# Selected topics in software development

**Today:**

Notation (domain-specific languages)

**Speaker:**

Jyrki Katajainen

**Course home page:**

http://www.diku.dk/forskning/performance-engineering/Software-development/

# Why to talk about notation?

The difference between a regular expression and a C++ program is big, but both are just notations for solving problems. Good notation makes it easier to say what we want and harder to say the wrong thing by mistake.

As tasks become so focused and well understood that programming them feels almost mechanical, it may be time to create a notation that naturally expresses the tasks and a language that implements it.

[Brian W. Kernighan & Rob Pike, The practice of programming, §9]

# Format specifiers

**Where used:** In `printf` control sequences and packet handling in a network protocol

```
int pack_type1(uchar* buffer, ushort count, uchar value, ulong data) {
  uchar* bp = buffer;
  *bp++ = 0x01;
  *bp++ = count >> 8;
  *bp++ = count;
  *bp++ = value;
  *bp++ = data >> 24;
  *bp++ = data >> 16;
  *bp++ = data >> 8;
  *bp++ = data;
  return bp - buffer;
}

int pack_type1(uchar* buffer, ushort count, uchar value, ulong data) {
  return pack(buffer, "cscl", 0x01, count, value, data);
}
```

[Brian W. Kernighan & Rob Pike, The practice of programming, §9.1]

# Specification languages

**Where used:** In FreeBSD kernel to specify the types of function call arguments and their locking pre- and postconditions

```
#
#% read   vp   L L L
#
vop_read {
  IN struct vnode* vp;
  INOUT struct uio* uio;
  IN int ioflag;
  IN struct ucred* cred;
}
```

588 lines of domain-specific code $\longrightarrow$
4 339 lines of C code and declarations

An Awk script creates:

- C code for packing the arguments into a single structure
- Declarations for structures holding the arguments and functions doing the work
- Initialized data specifying the contents of the structures
- The C code for implementing file-system layers
- Assertions for verifying the state of the locks when the function enters and exits.

[Diomidis Spinellis, Beautiful code, §17]

# Regular expressions

**Where used:** In many Unix tools like `lex`, `grep`, `awk`

```
jyrki@Per$ cd ~/CPHSTL/Presentation
jyrki@Per$ grep "landscape-slides" */*.tex

jyrki@Per$ cd ~/CPHSTL/Release/List-based-stack
jyrki@Per$ ~/CPHSTL/Script/search_and_replace * "\\bC\\b" "R"
/home/jyrki/CPHSTL/Release/List-based-stack/list_based_stack.c++
  template <typename E, typename A, typename C>
C ==> R: (y/n/!/q)?
```

[Brian W. Kernighan & Rob Pike, The practice of programming, §9.2]

# Scripting languages

**Where used:** In non-performance-critical software development when programmer productivity is of prime importance

```
"""
split.py: split input into one word per line
"""

import sys
import re

input_file = sys.argv[1]
h = open(input_file, 'r')
content = h.read()
h.close()
result = re.sub(r'\s+', '\n', content)
print result
```

The book shows how to implement a simple web server as a single-line shell script using Tcl, Perl, and Awk.

[Brian W. Kernighan & Rob Pike, The practice of programming, §9.3]

# Grammars

**Where used:** In `yacc` parser generator

```
program:
   program statement '\n'
   |
   ;

statement:
   expr                  { printf("%d\n", $1); }
   | VARIABLE '=' expr   { symboltable[$1] = $3; }
   ;

expr:
   INTEGER
   | VARIABLE            { $$ = symboltable[$1]; }
   | expr '+' expr       { $$ = $1 + $3; }
   | expr '-' expr       { $$ = $1 - $3; }
   | expr '*' expr       { $$ = $1 * $3; }
   | expr '/' expr       { $$ = $1 / $3; }
   | '(' expr ')'        { $$ = $2; }
   ;
```

[Brian W. Kernighan & Rob Pike, The practice of programming, §6.4]

# Semantic comments

**Where used:** In PostScript (%%), in Java to create documentation (/**), and in literate-programming tools

Andy's way of writing C++ code to check that the compiler catches program errors. The actual checking is done by a combination of shell and Awk scripts.

```
int f() {}
  /// warning.* non-void function .* should return a value

void g() {return 1;}
  /// error.* void function may not return a value
```

[Brian W. Kernighan & Rob Pike, The practice of programming, §9.5]

# Specialized languages for book writing

**Where used:** In tools used for producing graphs (`gnuplot`), pictures (`pstricks`), tables, mathematical formulas (`eqn`), and indices.

[Brian W. Kernighan & Rob Pike, The practice of programming, §9.5]

# Visual languages

**Where used:** In visual development systems and wizards that synthesize user-interface code out of mouse clicks.

[Brian W. Kernighan & Rob Pike, The practice of programming, §9.5]

# Macros

**Where used:** In C preprocessor, LaTeX, m4

```
#define MEASURE(code) {                          \      int main(void) {
  std::cout << "Testing routine "                 \
            << #code                               \        const unsigned int n = 1000000;
            << " ..." << std::endl;                \        char a[n], b[n];
  clock_t start = clock();                         \
  code;                                            \        TEST(::strcpy, a, b, n);
  clock_t ticks = clock() - start;                 \        TEST(::copy_string, a, b, n);
  float s = 1.0 / float(CLOCKS_PER_SEC);           \        TEST(std::strcpy, a, b, n);
  std::cout << "Running time (s): "                \
            << s * float(ticks)                    \        return 0;
            << std::endl;                          \      }
}
#define TEST(routine, x, y, n)                     \
  generate(x, n, 'x');                             \
  generate(y, n, 'y');                             \
  MEASURE((void) routine(x, y));                   \
  assert(strcmp(x, y) == 0)
```

[Brian W. Kernighan & Rob Pike, The practice of programming, §9.6]

# On-the-fly code generation

**Where used:** In performance-critical image processing (e.g. `BitBlt` in Windows 1.0) and just-in-time compilers

The idea is to generate specialized code on the fly for the particular situation so that handling of any special cases can be avoided. For example, write a mini compiler that translates the current regular expression into special code optimized for that expression.

```
int matchchar(int literal, char* text) {
  return *text == literal;
}

int matchchar(char* text) {
  return *text == 'x';
}
```

[Charles Petzold, Beautiful code, §8]

[Brian W. Kernighan & Rob Pike, The practice of programming, §9.6]

# Summary

- A practicing programmer's arsenal holds not only general-purpose languages like C++, but also programmable shells, scripting languages, and lots of application-specific languages.
- There are countless opportunities to create domain-specific languages for specialized applications. And designing and implementing such a language can be a lot of fun.
- Torben and Julia regularly offer a graduate course on domain-specific languages. This may be of interest for you!
- Possible topics for a master's thesis: self-testing and test automation.
- Think big!