



Hackhofer



DSL in the insurance business

Motivation

Objectives

TreeCalc language

Implementation

TreeCalc



Hackhofer Software GmbH
1070 Wien
www.hackhofer.com

DI Stefan Neubauer, 16.04.2026



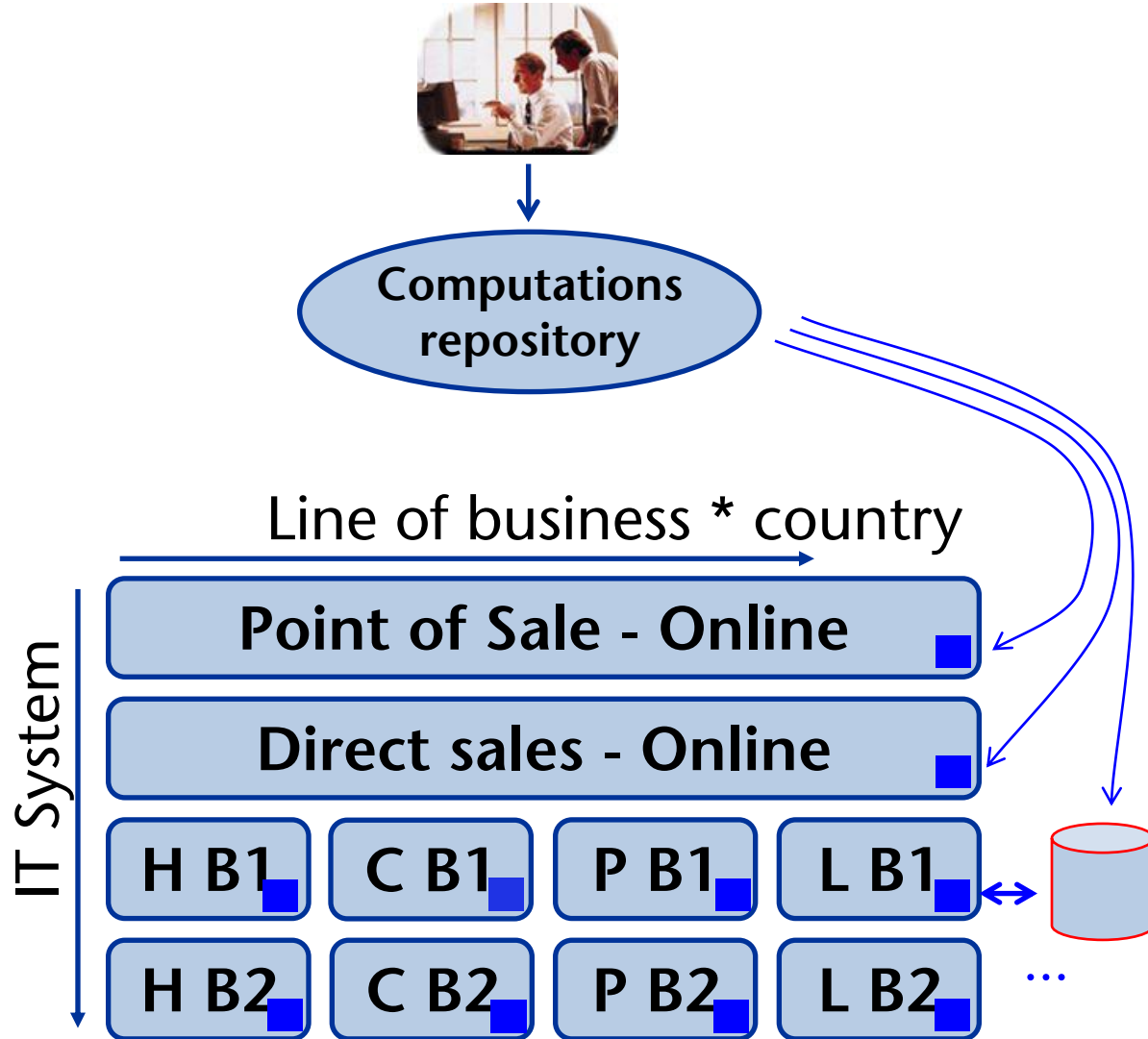
Motivation – Overview UNIQA Group

- 14 countries
- 20.000 employees
- 17 mn customers
- € 7.840 mn premium (2024)
- € 347 mn net profit (2024)



