NPRG077

TinyBASIC: A tiny interactive imperative programming system

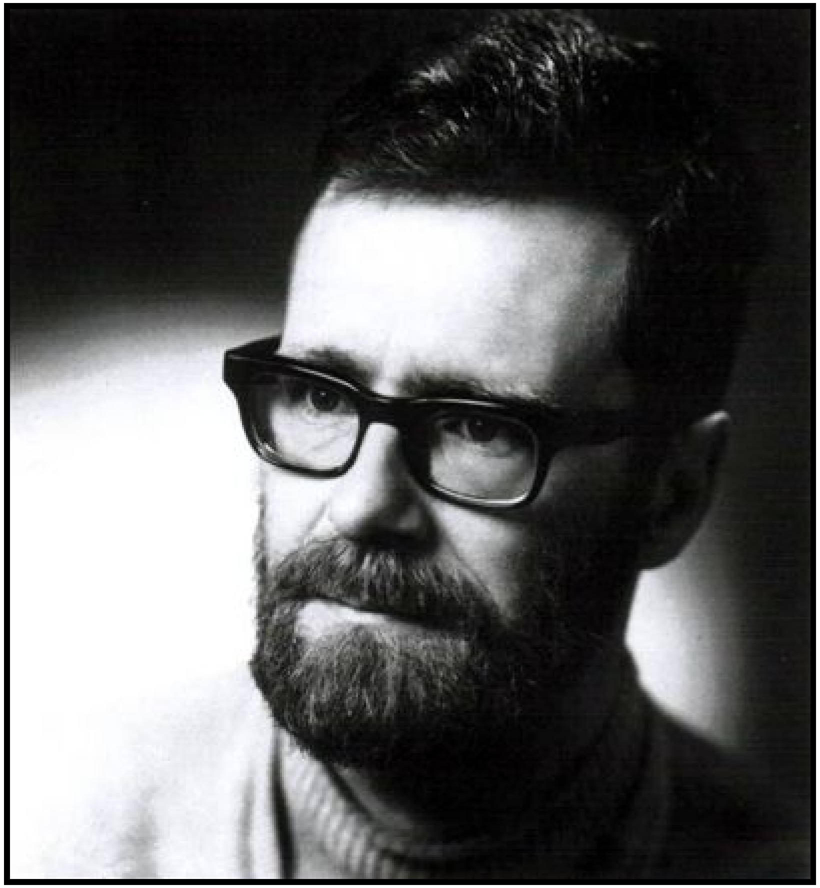
Tomáš Petříček, 309 (3rd floor)

✉ petricek@d3s.mff.cuni.cz

➔ https://tomasp.net | @tomaspetricek

➔ https://d3s.mff.cuni.cz/teaching/nprg077

Dijkstra on BASIC



It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

BASIC as a language?

Meh.

Charles
University

Why look at BASIC?

BASIC as a programming system

- Right at the birth of microcomputers
- Part of an early computing culture
- Interesting mode of interaction!

BASIC as a programming problem

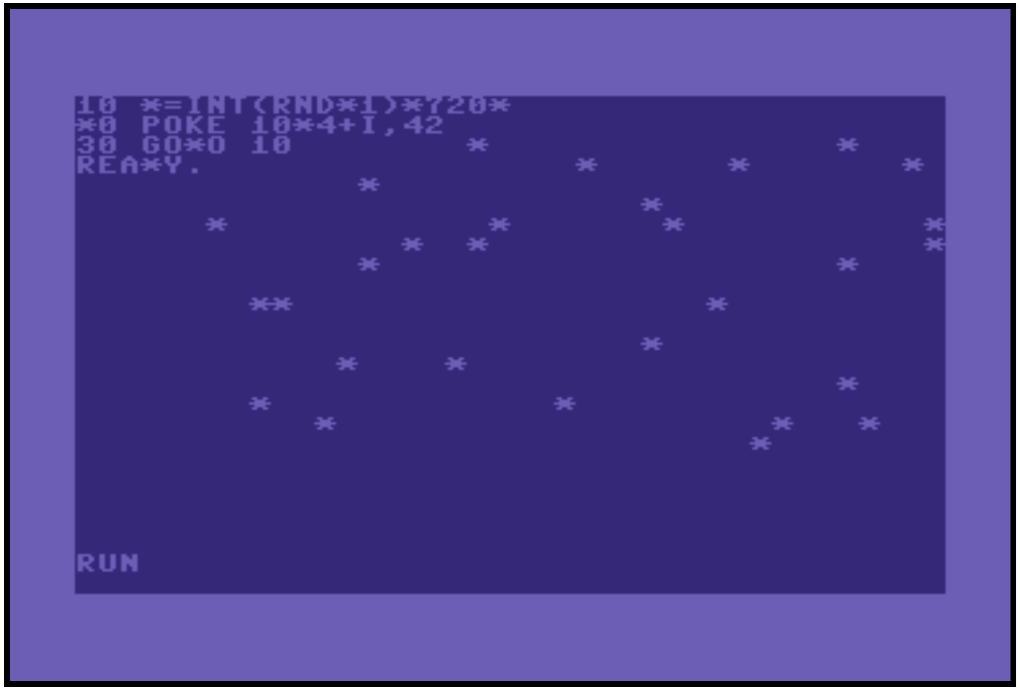
- Interpreter with richer state
- Statements vs. expressions
- More interesting F# programming!

