Regular Expressions I

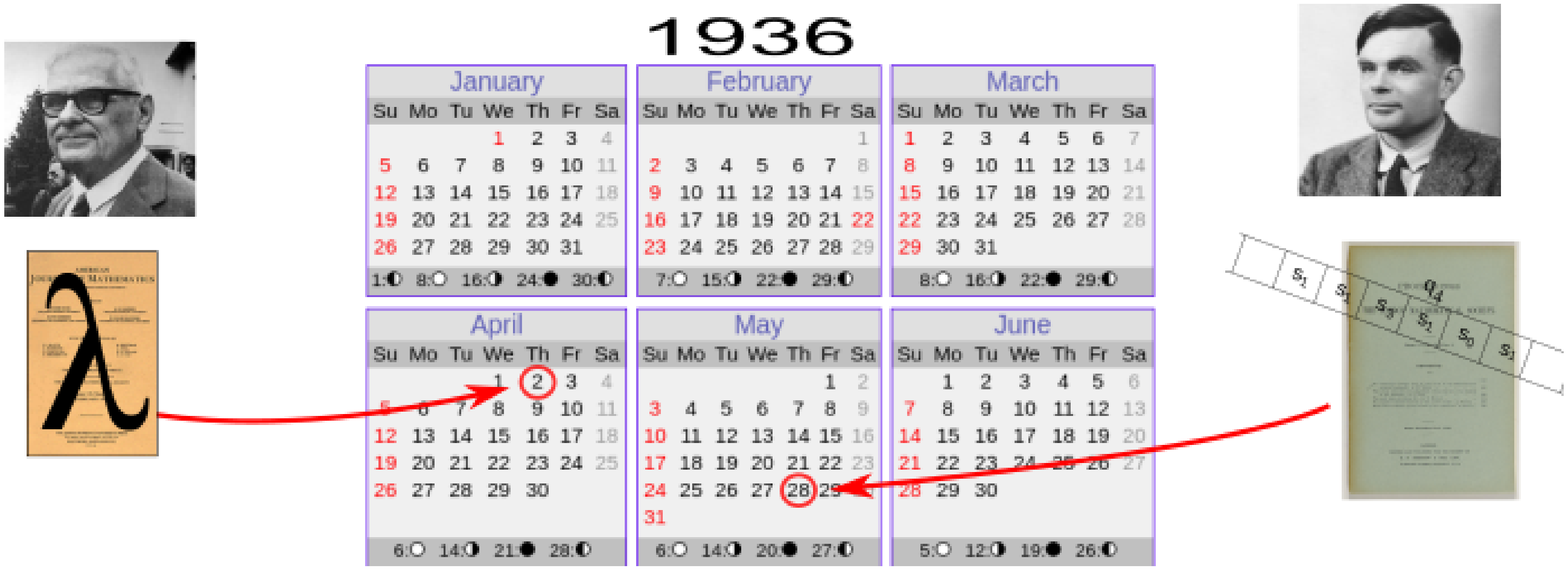
Languages, Equality Types, and the `regexp` type

15-150 M21

Lecture 0707
07 July 2021

0 Decision Problems

Birth Year of Computation

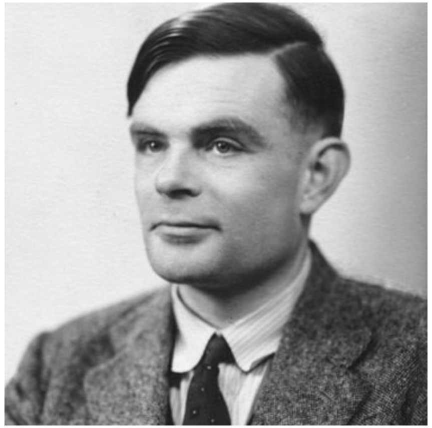


Warning: Great Theoretical Ideas

When working with Turing computability, we assume we have some (usually finite) set Σ – our **alphabet**

We'll be computing with the set Σ^* of all finite strings/sequences/lists of elements of Σ ("strings over Σ"). For instance,

$$\{a\}^* = \{\epsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa, \ldots\}$$



Computation is done by **machines**, which take some input (encoded as a string over some alphabet Σ) and either accepts it or rejects it.

$$M : \Sigma^* \rightharpoonup \{\textbf{ACCEPT}, \textbf{REJECT}\}$$

This is a partial function, because it could loop forever on some inputs.

For any Turing machine M, we write

$$\mathcal{L}(M) = \{s \in \Sigma^* \mid M(s) = \textbf{ACCEPT}\}$$

for the **language** of M.

Question: Are all subsets $L \subseteq \Sigma^*$ **computable**: is there a Turing Machine M such that $L = \mathcal{L}(M)$?

Thm. (Turing, 1936) No!

- Σ^* is countably-infinite, so $\mathcal{P}(\Sigma^*)$ is uncountably infinite. But there are only countably-many possible Turing Machines
- There are explicit subsets of Σ^* which are not computable, e.g. **HALTS**

We can have a similar idea in functional programming:

```
M : t -> bool
```

```
M : Sigma list -> bool
```

where `Sigma` is the alphabet type (e.g. `char`).

