Implementation of the TIL-Script Language

Filip Peterek

VSB – Technical University of Ostrava

May 2023

Project Goals

- Define the TIL-Script programming language
  - Improve upon the current grammar
  - Define the semantics of the language
- Create a working TIL-Script interpreter
- Document the language properly

Transparent Intensional Logic

- Logical analysis of natural language
- Based on typed Lambda calculi
- Rigorously defined type hierarchy
- Procedural and hyperintensional
- Constructions can mention other constructions
- Sentence meaning is carried by a procedure
  - We mostly care for the procedure itself, seldom do we care for the value it produces

TIL-Script

- Grammar closely resembles TIL grammar
- Adds upon TIL to form a useable programming language
  - Lists
  - Tuples
  - Structures
  - Strings
- Also adds restrictions imposed by computers with finite resources

Original Features

- Grammar
- All TIL constructions
- Type aliases
- Lists, Tuples
- Semantics not fully defined

New Features

- Comments
- Imports
- Nil value
- Distinction between definitions and declarations
- String type
- Tuples are now heterogenous
- Structs (user defined types)
- Types as language objects

Declarations And Definitions

- Declaration only specifies a name and a type
  - I.e. if we only know, or only care for, the name
  - Halting problem – we may want to refer to it, but we can't solve it
  - Function Halts/(Bool Program) cannot be implemented
- Definition also specifies semantics or value

Code Example

```
numbers -> List(Int) := ['ListOf '1 '6 '2 '5 '3 '4].

Defn LessThanFive(num: Int) -> Bool := ['< num '5].

Defn Filter(list: List(Int), pred: (Bool Int))
        -> List(Int) :=
    ['If ['IsEmpty list]
        list
        ['If [pred ['Head list]]
            ['Cons
                ['Head list]
                ['Filter ['Tail list] pred]]
            ['Filter ['Tail list] pred]]].

['PrintIn ['Filter numbers 'LessThanFive]].
```

Implementation

- Kotlin
  - Algebraic data types
  - Null safety
  - Immutable interfaces for collections
- Gradle
- Antlr

Project Structure

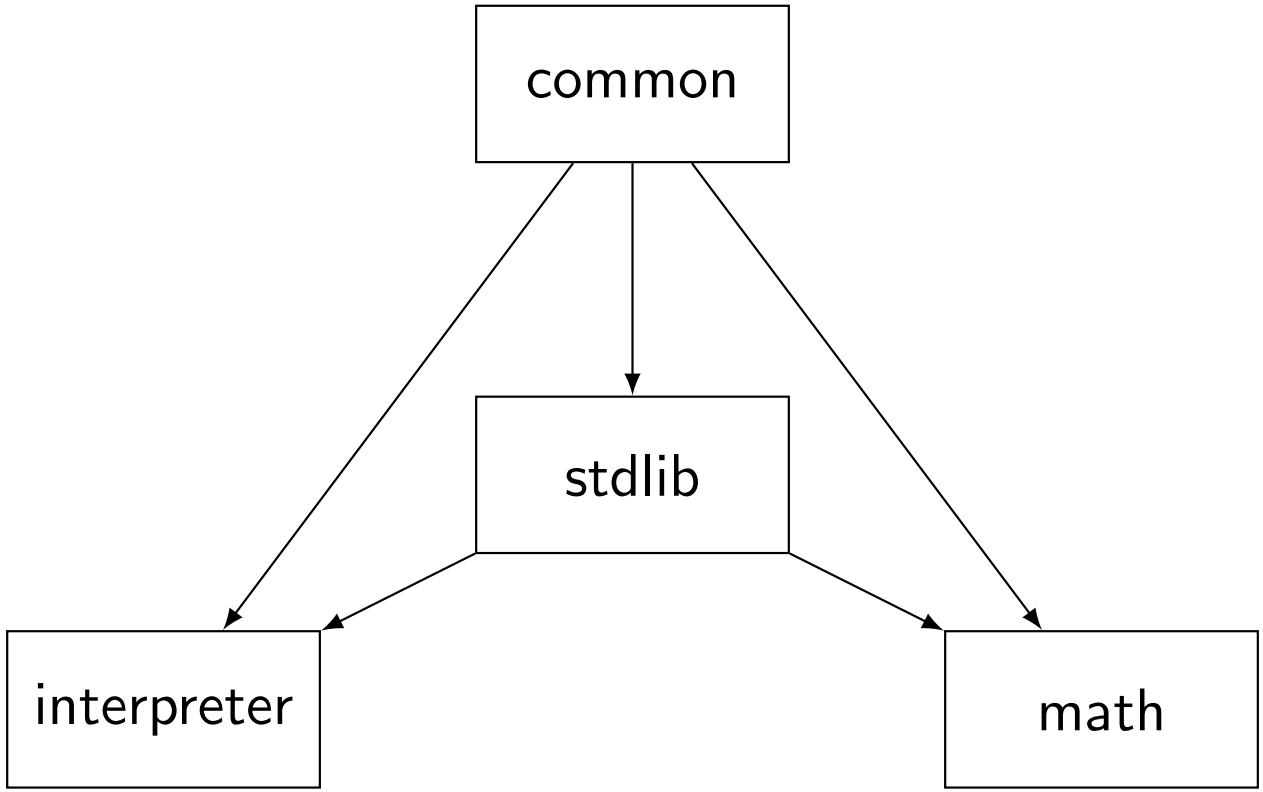

Figure: Project Structure

Interpreter Implementation

- Written in Kotlin
- Parser is autogenerated
- Functions can be written in TIL-Script or JVM languages
  - Allows for access to the entire JVM ecosystem
  - I.e. database libraries don't have to be implemented from scratch
  - JDBC can be used
  - No need to handle syscalls
  - No need to reimplement math functions

Interpreter Implementation

- Object oriented design
  - Not the best choice for an interpreter
  - Simplifies dealing with functions of different implementations
  - The interpreter can be replaced with a better, drop-in replacement
  - Limited choice in the JVM
- Functional approach where possible
- Immutability is preferred

Current State

- Working prototype
- All features work as intended
- Lists, tuples, user-defined structures, mutually recursive functions, variables, imports, etc.
- Type coherency checking

Limitations

- Working prototype
- No bytecode
  - Negative performance implications
  - Callstack is tied to the JVM callstack
  - Bytecode conversion must be bijective
- No TCO
  - Unfortunate, but cannot be done without bytecode

Possible Improvements

- REPL
- Editor support and an LSP
- Compilation to bytecode

The End