Charles University

BASIC

Interaction and emulators

Charles University



```
10 *=INT(RND*1)*720*
*0 POKE 10*4+I,42
30 GO*O 10
REA*Y.
```

```
RUN
```

Demo

Writing BASIC in C64 emulator

Realistic machine-level system emulator

All the clever hacks with **POKE** work!

See: C64 emulator

What is interesting about it?

Learnability

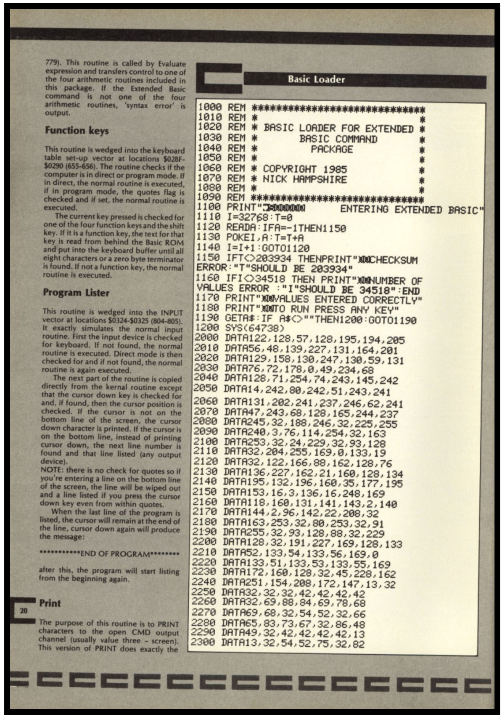
- Your computer boots into BASIC
- Copy games code from magazines

From novice to hacker

- Simple, but you can do lots with **POKE**

Interaction mode

- Code editor and REPL at the same time



779). This routine is called by Evaluate expression and transfers control to one of the four arithmetic routines included in this package. If the Extended Basic command is not one of the four arithmetic routines, 'syntax error' is output.

Function keys

This routine is wedged into the keyboard table set-up vector at locations $028F-$0290 (655-656). The routine checks if the computer is in direct or program mode. If in direct, the normal routine is executed, if in program mode, the quotes flag is checked and if set, the normal routine is executed.

The current key pressed is checked for one of the four function keys and the shift key. If it is a function key, the text for that key is read from behind the Basic ROM and put into the keyboard buffer until all eight characters or a zero byte terminator is found. If not a function key, the normal routine is executed.

Program Lister

This routine is wedged into the INPUT vector at locations $0324-$0325 (804-805). It exactly simulates the normal input routine. First the input device is checked for keyboard. If not found, the normal routine is executed. Direct mode is then checked for and if not found, the normal routine is again executed.

The next part of the routine is copied directly from the kernal routine except that the cursor down key is checked for and, if found, then the cursor position is checked. If the cursor is not on the bottom line of the screen, the cursor down character is printed. If the cursor is on the bottom line, instead of printing cursor down, the next line number is found and that line listed (any output device).

NOTE: there is no check for quotes so if you're entering a line on the bottom line of the screen, the line will be wiped out and a line listed if you press the cursor down key even from within quotes.

When the last line of the program is listed, the cursor will remain at the end of the line, cursor down again will produce the message:

***********END OF PROGRAM********

after this, the program will start listing from the beginning again.

