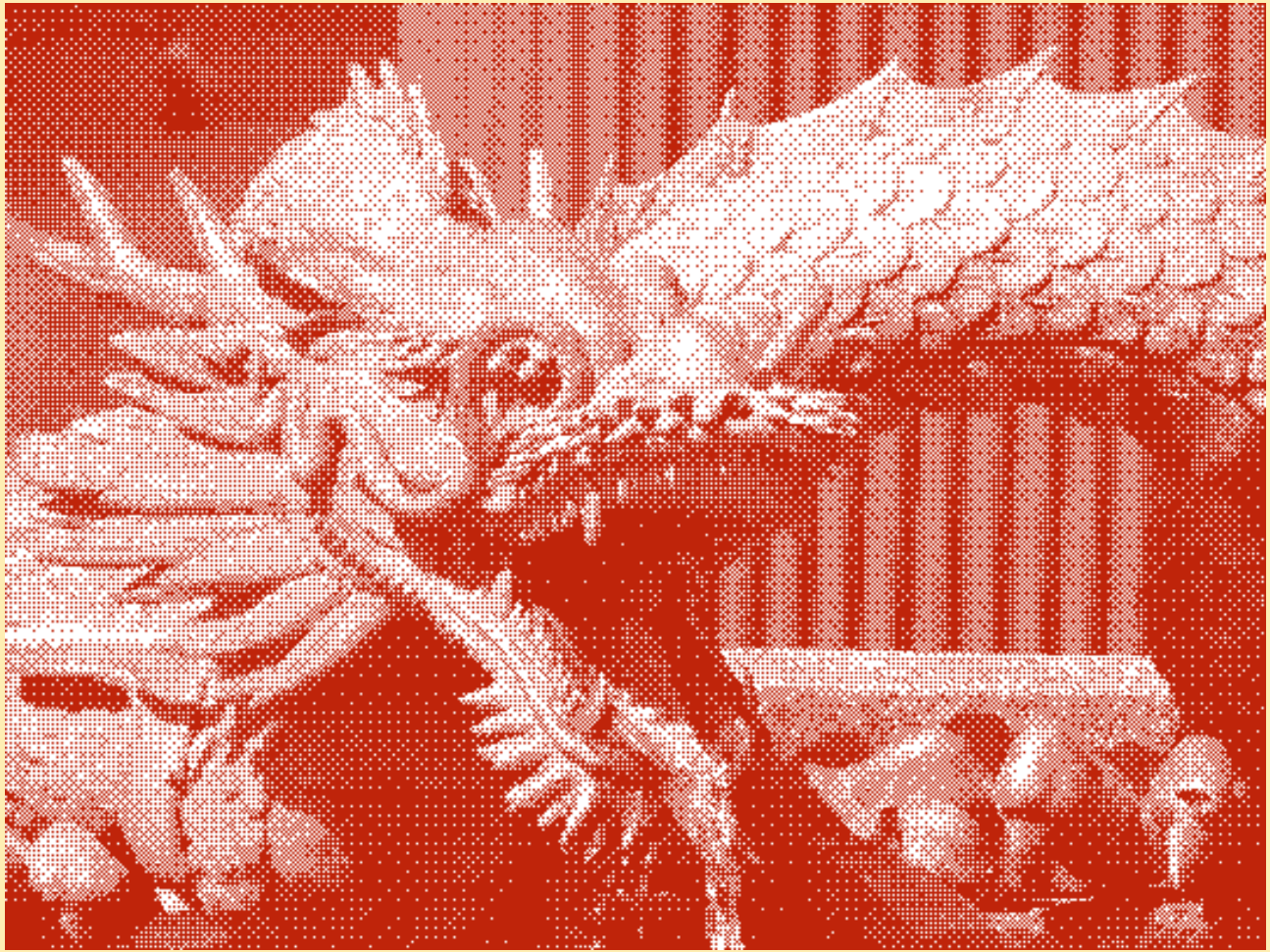
A tale of two compilers



Part I. The first compiler

Long ago, I was approached by a researcher in Japan about the possibility of accelerating meteorological code on FPGAs.