<https://www.uniqagroup.com/grp/about-us/uniqa-europe/europe/europa.de.html>
<https://www.uniqagroup.com/grp/about-us/companyportrait/unternehmenskennzahlen.de.html>

Motivation – business view



- VP/MS ® from DXC Technology
 - Main calculation engine for the UNIQA group
 - Modelling part: Eclipse based; used by actuaries / business units
 - Execution part: Native library (Windows, Linux, z/OS, AS/400)
 - Challenges / Cons
 - Proprietary, enterprise license model
 - Hard times when instabilities/errors occur with new versions
- TreeCalc
 - Open Source → <https://github.com/treecalc>
 - Safer, better, faster 😊
 - Conversion from VP/MS models to TreeCalc implemented (not open source)
 - Playground for memoization

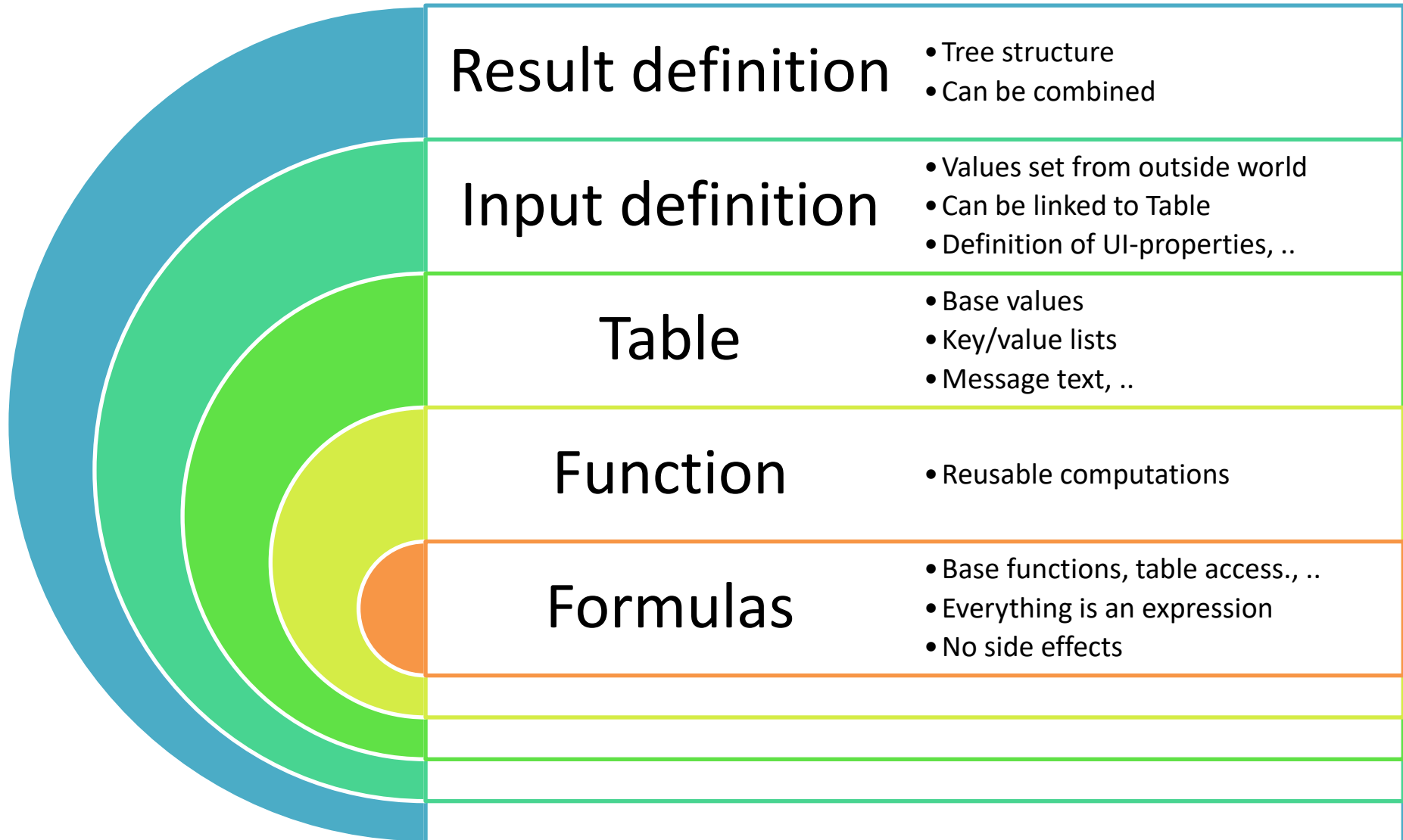


- Language
 - Declarative, „safe“: no destructive assignment, no loops, ..
 - Text format (diff/merge with git etc. for free)
 - Calculations organized in trees
- Simple API
 - set value
 - compute result
 - get list of key/value pairs
 - input needed?
- Execution
 - Optimal integration for: Java, JavaScript
 - Fast, safe, correct, scaleable