Print

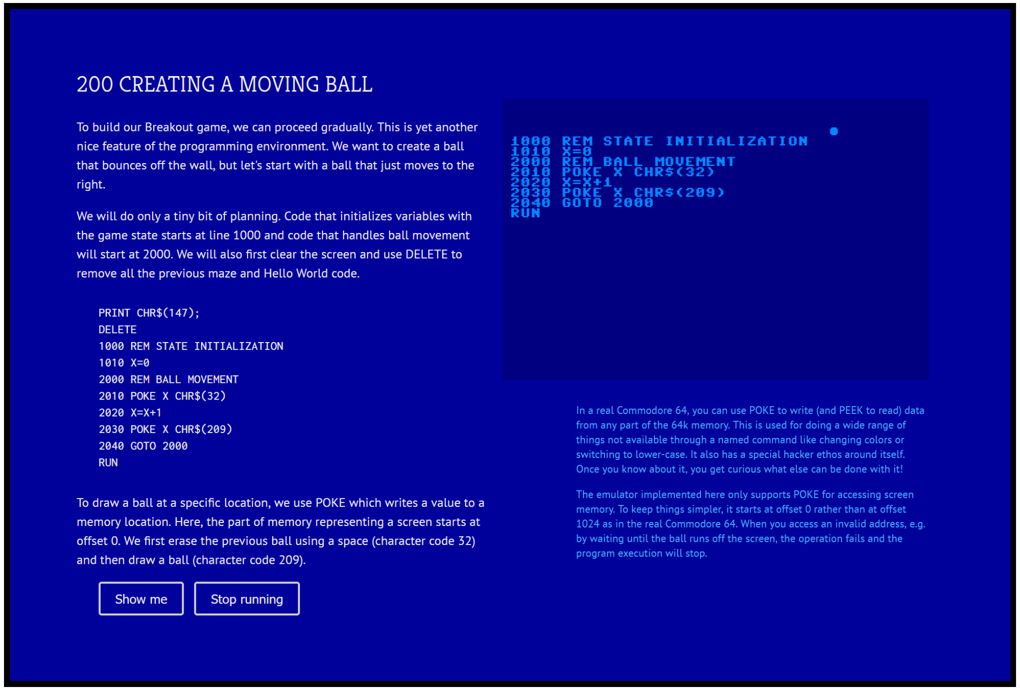
The purpose of this routine is to PRINT characters to the open CMD output channel (usually value three – screen). This version of PRINT does exactly the

Basic Loader

```
1000 REM ******************************
1010 REM *                            *
1020 REM * BASIC LOADER FOR EXTENDED *
1030 REM *       BASIC COMMAND        *
1040 REM *         PACKAGE            *
1050 REM *                            *
1060 REM * COPYRIGHT 1985             *
1070 REM * NICK HAMPSHIRE             *
1080 REM *                            *
1090 REM ******************************
1100 PRINT"          ENTERING EXTENDED BASIC"
1110 I=32768:T=0
1120 READA:IFA=-1THEN1150
1130 POKEI,A:T=T+A
1140 I=I+1:GOTO1120
1150 IFT<>203934 THENPRINT"CHECKSUM
ERROR:"T"SHOULD BE 203934"
1160 IFI<>34518 THEN PRINT"NUMBER OF
VALUES ERROR :"I"SHOULD BE 34518":END
1170 PRINT"VALUES ENTERED CORRECTLY"
1180 PRINT"TO RUN PRESS ANY KEY"
1190 GETA$:IF A$<>""THEN1200:GOTO1190
1200 SYS(64738)
2000 DATA122,128,57,128,195,194,205
2010 DATA56,48,139,227,131,164,201
2020 DATA129,158,130,247,130,59,131
2030 DATA76,72,178,0,49,234,68
2040 DATA128,71,254,74,243,145,242
2050 DATA14,242,80,242,51,243,241
2060 DATA131,202,241,237,246,62,241
2070 DATA47,243,68,128,165,244,237
2080 DATA245,32,188,246,32,225,255
2090 DATA240,3,76,114,254,32,163
2100 DATA253,32,24,229,32,93,128
2110 DATA32,204,255,169,0,133,19
2120 DATA32,122,166,88,162,128,76
2130 DATA136,227,162,21,160,128,134
2140 DATA195,132,196,160,35,177,195
2150 DATA153,16,3,136,16,248,169
2160 DATA118,160,131,141,143,2,140
2170 DATA144,2,96,142,22,208,32
2180 DATA163,253,32,80,253,32,91
2190 DATA255,32,93,128,88,32,229
2200 DATA128,32,191,227,169,128,133
2210 DATA52,133,54,133,56,169,0
2220 DATA133,51,133,53,133,55,169
2230 DATA172,160,128,32,45,228,162
2240 DATA251,154,208,172,147,13,32
2250 DATA32,32,32,42,42,42,42
2260 DATA32,69,88,84,69,78,68
2270 DATA69,68,32,54,52,32,66
2280 DATA65,83,73,67,32,86,48
2290 DATA49,32,42,42,42,42,13
2300 DATA13,32,54,52,75,32,82
```

