# Concepts of programming languages

## Janus

Joris ten Tusscher, Joris Burgers, Ivo Gabe de Wolff, Cas van der Rest, Orestis Melkonian

# Example

## `fib`: calculates (n+1)-th and (n+2)-th Fibonacci number

```
procedure fib
    if n = 0 then
        x1 += 1     ; -- 1st Fib nr is 1.
        x2 += 1     ; -- 2nd Fib nr is 1.
    else
        n -= 1
        call fib
        x1 += x2
        x1 <=> x2
    fi x1 = x2
```

# Example

`fib`: calculates (n+1)-th and (n+2)-th Fibonacci number

```
procedure fib
    if n = 0 then
        x1 += 1     ; -- 1st Fib nr is 1.
        x2 += 1     ; -- 2nd Fib nr is 1.
    else
        n -= 1
        call fib
        x1 += x2
        x1 <=> x2
    fi x1 = x2      ; -- Why do we need this?
```

# Example

## `fib`: calculates (n+1)-th and (n+2)-th Fibonacci number

```
procedure fib
    if n = 0 then
        x1 += 1     ; -- 1st Fib nr is 1.
        x2 += 1     ; -- 2nd Fib nr is 1.
    else
        n -= 1
        call fib
        x1 += x2
        x1 <=> x2
    fi x1 = x2      ; -- Why do we need this?
```

- Q: How do we calculate the inverse?

# Example

`fib`: calculates (n+1)-th and (n+2)-th Fibonacci number

```
procedure fib
    if n = 0 then
        x1 += 1     ; -- 1st Fib nr is 1.
        x2 += 1     ; -- 2nd Fib nr is 1.
    else
        n -= 1
        call fib
        x1 += x2
        x1 <=> x2
    fi x1 = x2      ; -- Used for inverting the if-statement.
```

- Q: How do we calculate the inverse?

$\mathcal{I}[\![\texttt{if } e_1 \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi } e_2]\!] \quad = \texttt{if } e_2 \texttt{ then } \mathcal{I}[\![s_1]\!] \texttt{ else } \mathcal{I}[\![s_2]\!] \texttt{ fi}$

# Example

## `fib`: calculates (n+1)-th and (n+2)-th Fibonacci number

```
procedure fibInverse
    if x1 = x2 then
        x2 -= 1             ; -- 2nd Fib nr is 1.
        x1 -= 1             ; -- 1st Fib nr is 1.
    else
        x1 <=> x2
        x1 -= x2
        call fibInverse
        n += 1
    fi n = 0
```

# Example

## `fib`: calculates (n+1)-th and (n+2)-th Fibonacci number

- Q: What does the inverse of fib do?

```
procedure fibInverse
    if x1 = x2 then
        x2 -= 1             ; -- 2nd Fib nr is 1.
        x1 -= 1             ; -- 1st Fib nr is 1.
    else
        x1 <=> x2
        x1 -= x2
        call fibInverse
        n += 1
    fi n = 0
```

# Relational Programming

| **Injective Programming** | **Relational Programming** |
|---|---|
| $r$-Turing Complete | Turing Complete |
| backwards deterministic | backwards non-deterministic |
| restricted language constructs | search procedure (aka *resolution*) |

# Prolog basics

A logic programs consists of *facts* and *rules*.

```
parent(alice, joe).
parent(bob, joe).
parent(joe, mary).
parent(gloria, mary).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

descendant(X, Y) :- ancestor(Y, X).
```

The user can then *query* the runtime system, as such:

```
?- parent(X, joe).
X = alice;
X = bob.

?- ancestor(X, mary).
X = joe;
X = gloria;
X = alice;
X = bob.

?- ancestor(X, mary), descendant(X, alice).
X = joe.
```

# Demonstration - Type Inference

Assume a `type` predicate, relating expressions with types:

```
type(expr, t) :- ... .
```

You would normally use it to perform *type-checking*:

```
?- type(1 + 1, int).
true.
?- type(1 + 1, string).
false.
```

But you can also performing *type-inference*:

```
?- type(1 + 1, Type).
Type = int.
?- type("hello world", Type).
Type = string.
?- type(\x:int -> x, Type).
Type = int -> int.
?- type(\x -> x, Type).
Type = ?42 -> ?42.
?- type(\x -> x, int -> Type).
Type = int.
```

Going in the reverse direction, you can query the expression:

```
?- type(Expr, int).
Expr = 1;
Expr = 2;
...
Expr = 1 + 1;
Expr = 1 + 2;
...
Expr = if true then 1 else 1;
...
```

Of course, this does not make much sense without a sufficiently expressive type system.

# Demonstration - Program Synthesis

Assume you have implemented a *relational interpreter*:

```
eval(program, result) :- ... .

?- eval(map (+ 1) [1 2 3], Result).
Result = [2 3 4].
```

But you can also perform *program synthesis* by-example:

```
?- eval(F 1, 2),...,eval(map F [1 2 3], [2 3 4]).
...
F = \x -> x + 1;
...
F = \x -> x - 10 + 10 + 1;
...
```

Quine generation is pretty straightforward:

```
?- eval(Quine, Quine).
...
Quine = (\a -> a ++ show a) "(\\a -> a ++ show a) ";
...
```

# Logic Programming IRL

In practice, bi-directionality breaks with the usage of *extra-logical* features:

- **Variable projection**: inspecting values at runtime
- **Cut (!)**: disables backtracking in certain places
- **Assert/Retract**: Dynamically insert/remove facts

**MiniKanren** is a more recent logic programming language, which avoids extra-logical features (as much as possible).

# Higher abstraction

- Relational programming, as well as functional programming, both belong to the *declarative* paradigm.
- They both raise the level of abstraction, by enabling the programmer to express *what* needs to be done, instead of *how*.

**Question**

*How can we combine them, to get the best of both worlds?*

# Hanus: Janus embedded in Haskell

In our research project, we use *TemplateHaskell* and *QuasiQuotation* to embed Janus in Haskell:

```
[hanus|
  procedure encode(im :: Image, ret :: [Byte]) {
    -- Janus commands containing Haskell code
    -- e.g. janus_variable += <Haskell code>
  }
|]
encode :: Image -> [Byte]
encode = call encode
decode :: [Byte] -> Image
decode = uncall encode
```

Come and check out our poster in de Vagant!

# Thanks!

Feel free to ask any questions