- Internal DSL / Embedded DSL
 - Hosted: Scala, Haskell, (Meta)OCaml, Lisp (Clojure, ...), ...
 - Fluent interface: Java, C#, ...
 - Main pro: „cheap“
 - Main cons: highly dependent on host language + implementation

External DSL

- Custom syntax, custom parsing
- Semantic model
- Interpretation / Code generation
- Main pro: most flexible
- Main cons: higher effort



- Declarative / functional
 - No destructive assignment
 - Results only dependent on input values
 - Referential transparency
 - Recursion instead of loop
 - Everything is an expression
- Data types
 - Number (IEEE 754 binary double), String, Boolean, Date, List
 - Dynamic, weakly typed



Language – Example input definition

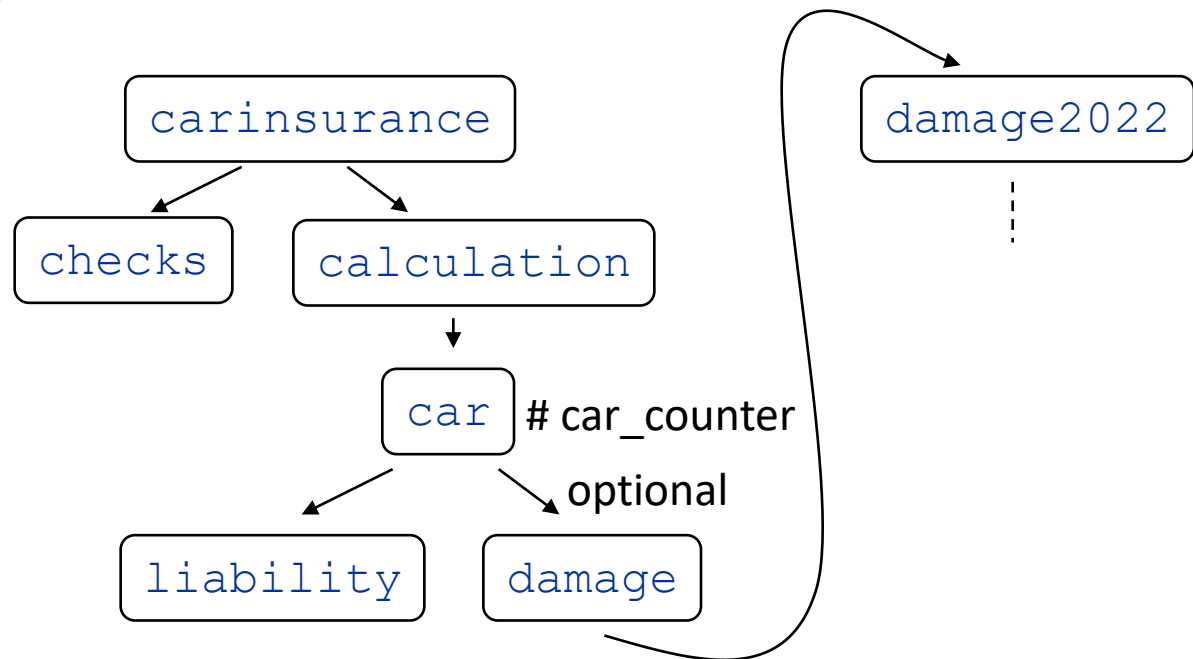
```
INPUT I_DateOfBirth ;
```

```
INPUT I_Damage_Sum {  
    visible = I_Damage_YN ;  
    default = 10000 ;  
    list = T_Damage_Sum ;  
    select = key=1 || F_Age(I_DateOfBirth) > 30 ;  
}
```

```

TREE carinsurance {
  NODE checks ;
  NODE calculation {
    NODE car TIMES I_CarCounter {
      NODE liability ;
      NODE damage IF I_Damage_YN {
        LINK damage2022 ;
      }
    }
  }
}