Japan!

『源氏は寺の後にある山を登っていた、
そして街の方を見た。
森は春の霧の中に消えてた。
「絵みたいだろう」、彼は言った。
「こんあ素晴らしい街に住む人は
どこにも行きたくないんで。」』

First contact

This was my first contact with FORTRAN 77

A source-to-source compiler

- There was no Fortran front end for LLVM
- LLVM itself was quite immature
- So I decided to write my own, case-specific source-to-source translator from FORTRAN 77 to Fortran 95
- Which language?

Which language?

- I know!
- I will write it in Haskell ...

Haskell!

Haskell?

- Compiling Haskell to a platform independent static binary?

無理

(impossible)

Haskell?

- Expecting my friend to compile Haskell code on his platform?

無理！

(impossible!)

a suitable alternative...

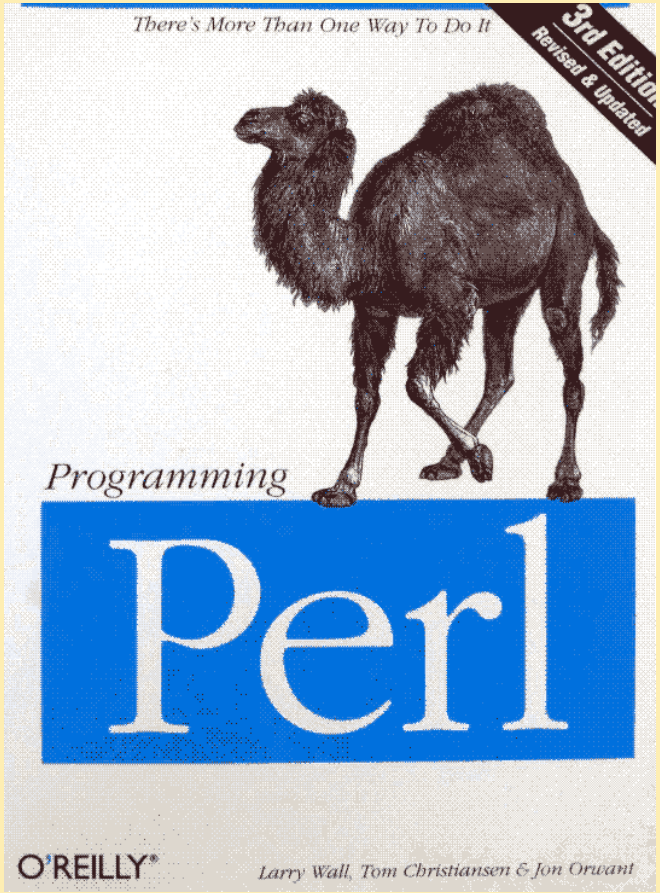
- So I needed a different language, platform-independent, ubiquitous, ...
- Luckily, there is such a language, very similar to Haskell …

very similar to Haskell ...

