document tries to be compiler agnostic.

2 Preprocessing

A preprocessor directive is a line that starts with `%`.

2.1 include

```
def %include !? Include... \n;
def :Include (Include...);
def :Include Identifier;
```

```
def :Include Identifier / Include;
```

Each Include is a set of source file paths. Each path points either to a file in the std directory or in the local directory. Prefixing a path with `std/` or `local/` makes this explicit, otherwise the local path has priority.

A source file ends in `lzy`. Including a file twice has no effect, unless `%include! file` is used.
Once files `file1` to `filen` have been included, the file

```
file1_x_file2_x_ ... _x_filen.lzy
```

if it exists is also included after the place where `filen` was included.
The files have to be sorted in alphabetical order. Such combination files of small n are included first.

3 Lexing

Lexical analysis transforms a stream of source lines into a stream of lexemes.

3.1 Vocabulary

An **Identifier** is a name, type name or symbol. An identifier is a lexeme.

A **Token** is a number that points to a string representing an identifier.

A **Lexeme** corresponds to a substring of the source code. However, it is an actual object in lazy-dog. It can be either

- an identifier. In that case, the lexeme has the token value.
- a string or number literal. Here, the lexeme has a string representing the contents of the a string or the digits of a number.

Lexemes can be manipulated in a number of ways, but this is not part of lexical analysis.

3.2 Lexical Rules

No lexeme spans multiple lines, including string literals.

Whitespace (ie space, newline and tab) in between lexemes is discarded.

Comments starting with `#` until the end of the line are ignored.

A **Name** contains lowercase letters, underscores and digits. It starts with a letter and doesn't end in a underscore.

A **Type Name** contains letters and digits. It starts with an uppercase letter and doesn't end in a digit.

A **Symbol** is a single character of the following string:

```
!$%&'()*+,-./:;<=>?@[\]^_{|}~
```

These are all printable ascii characters except letters, digits, double quotes `"`, the hash `#` and the backtick `‘`.

A **Number Literal** consists only of digits. Underscores between the digits are allowed and are just ignored. Note that a number literal is *not* a number, so we don't care about representing negative numbers and the likes right now.

A **String Literal** starts with double quotes and ends with double quotes following an even amount of backslashes (including 0). Normal C style escape rules apply. The outer double quotes are discarded.

4 Objects and Types

As mentioned above, lexemes are objects. All data in lazy-dog is just a list of objects.
Each Object has a type.

4.1 Data

For each non-negative multiple of 8 n, there is the type `Data`n, for example `Data64` or `Data16`.

This type denotes that an object represents n bits of data. Keep in mind that due to alignment/padding issues, the actual size of the object might differ from n.

Each object has to have a Data type.

4.2 Concatenation

Two objects `t` and `u` of types `T` and `U` can be concatenated into `t u` which has type `T U`

Concatenation is associative and has a neutral element, the empty type.

The resulting object has a Data type that is the sum of the Data types of `t` and `u`.

4.3 Tags

Many types are not subsets of Data types. They are called **Tags**. Such types of course can't have any objects associated to them, but they can be used in the type algebra to produce other types.

If `T` has data and `U` is a tag, then `T U` is the type `T` *tagged with* `U`. If both `T` and `U` are tags, `T U` is their *union*.

This operation is not associative. The following rules apply however:

```
D (T1 T2) == (D T1) T2
(T1 T2) T3 == T1 (T2 T3)
T T == T
T1 T2 == T2 T1
```

This means that unioning tags is associative, idempotent and commutative. But tagging data generally is not, ie in general:

```
D1 D2 T == D1 (D2 T) != (D1 D2) T
```

4.4 Alternation

Two tags `T1`, `T2` may be alternated. This gives `T1|T2`. Alternation is associative. If an alternation tags a data type `D` then each object of type `D T1|T2` also stores an integer that tells us which of these tags is active.
This can be checked via the builtin function `is x type`