Question: For which sets L of values of type `Sigma list` is there an SML function `M:Sigma list -> bool` such that

$$L = \{\texttt{v:Sigma list} \mid \texttt{M(v)} \Longrightarrow \texttt{true}\}?$$

Thm. For each finite set Σ (with corresponding SML type `Sigma`), and each subset L of Σ^* the following are equivalent:

- there exists a Turing Machine M such that $L = \mathcal{L}(M)$
- there exists an SML function `M : Sigma list -> bool` such that

$$L = \{\texttt{v:Sigma list} \mid \texttt{M(v)} \Longrightarrow \texttt{true}\}$$

A type `t` is said to be an **equality type** if there is a total function `(op =): t * t -> bool` deciding whether elements of that type are equal or not.

Examples: `int`, `bool`, `char`, `string`, `int list`

Non-Examples: `real`, `int -> int`

We can specify that a polymorphic type variable *must* be instantiated to an equality type by writing it with double tick-marks:

```
(op =) : ''a * ''a -> bool
Fn.equal : ''a -> ''a -> bool
Fn.notEqual : ''a -> ''a -> bool
```

Module: `Language`

`https://github.com/smlhelp/aux-library/blob/main/Language.sml`

aux-library/Language.sml

```
fun everything (x:'S list) = true
```

For each type `Sigma`, we identify each `L:Sigma language` with the set of values `s:Sigma list` such that `L(v)` $\Longrightarrow$ `true`.

aux-library/Language.sml

```
val everything : 'S language
val nothing : 'S language
val singleton : ''S list -> ''S language
```

Language Combinators

aux-library/Language.sml

```
  val Or
    : 'S language * 'S language -> 'S language
  val And
    : 'S language * 'S language -> 'S language
  val Not
    : 'S language -> 'S language
  val Xor
```

Goal: Write more complex languages

5-minute break

`string` versus `char list`

Take `Sigma` to be `char`. We have a special way of dealing with `char list`s... `string`s!

```
String.explode : string -> char list
String.implode : char list -> string
```

aux-library/Language.sml

1 Regular Expressions

Idea:

Come up with an SML datatype whose elements encode different languages

We'll define a datatype parametrized by a single equality type variable

```
datatype ''S regexp = ...
```

such that each value `R : t regexp` defines a language

$$\mathcal{L}(\texttt{R}) = \{\texttt{v : t list} \mid \texttt{v} \text{ “matches with” } \texttt{R}\}$$

Module: Regexp

`https://github.com/smlhelp/aux-library/blob/main/Regexp.sml`

(requires `Language.sml` in the same directory) aux-library/Regexp.sml

```
end
structure Regexp : REGEXP =
struct
  datatype ''S regexp =
        Zero
      | One
      | Const of ''S
```

Note: you can't actually require that the parameter of a datatype be an equality type. SML will treat this the same as `datatype 'S regexp = ...`, but the `''S` reminds us to use this with equality types aux-library/Regexp.sml

```
        | Times of ''S regexp * ''S regexp
        | Star of ''S regexp

  fun depth Zero = 0
    | depth One = 0
    | depth (Const(_)) = 0
    | depth (Plus(R1,R2)) =
        1 + Int.max(depth R1,depth R2)
    | depth (Times(R1,R2)) =
```

Idea: For each `R : Sigma regexp`, define the language $\mathcal{L}(\texttt{R})$ to be the set of all values of type `Sigma list` which "match" or "are accepted" by `R`. We'll do this recursively based on `R`.

> `LL : ''S regexp -> ''S language`
> REQUIRES: true
> ENSURES: `LL(R)` is a total function deciding $\mathcal{L}(\texttt{R})$, i.e. `LL R s` $\Longrightarrow$ `true` for all `s` $\in \mathcal{L}(\texttt{R})$, and `LL R s` $\Longrightarrow$ `false` for all `s` $\notin \mathcal{L}(\texttt{R})$.

- `Const v` only matches with `[v]`

$$\mathcal{L}(\texttt{Const v}) = \{\texttt{[v]}\}$$

- `One` only matches with `[]`

$$\mathcal{L}(\texttt{One}) = \{\texttt{[]}\}$$

- `Zero` does not match with anything

$$\mathcal{L}(\texttt{Zero}) = \emptyset$$

- `Plus(R1,R2)` matches with any list which matches either `R1` or `R2`

$$\mathcal{L}(\texttt{Plus(R1,R2)}) = \mathcal{L}(\texttt{R1}) \cup \mathcal{L}(\texttt{R2})$$

- `Times(R1,R2)` matches with any list consisting of an `R1` list appended to an `R2` list

$$\mathcal{L}(\texttt{Times(R1,R2)}) = \{\texttt{v1@v2} \mid \texttt{v1} \in \mathcal{L}(\texttt{R1}) \text{ and } \texttt{v2} \in \mathcal{L}(\texttt{R2})\}$$

- `Star(R)` matches with any list consisting of finitely-many `R`-matching lists appended together

$$\mathcal{L}(\texttt{Star(R)}) = \{\texttt{v}_1 \texttt{@} \texttt{v}_2 \texttt{@} \texttt{...} \texttt{@} \texttt{v}_n \mid n \in \mathbb{N} \text{ and } \texttt{v}_i \in \mathcal{L}(\texttt{R}) \text{ for each } 1 \leq i \leq n\}$$

Note: $\texttt{[]} \in \mathcal{L}(\texttt{Star(R)})$ for all `R`.

Next Time:

How `Regexp.LL` is implemented

Thank you!