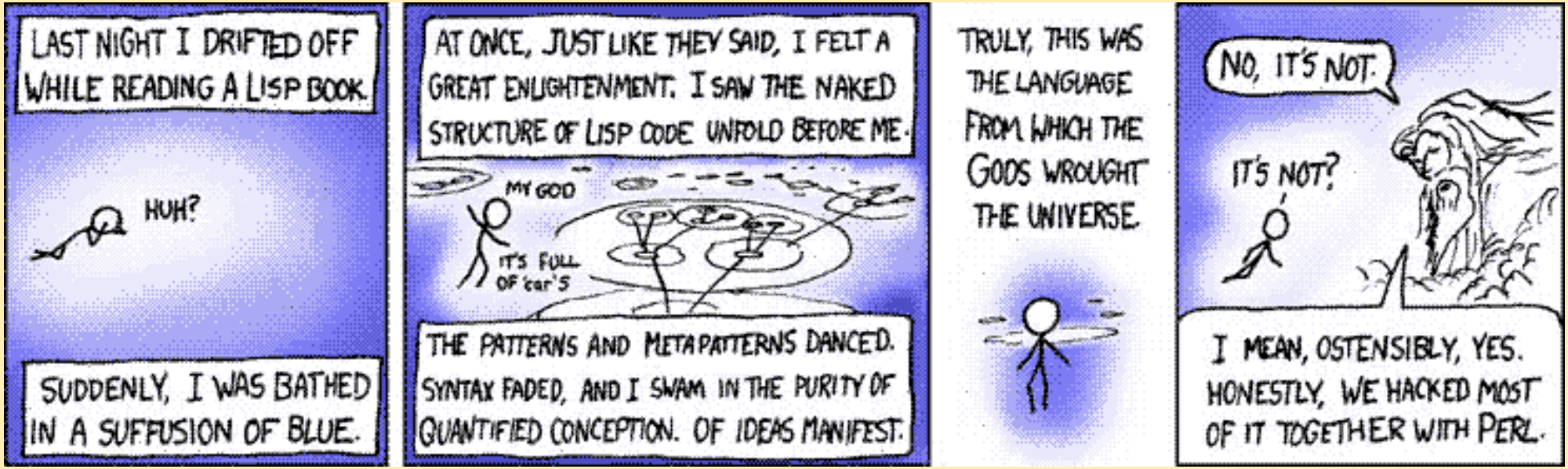
(apart from being
interpreted,
dynamically typed,
strict, and
whitespace-insensitve)

… Perl

language is ideology



Perl ...


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

So I wrote a source to source compiler from FORTRAN 77 to Fortran 90,

and from there to OpenCL and many other things besides.

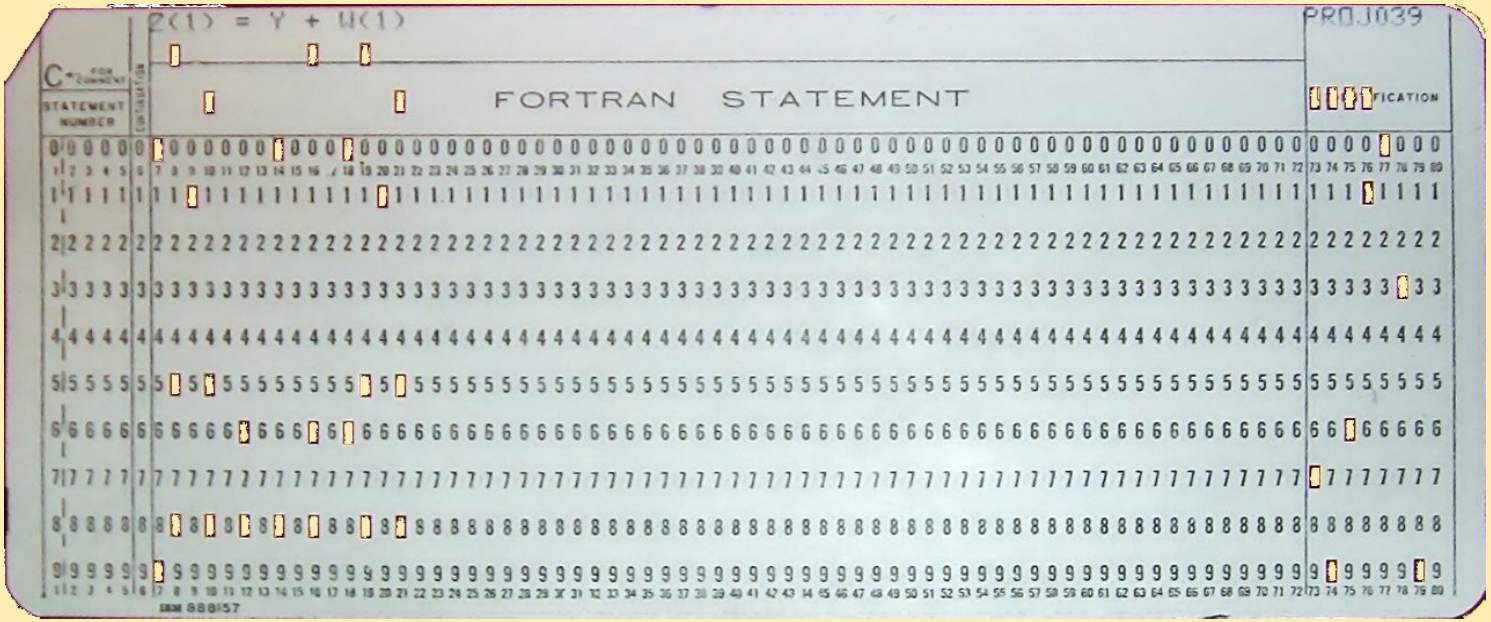
It was fun (and there was profit).