Demo

My C64 essay



200 CREATING A MOVING BALL

To build our Breakout game, we can proceed gradually. This is yet another nice feature of the programming environment. We want to create a ball that bounces off the wall, but let's start with a ball that just moves to the right.

We will do only a tiny bit of planning. Code that initializes variables with the game state starts at line 1000 and code that handles ball movement will start at 2000. We will also first clear the screen and use DELETE to remove all the previous maze and Hello World code.

```
PRINT CHR$(147);
DELETE
1000 REM STATE INITIALIZATION
1010 X=0
2000 REM BALL MOVEMENT
2010 POKE X CHR$(32)
2020 X=X+1
2030 POKE X CHR$(209)
2040 GOTO 2000
RUN
```

To draw a ball at a specific location, we use POKE which writes a value to a memory location. Here, the part of memory representing a screen starts at offset 0. We first erase the previous ball using a space (character code 32) and then draw a ball (character code 209).

```
1000 REM STATE INITIALIZATION
1010 X=0
2000 REM BALL MOVEMENT
2010 POKE X CHR$(32)
2020 X=X+1
2030 POKE X CHR$(209)
2040 GOTO 2000
RUN
```

In a real Commodore 64, you can use POKE to write (and PEEK to read) data from any part of the 64k memory. This is used for doing a wide range of things not available through a named command like changing colors or switching to lower-case. It also has a special hacker ethos around itself. Once you know about it, you get curious what else can be done with it!

The emulator implemented here only supports POKE for accessing screen memory. To keep things simpler, it starts at offset 0 rather than at offset 1024 as in the real Commodore 64. When you access an invalid address, e.g. by waiting until the ball runs off the screen, the operation fails and the program execution will stop.

Explore the interaction

How it helps write, test and debug code?

Not fully accurate

Program does not live in memory, **POKE** offsets are wrong

The F# language

What you need to know

TinyBASIC

What F# do you need to know

- Project, console and tail recursion
- F# language logic and data types
- Records, functions, tuples, patterns
- List processing using built-in functions

Demo

Project, console, recursion

Charles University

Tuples, patterns and functions

```
let point = (1, 10)
let (x, y) = point

(* (int*int) -> (int*int) *)
let rotate (x, y) = (y, x)

(* int -> (int*int) -> (int*int) *)
let moveX by (x, y) = (x + by, y)

(* (int*int) -> int *)
let area (x, y) =
  match x, y with
  | 0, _ | _, 0 -> 0
  | x, y -> x * y

(* (int*int) -> int *)
let area pt =
  match pt with
  | ((0, _) | (_, 0)) -> 0
  | x, y -> x * y
```

Tuple type `int * int` is just another ordinary type of values

Pattern `(x,y)` can appear in multiple locations in code

Functions can mix arguments and tuples

Charles University

SKETCH

Tuples and patterns

Charles University

List constructors and list patterns

```
let l1 = [1; 2; 3; 4]
let l2 = 1::2::3::4::[]
let l3 = [1..4]

(* Pattern matching lists *)
match list with
| [e1; e2] -> (...)
| el::els -> (...)
| [] -> (...)

(* Possible, but not very useful *)
let (e::es) = list
let foo [e1;e2] = (...)

(* Higher-order list functions *)
let twice x = x * x
List.map twice [1..10]
List.map (fun x -> x * x) [1..10]
List.sum [1..10]
```

List type written as `list<int>` or `int list`

Constructed using `::` (rare) and `[..]` (often)

Patterns `::` and `[..]` can appear anywhere, but are partial

Demo

Real-world list processing

Charles
University

The pipe operator

Fluent style for functional data processing

```
let (|>) x f = f x
```

In bash scripting (|), adopted by R (%>%), maybe JavaScript

Charles University

Demo

Real-world list processing (2/2)