```



Language – Example result definition

```
CALC carinsurance.calculation {  
  RX_Prem = R_Prem  
  *  
  IF I_Discount_YN THEN  
    0.8  
  ELSE  
    1  
  ENDIF ;  
}  
  
CALC carinsurance.calculation.car.liability {  
  R_Prem = I_kw * T_Area[I_Area].fact ;  
  ...  
}  
  
CALC damage2022.calculation {  
  R_Prem = ...  
}
```

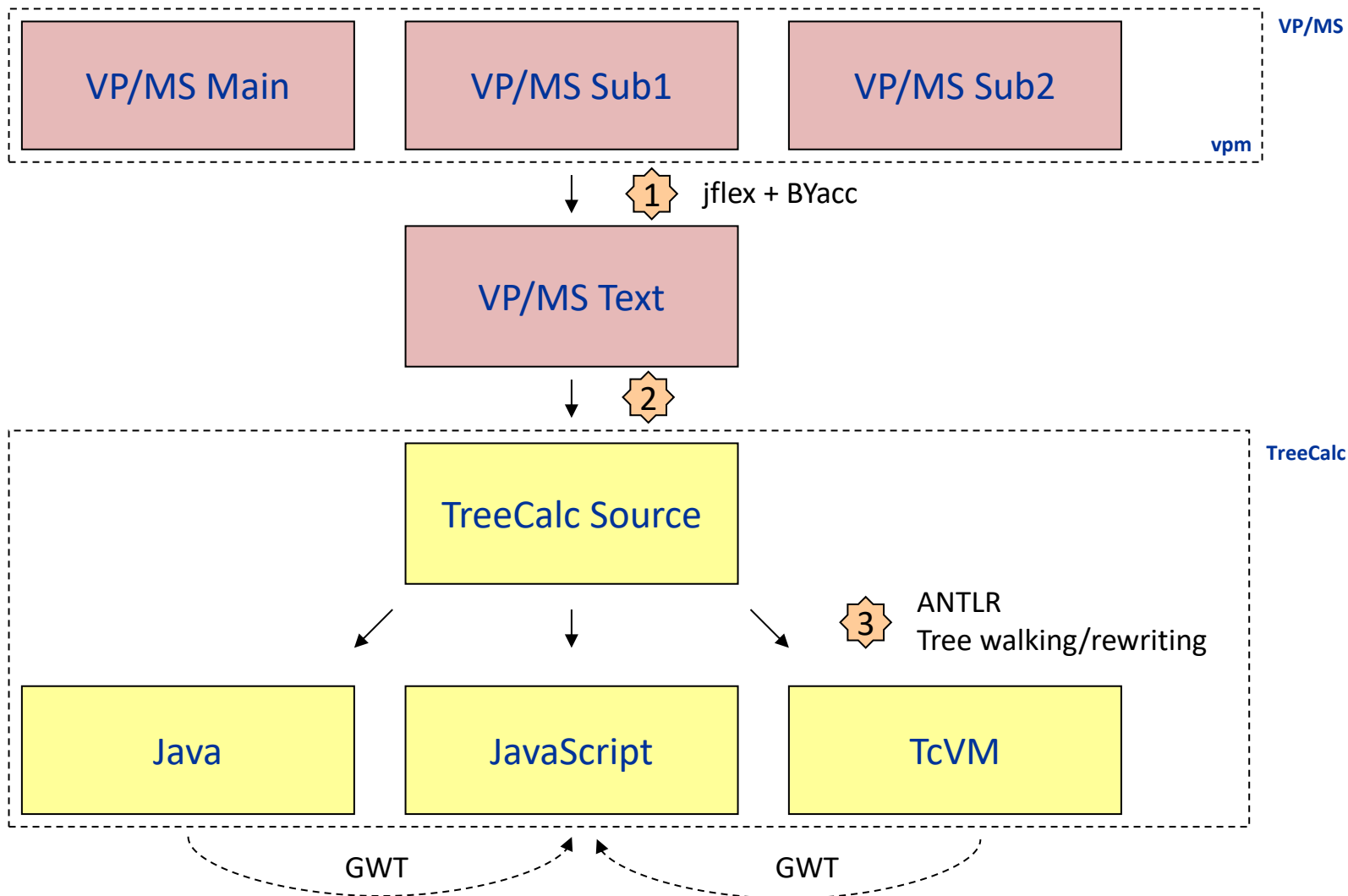


Language – Example table definition

```
TABLE T_Mortality (age, qx, qy) {  
    16, 0.0006380, 0.0003980 ;  
    17, 0.0007200, 0.0004160 ;  
    18, 0.0007760, 0.0004060 ;  
    19, 0.0008060, 0.0003720 ;  
    20, 0.0008400, 0.0003580 ;  
    ...  
}
```

```
TABLE T_Liability_Sum (key, text) {  
    1, "€ 6.000.000,-" ;  
    2, "€ 12.000.000,-" ;  
}
```

Implementation – Overview



- ANTLR


- Lexer + Parser + Tree construction
- LL(*), semantic/syntactic predicates
- Nice grammar, lots of tools, automatic error correction, ...
- Generated code a lot bigger than lex/yacc & co