And it was hard.

But not because of Perl;

because of FORTRAN.

FORTRAN



- One card per statement, e.g. `Z(1) = Y + W(1)`
- One column per character

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–72)
C		PRIME NUMBER PROBLEM
100		PRINT 8
8		FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
	1	19X,1H1/19X,1H2)
101		I=3
3		A=I
102		A=SQRTF(A)
103		J=A
104		DO 1 K=2,J
105		L=I/K
106		IF(L*K-I)1,,2,4
1		CONTINUE
107		PRINT 5,I
5		FORMAT (I20)
2		I=I+2
108		IF(1000-I)7,,4,,3
4		PRINT 9
9		FORMAT (14H PROGRAM ERROR)
7		PRINT 6
6		FORMAT(31H THIS IS THE END OF THE PROGRAM)
109		STOP

from **IBM FORTRAN II General Information Manual**, 1961

FORTRAN

- Implicit typing: variable names starting with `I` to `N` are `INTEGER`, rest are `REAL`
- No pointers, but plenty of rope via `COMMON` and `EQUIVALENCE`
- “Playful” attitude to type safety
- Control flow via jumps to labels
- No dynamic allocation

FORTRAN 77

- Has n-dimensional arrays that know their size and shape
- And even limited higher-order function support
- There are type declarations
- And control flow is marginally better

Fortran 90/95

```
! Prime number problem
program PrimeNumberProblem
    use Prime, only: testPrime
    implicit none
    integer :: i,p
    print *, 'following is a list of prime &
              numbers from 1 to 1000'
    do i=3,1000,2
        p = testPrime(i,int(sqrt(real(i))),2)
        if (p /= 0) print *,p
    end do
    print *,'this is the end of the program'
end program PrimeNumberProblem
```

```
module Prime
    contains
    pure recursive integer &
      function testPrime(i,j,k) result(p)
        integer, intent(in) :: i,j,k
        if ((i/k)*k /= i) then
            if (k == j) then
                p = i
            else
                p = testPrime(i,j,k+1)
            end if
        else
            p = 0
        end if
    end function testPrime
end module Prime
```

Fortran 90/95

- Much better type annotations, with attributes
- Specifically: `intent`, `kind`
- Allows to define derived types (record types)
- Subroutines and functions can have explicit `pure` and `recursive` qualifiers
- Has pointers and dynamic allocation
- Has a decent module system