TinyML

Interpreter structure

Charles University

BASIC interpreter structure (1/2)

```
type Value = (* .. *)
type Expression = (* .. *)

type Command =
  (* Jumps and subroutines *)
  | Goto of int
  | GoSub of int
  | Return
  (* I/O operations *)
  | Clear
  | Print of Expression list
  | Input of string
  (* If, variables and control *)
  | If of Expression * Command
  | Assign of string * Expression
  | Run
  | Stop
```

Expressions evaluate to Values and are simple

Commands contain all the operations that modify the program state

BASIC interpreter structure (2/2)

```
(* State of the interpreter stores
   program lines as sorted list,
   variables in a dictionary,
   generator for the RND function
   and stack for GOSUB/RETURN *)
type State =
  { Program : list<int * Command>
    Variables : Map<string, Value>
    Random : System.Random
    ReturnStack : int list }

(* Evaluate a command and then
   run the next one (if any)
   until the program ends.
 : State -> (int * Cmd) -> State *)
let rec runCommand state (line, cmd) =
  (* ... *)

(* Find the next line after 'line'
   and run that or stop if none *)
and runNextLine state line =
  (* .. *)
```

State is the program source code, variables (and a few extras)

Current line is also a part of the state (function argument)

Demo

BASIC Hello World

TinyBASIC

How the BASIC language works

Charles
University

BASIC basics

```
REM You can write comments!
REM Jumping and calls
GOTO 10
GOSUB 10
RETURN

REM Printing to the screen
POKE 1024 CHR$(42)
PRINT "HELLO ";X

REM Variables and ifs
X=10
IF (X>0) GOTO 10

REM Control
RUN
STOP
```

GOSUB jumps, but keeps return location on stack for **RETURN**

PRINT takes a sequence of expressions (and we ignore cursor moving)

POKE writes a byte to memory (we will cheat)

We ignore command chaining (:)

Demo

Elegant programs with GOSUB :-)

TinyBASIC

A bit of theory

Charles
University

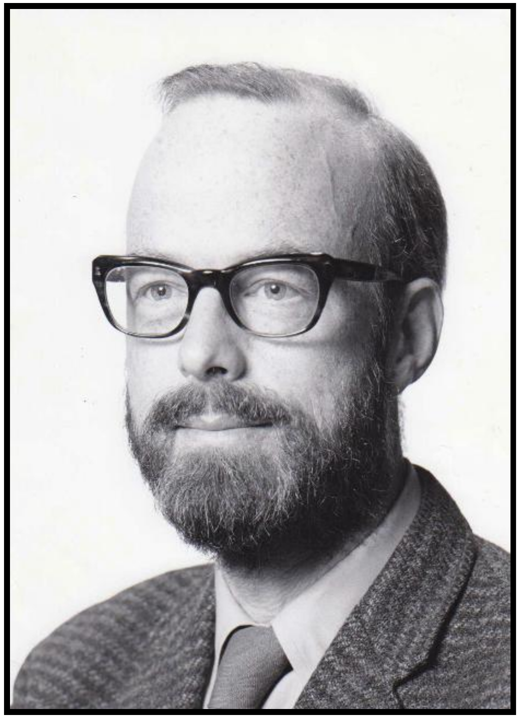
Meaning of programs

Functional languages

- Compositional semantics
- Define meaning of $e_1 + e_2$ in terms of the meaning of e_1 and e_2

Imperative languages

- What is the meaning of `PRINT "HI"`?
- What is the meaning of `GOTO 10`?
- Whatever the interpreter does..
- Not very good for program proofs!