- Implementation strategies

- Homogeneous AST
- External visitor for passes
 1. Scopes
 2. Symbol table
 3. Resolve names + rewrite AST
 4. Create semantic model

```
void visitNodes(Ast node) {  
    switch (node.getType()) {  
        case TT_COMPUNIT: {  
            List<Ast> c = node.getChildren();  
            for (Ast child : c) {  
                visitNodes(child);  
            }  
            break;  
        }  
        case KEYWORD_TREE: {  
            ...  
        }  
    }  
}
```

- `out.print(...)` 😊
- Simple because of semantic model + AST
 - Same model and helper classes
 - Separate writer classes for Java, JavaScript, TcVM
- Formulas
 - intermediate vars `_1`, `_2`, ...
 - AST node
 - optional: `out.print(...)`
 - expression string (short expression or varname)



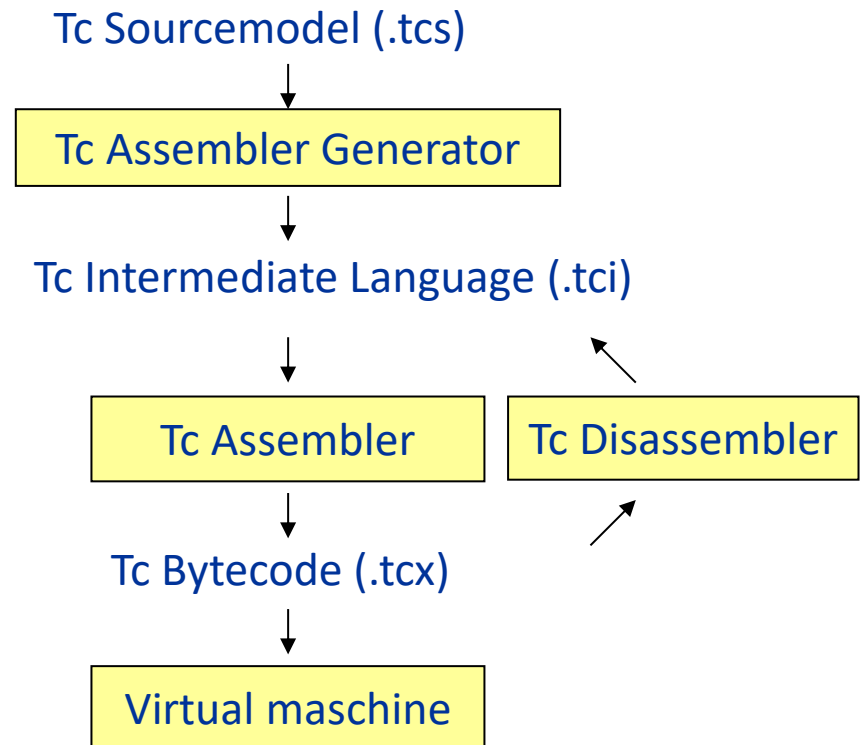
```
F_LI_Lx(age; sex; risk):  
  if(age <= 0;  
    100000;  
    F_LI_Lx(age-1; sex; risk) * (1 - F_LI_qx(age-1; sex; risk))  
  )
```