Fortran in 2024

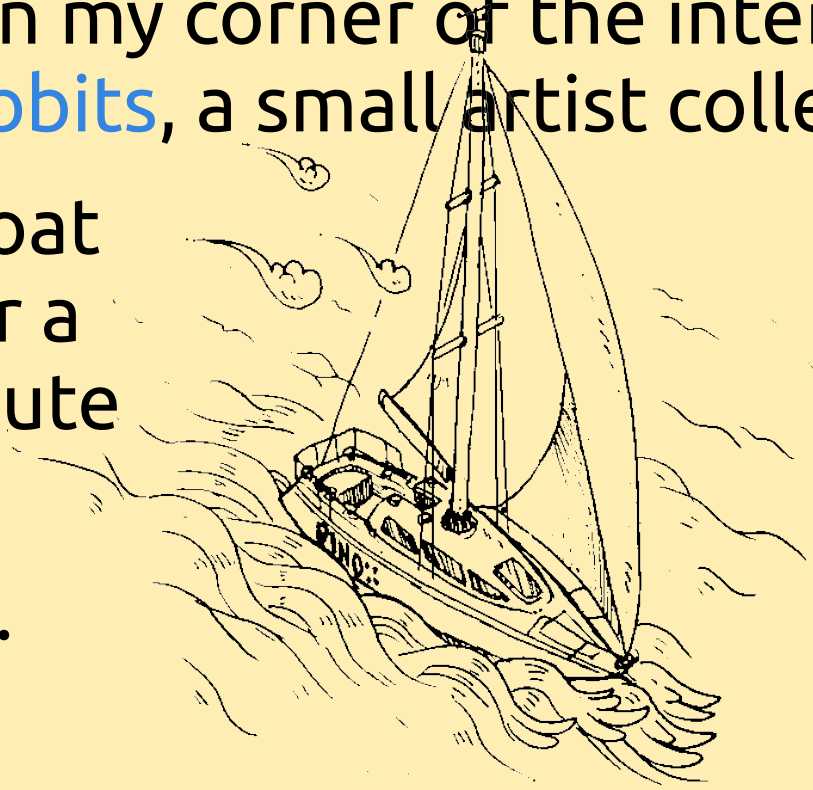
- Latest standard: ISO/IEC 1539:2023
- Released November 17, 2023
- But most code is still effectively F77/F95
- Still the dominant supercomputer workload

Part II. The second compiler

Many years later, in my corner of the internet,
I met Hundred Rabbits, a small artist collective.

They live on sail boat
and had a need for a
really frugal compute
stack.

So they made one.



The VM: Uxn

An 8-bit VM with 64 kB memory

Shared code and data

Stack-based, no registers

Two stacks of 256 bytes

I/O is done via devices

The language: Uxntal

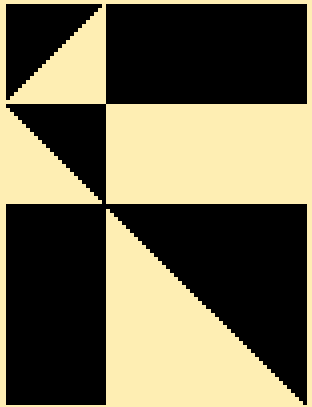
```
@isqrt ( num* -- sqrt* )
    ,&num STR2
    &while1 ,&bit LDR2 ,&num LDR2 GTH2 #01 SWP SUB ,&while2 JCN
            ,&bit LDR2 #02 SFT2 ,&bit STR2
       ,&while1 JMP
    &while2 ,&bit LDR2 #0000 EQU2 ,&done2 JCN
            ,&num LDR2 ,&res LDR2 ,&bit LDR2 ADD2 LTH2 ,&ift JCN
                 ,&num LDR2 ,&res LDR2 ,&bit LDR2 ADD2 SUB2 ,&num STR2
                 ,&res LDR2 #01 SFT2 ,&bit LDR2 ADD2 ,&res STR2
                 ,&if-done JMP
             &ift
                  ,&res LDR2 #01 SFT2 ,&res STR2
             &if-done
            ,&bit LDR2 #02 SFT2 ,&bit STR2
            ,&while2 JMP
    &done2
   ,&res LDR2
JMP2r
&num $2 &res $2 &bit 4000
```

Uxntal

- Assembly language
- Postfix syntax
- Not whitespace-sensitve
- Opcodes with modifiers (2,k,r)
- Non-word characters (“runes”) are used to indicate type of label/addressing

Funktal

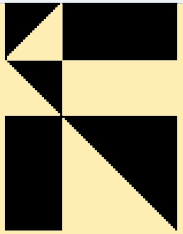
- I wanted to see if it was possible to create a statically typed functional language with ADTs and function types for this platform.
- It was
- Enter Funktal, an art project