Reasoning about BASIC programs

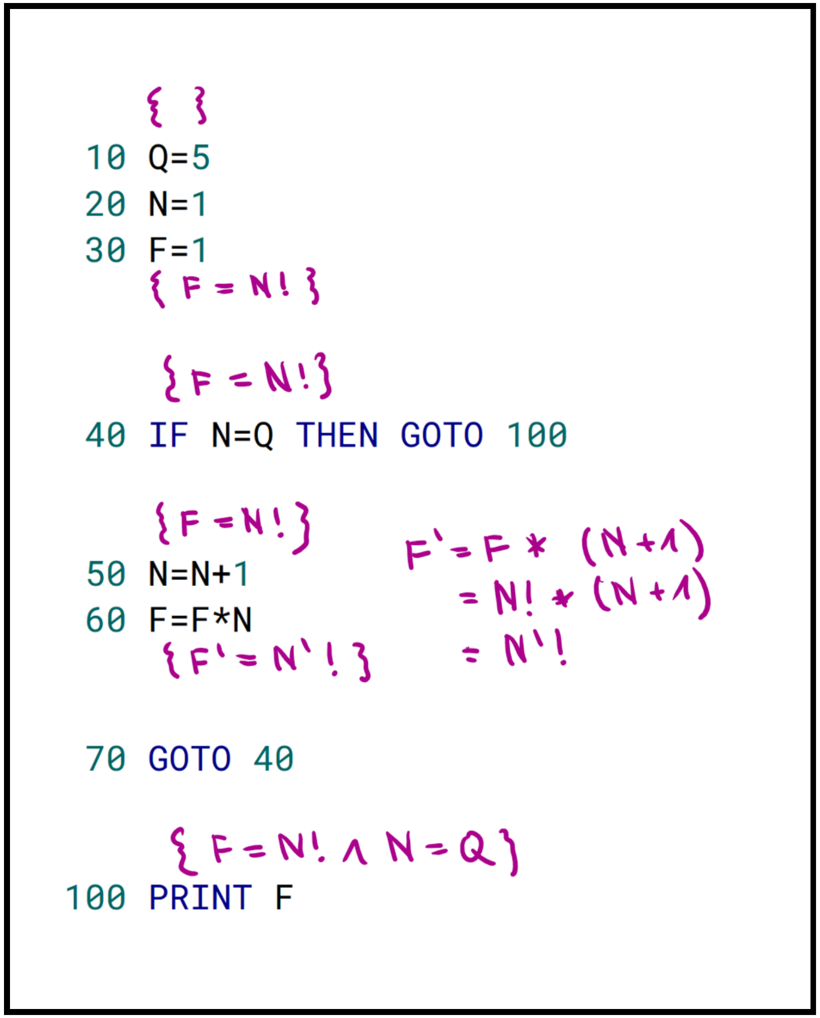
```
 00 REM FACTORIAL IN BASIC
 10 Q=5
 20 N=1
 30 F=1
 40 IF N=Q THEN GOTO 100
 50 N=N+1
 60 F=F*N
 70 GOTO 40
100 PRINT F
```

Hoare triples $\{P\}c\{Q\}$

Pre-condition P what is true before the command execution

Post-condition Q what is true after the command execution

Reasoning about BASIC programs



```
    { }
10  Q=5
20  N=1
30  F=1
    { F = N! }

    { F = N! }
40  IF N=Q THEN GOTO 100

    { F = N! }              F' = F * (N+1)
50  N=N+1                      = N! * (N+1)
60  F=F*N                      = N'!
    { F' = N'! }

70  GOTO 40

    { F = N! ∧ N = Q }
100 PRINT F
```

Postconditions of a command before have to match preconditions of a command after

Coming up with the right properties is tricky!

Charles University

Lab overview

TinyBASIC system step-by-step

Charles University

TinyBASIC - Basic tasks

1. Add `GOTO` and better `PRINT` for infinite loop fun!
   Evaluation of expressions, finding of the next line
2. Implement interactive program editing
   Handle commands that edit the program code
3. Add variables and the conditional command
   Needs `Map<string, Value>` in the program state
4. Random function and (not quite correct) `POKE`
   To be able to generate random stars!
5. A few more functions and operators
   As required by the Nim (subtraction) game

TinyBASIC - Bonus tasks

1. Add support for more elegant programs with GOSUB
   Needs **`list<int>`** (stack of return line numbers) in state
2. Refactor our Nim code sample to use it
   Dijkstra will still not be happy, but we avoid repetition
3. Implement an "AI" player for our Nim game
   Wikipedia says this is a solved problem :-)

Closing

A tiny imperative programming system

Conclusions

A tiny interactive imperative programming system

- Evaluation logic not that far from TinyML!
- Imperative interpreter needs much more state
- How exactly interactive editing worked?!
- Parsing & interactive editing out of scope :-(

Tomáš Petříček, 309 (3rd floor)
✉ petricek@d3s.mff.cuni.cz
➜ https://tomasp.net | @tomaspetricek
➜ https://d3s.mff.cuni.cz/teaching/nprg077

Charles University