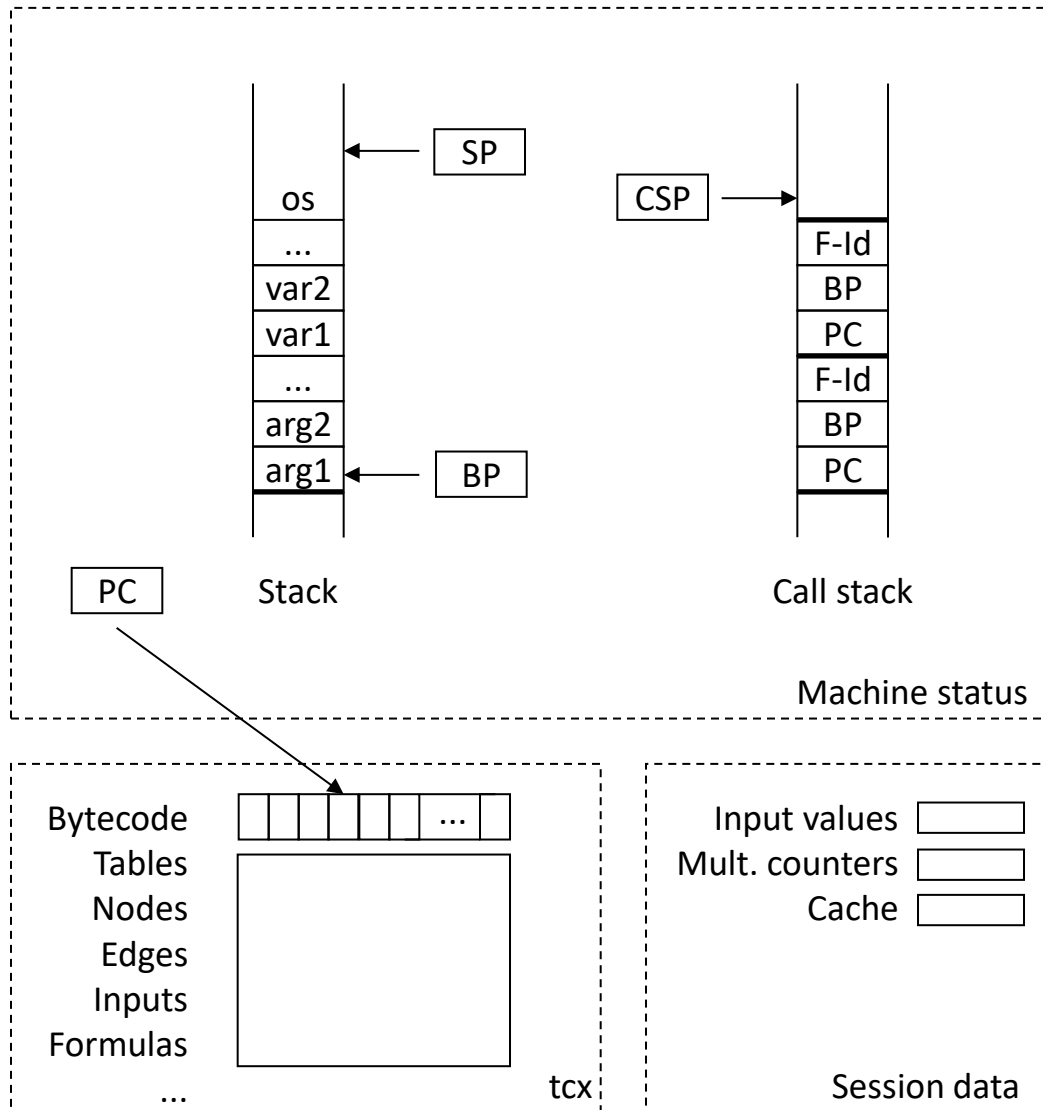
```
static final V F_LI_LX(S _s, V age, V sex, V risk) {  
  Object cacheKey = _s.getCacheKey(8156345, age, sex, risk);  
  V ret = _s.readCache(cacheKey);  
  if (ret!=null) { return ret; }  
  V _1;  
  V _2 = age.smleq(_i0);  
  if (_2.booleanValue()) {  
    _1 = _i100000;  
  } else {  
    V _3 = age.sub(_i1);  
    V _4 = age.sub(_i1);  
    V _5 = _i1.sub(F.F_LI_QX(_s, _4, sex, risk));  
    V _6 = F.F_LI_LX(_s, _3, sex, risk).mult(_5);  
    _1 = _6;  
  }  
  ret = _1;  
  _s.writeCache(cacheKey, ret);  
  return ret;  
}
```