Funktal features

* Stack-based, postfix syntax
* Statically typed, with type inference
* Algebraic data types and function types
* Unsound polymorphism
* Lambda functions and higher-order functions
* Impure IO and state

Funktal examples: lambdas



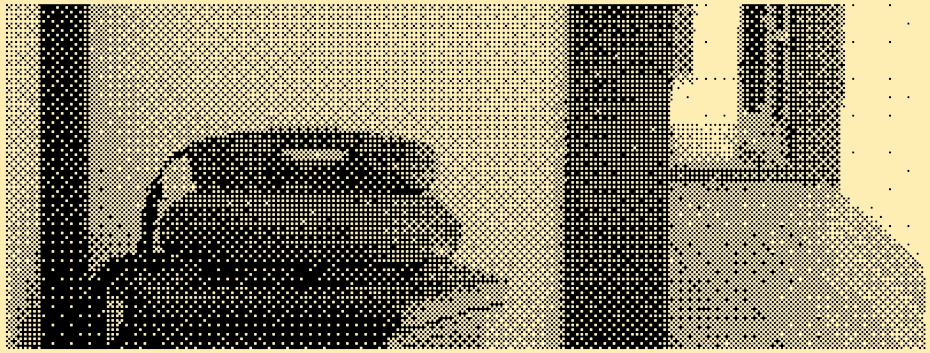
```
main {
    6 (\x. x x * x 2 * + x - print ) -- 42
    6 (\x. 7 (\y . x y * print ) ) -- 42
    2 84 '( / ) (\ x y div . y x div apply ) print -- 0x002a
}
```

Funktal examples: types

```
types {
        List = List Any Cons | Nil
}
functions {
        head = (\ Any <- (xs x Cons) . x )
        tail = (\ List <- (xs x Cons) . xs )
        fold = (\ Any <- lst : List <- acc : Any  <- f : Any .
                '( acc )
                '( lst tail acc lst head f apply f fold )
                lst 'Nil is if
        )
}
```

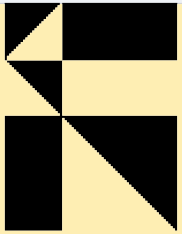
The compiler

* A language is only frugal if the compiler is frugal
* Written in retro old-skool Fortran
* To keep memory usage low
* And make porting to Uxntal straightforward



The internals

* Only data structure is the
    1-dim array of 2-byte ints
* So a lot of indirection
* Also lots of bitmaps
* And GOTOs



Harm, evil, purity and foolishness

- "the **go to** statement should be abolished"

  -- E. W. Dijkstra, 'Go To Statement Considered Harmful', Comm. ACM, 1968

- "Global variables are evil"

  -- P. Koopman, 'Better Embedded System Software', Ch 19, 2010

- Goto: "not for the faint of heart (nor for the pure of heart)"
  Global variables: "help you not to use global variables--or at least, not use them foolishly."

  -- paraphrased from 'Programming Perl' by L. Wall et al., 2000

Part III. The first compiler, revisited

The Funktal compiler ended up being ~8,000 lines of code. Too much to port manually; also, what's the fun?

So I started on an Uxntal backend for my Fortran compiler, to compile the compiler to Uxntal.

Fortran to Uxntal

- *“Fortran argument association is* usually *similar to call by reference and call by value-result.” (ISO/IEC 1539-1:2004)*
- Function calls and recursion: we don’t want stack frames by default, too much overhead. So we analyse the code and only use stack frames for non-tail recursion.
- Most other issues are to do with string manipulation.

Keeping it small

- Constant propagation, constant folding (easy with `eval`), branch elimination
- Only use a call stack when necessary. Therefore, static analysis of tail recursion.

  But the only tail call optimisation is not to use a call stack
- Minimal stack frame size calculated at compile time

Keeping it small

- Precomputed stack size

```
@reconstructTypeNameExpr
    push-frame
    #021e stack-alloc
    ( … )
    !pop-frame
```

Keeping it simple

- Function arguments passed via the stack but stored in memory unless a call stack is necessary. The assembler* will optimise this.

```
@add
    ADD
  JMP2r
```

```
@add ,&x STR ,&y STR
     ,&x LDR ,&y LDR ADD ,&z STR
     ,&z LDR JMP2r
&x $1 &y $1 &z $1 ( static alloc )
```

(*yes, I wrote an assembler as well, but that's another story)

Keeping it simple

- Use of higher-order functions and Uxntal's lambda functions, e.g.

```
map (\iter→ rawhex8toStr_csu[iter]=0) 2 .. 8
```

```
{
  #00 ROT ROT ;rawhex8toStr_csu ADD2 STA
  JMP2r
} STH2r
#0008 #0002 range-map-short
```

Coda: FRACTRAN

- An esoteric programming language created by John Conway
- A FRACTRAN program consists of and ordered list of positive fractions and an initial positive integer input n. The program is run by updating n as follows:
  - for the first fraction f in the list for which $n.f$ is an integer, replace n by $n.f$
  - repeat this rule until no fraction in the list produces an integer when multiplied by n, then halt.

[1] Conway, John H. "Fractran: A simple universal programming language for arithmetic." *Open problems in Communication and Computation*. New York, NY: Springer New York, 1987. 4-26.

Fractran example: Fibonacci

$$2^5.3.5.7$$

$$\frac{11}{7.2}, \frac{11.19}{11.3}, \frac{11.17.19}{11.5}, \frac{13}{11}, \frac{13.3}{13.17}, \frac{13.5}{13.19}, \frac{7}{13}, \frac{1}{3}, \frac{1}{7}$$

$$5^{13}$$

Fractran virtual machine

- The fractions are an obfuscation: the FRACTRAN interpreter can be viewed as a machine with an infinite number of registers
- The program is an ordered list of terms; the terms are tuples of registers sets (t,n).
- The input to the program is a separate register set acc (the accumulator).

Fractran virtual machine

- The rule for updating *acc* becomes:
  - Step 1: check if the term matches the accumulator.
    - If a register from *n* occurs in *acc*, the value in acc must be greater than or equal to that in *n*.
    - If that condition is met, go to Step 2
  - Step 2: update the accumulator
    - Increment the registers in *acc* with the values of the corresponding registers in *t*;
    - Decrement the registers in *acc* with the values of the corresponding registers in *n*;
    - Any reg in *t* that was not in *acc* is added to *acc*
  - Step 3: run the program again for as long as one of the terms matches the accumulator.

Fractran as a rewrite system

If the LHS matches, execute the rule (dec based on LHS, inc based on RHS)

-- Keep going while the n register is present

`fib n ⇒ fib`$_{sft}$`,`

-- Shift the scrolling window to show two numbers

`fib`$_{sft}$ `last ⇒ fib`$_{sft}$ `B,fib`$_{sft}$ `res ⇒ fib`$_{sft}$ `A B, fib`$_{mv}$ `⇒ fib`$_{sft}$`,`

-- Move the temporary registers back by one number

`fib`$_{mv}$ `A ⇒ fib`$_{mv}$ `last, fib`$_{mv}$ `B ⇒ fib`$_{mv}$ `res, fib`$_{mv}$ `⇒ fib,`

-- Cleanup temporary registers at the end of the program

`last ⇒, fib ⇒`

See https://wiki.xxiivv.com/site/fractran for more info

Fractran in Fortran and Uxntal

- The core of the interpreter is about 50 lines of Fortran 90
- The Fortran to Uxntal compiler can handle this
- And that is my tale so far

Thank you!

Code

- rf4a: github.com/wimvanderbauwhede/RefactorF4Acc
- uxn: git.sr.ht/~rabbits/uxn
- funktal: codeberg.org/wimvanderbauwhede/funktal
- yaku: codeberg.org/wimvanderbauwhede/yaku
- furakutoran: codeberg.org/wimvanderbauwhede/furakutoran