- Performance tweaks
 - Internal tables: sorted or direct access $\rightarrow O(1)$ or $O(\log n)$
 - final static methods, constant pool, ids instead of names
 - Bit sets for node structure
 - dynamic dispatch (calc, table, function, ..)
 - `switch(id) { ... }` instead of reflection
 - Caching
 - LRU instead of HashMap
 - random id; fast key object creation
 - no caching for simple formulas (heuristic)
- Handicaps
 - 16-bit limits \rightarrow split methods/classes
 - Big code blocks for dynamic dispatch

- Some things easier
 - Dynamic dispatch
 - No 16-bit limits
- Some things harder (few years ago)
 - Missing base libraries: HashMap, ..
 - Development environment
- Handicaps
 - Large TreeCalc models → big JavaScript source

- Why
 - Optimized for size
 - Prototype for tariff calculator
 - Easy to play around with
 - Memoization strategies
- Implementation
 - TcVM implemented in Java
 - For some models too slow





```
.func func=1 name=F_FACT args=1 simple=false formula=1
.formula formula=1 ; line 6830
  //start of if statement, line 6830
  : load 0 ; N
  : pushconst 0
  : cmpsmleq
  : iffalse L0
  : pushconst 1
  : goto L1
L0:
  : load 0 ; N
  : dup
  : pushconst 1
  : sub
  : callfunc 1 ; F_FACT
  : mult
L1:
  //end of if statement
  : return
```

- Numbers
 - Functionality: 100 %
 - Performance: 10 times slower
 - Size: 15 % of generated Java class file size
- Code generation
 - Text representation (.tci) proved to be useful
 - Generation simpler than Java/JavaScript source code generation
- Implementation
 - Tree access with TcVM „macro programs“, not neatest solution
 - Base classes reused from Java implementation



- Extensions
 - VP/MS to TreeCalc converter Open Source (clarify legal situation first)
 - Partial generation (e.g. just for UI-control)
 - Modelling environment (Xtext, ...)
- Organization/Community
 - Build up / extend user group and contributors
 - Landing page, Wiki, Tutorial, Bug tracker, ...
 - Books, Training and Certifications ;-)
- Improvements
 - Tooling: build chain, maven plugin, ... (learn from Clojure, Frege, ..)
 - Add modules concept
 - Test cases + examples
 - Upgrade from ANTLR 3 to ANTLR 4
 - Improve JavaScript port
 - Implement with Truffle / GraalVM
 - Performance optimizations: Type inference, dynamic memoization, ..
 - Implement features from VP/MS® Runtime XE

Interested? Contact sneubauer@gmx.at

- Domain Specific Languages, Martin Fowler, Addison-Wesley, 2010
- Language Implementation Patterns, Terence Parr, Pragmatic Bookshelf, 2009
- The Definitive ANTLR Reference, Terence Parr, Pragmatic Bookshelf, 2007
- <http://www.antlr.org/>
- <https://github.com/treecalc>