BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

The Eternal Battle Against Redundancies

A redundant history of Programming Part I: Early Concepts

Christoph Knabe @ Scala User Group Berlin-Brandenburg, 15.05.2013

Contents

- My Biography & Introduction
- Addressing
- Formula Translation
- Parameterizable Subroutines
- Control Structures
- Middle-Testing Loops
- Constants & Dimensioning
- Preprocessor Features
- Array Initialization
- User Defined Datatypes
- Type Inference

Biography & Introduction

1971 learned programming on a 15 years old Zuse 22 at school.
1972... studied Computer Science in Bonn
1981... worked as Software Developer at PSI, Berlin
1990... Professor of Software Engineering at Beuth University, Berlin

Interests: Web Development, Scala, Redundancy elimination
Programmed intensively in 14 languages:
Freiburger Code, ALGOL 60, Basic, FORTRAN IV, PL/1, LISP F4, COBOL 74, Pascal, Unix-C, C++, Ada '83, Java, AspectJ, Scala

Redundancy: Multiple Description of same feature
Enemy of maintainability

Thesis: Programming Languages evolved driven by the necessity to avoid redundancies in source code.

The Zuse computers

1938 **Z1**: mechanical, programmable, binary floating point, modules: processor, storage, program decoder, I/O.

1941 **Z3**: elektromechanical, 24bit, working successor, Turing-complete

Image: Konrad Zuse with reconstructed Z3

1949 **Z4**: First commercially used computer (leased)

1955 **Z22**: Electronic Valves, 38-bit words, 16 kernel registers, 8192 storage words on magnetical drum, analytical instruction code. Sold: 55 machines.

Battle Against Redundancy

Zuse 22 Freiburger Code (1958): Absolute Adresses

Summation of Numbers from 10 to 1 (decrementing)

Adress	Instruction	Explanation
	`T2048T`	**T**ransport the following from punched tape to words 2048 ff.
2048	`10'`	**i**: Initial value for i is n, here the integer number 10.
2049	`0'`	**sum**: Initial value is the integer number 0.
2050	`B2049`	**B**ring the sum into the accu(mulator).
2051	`A2048`	**A**dd i to the accu.
2052	`U2049`	Store (**U**mspeichern) accu to sum.
2053	`B2048`	**B**ring i into the accu.
2054	`SC1`	**S**ubtract the **C**onstant value 1 from the accu.
2055	`U2048`	Store (**U**mspeichern) accu to i.
2056	`PPE2050`	If accu **P**ositive **E**xecute from (go to) 2050.
2057	`B2049`	**B**ring sum into the accu.
2058	`D`	Print (**D**rucke) accu.
2059	`Z0`	Stop
	`E2050E`	**E**xecute now from 2050

Redundante Adressen: 2048, 2049, 2050

Relative and Symbolic Addressing

Avoiding redundant absolute Addresses. Give it a name!

Z22 absolute	Z22 relative, base reg. 12	Z23 symbolic	Explanation
`T2048T`	`T2048T...`	`T2048T`	**T**ransport the following to words 2048 ff.
`10'`	`10'`	`(I) 10'`	**i**: Initial value for i is n, here integer 10.
`0'`	`0'`	`(SUM) 0'`	**sum**: Initial value is the integer number 0.
`B2049`	`B1A12`	`(BEGIN) B(SUM)`	**B**ring the sum into the accu(mulator).
`A2048`	`A0A12`	`A(I)`	**A**dd i to the accu.
`U2049`	`U1A12`	`U(SUM)`	Store (**U**mspeichern) accu to sum.
`B2048`	`B0A12`	`B(I)`	**B**ring i into the accu.
`SC1`	`SC1`	`SC1`	**S**ubtract the **C**onstant value 1 from the accu.
`U2048`	`U0A12`	`U(I)`	Store (**U**mspeichern) accu to i.
`PPE2050`	`PPE2A12`	`PPE(BEGIN)`	If accu **P**ositive **E**xecute from (go to) beginning.
`B2049`	`B1A12`	`B(SUM)`	**B**ring sum into the accu.
`D`	`D`	`D`	Print (**D**rucke) accu.
`Z0`	`Z0`	`Z0`	Stop
`E2050E`	`E2A12E`	`E(BEGIN)E`	**E**xecute now from (go to) beginning.

Formula Translation

Computing initially dominated by science & technics.

Assembler: Simple addition needs 3 instructions (bring, add, store).

Formula (a+b)*(a-b) would need about 7 instructions.

FORTRAN introduced **For**mula **Tran**slation in 1956.

Formulae had variable **identifiers** up to 6 characters, **literals**, **operator** signs, operator **priorities**, and **parentheses**.

No possibility to define own operators.

Subroutines

Instruction sequence needed several times ⇒ On Zuse 22 jump there by F (ruF=call) instruction from different program locations.

Besides jumping F loaded a „jump back“ instruction into register 5 ⇒ convention: Copy this to end of subroutine. Assures jump back to calling location.

For parameterized processing reserve words before the subroutine for arguments and result. Caller must store there before call and retrieve result afterwards.

Call a Zuse 23 subroutine for summing up 1..20:

```
BC20    U(N)    F(SUMUP)    B(SUM)    D
```

Subroutine in Zuse 23 Assembler

Summation of Numbers from N to 1 (decrementing)

Address	Instruction	Explanation
	`T2048T`	**T**ransport the following from punched tape to words 2048 ff.
`(N)`	`999999'`	Space for argument **n**, lateron decremented as **i**.
`(SUM)`	`999999'`	Space for result **sum**. Also used as intermediate result.
`(SUMUP)`	`B5`	Entry point: Bring „jump back“ instruction into accu.
	`UN(BACK)`	Store (**U**mspeichern) accu to „jump back“ word. Nullify accu.
	`U(SUM)`	Store accu, which is 0, to **sum**.
`(LOOP)`	`B(SUM)`	**B**ring the sum into the accu(mulator).
	`A(N)`	**A**dd i to the accu.
	`U(SUM)`	Store (**U**mspeichern) accu to sum.
	`B(N)`	**B**ring i into the accu.
	`SC1`	**S**ubtract the **C**onstant value 1 from the accu.
	`U(N)`	Store (**U**mspeichern) accu to i.
	`PPE(LOOP)`	If accu **P**ositive **E**xecute from (go to) 2050.
`(BACK)`	`Z0`	Space for „jump back“ instruction

Subroutine as Programming Construct

Convention on Zuse 22/23:

Space for arguments, result value

call instruction, must save „jump back“ instruction

Automated by the concept of subroutine call with argument list and return value in ALGOL and FORTRAN II in 1958.

```
C       COMPUTES SUM FROM 1 TO N IN FORTRAN II
        FUNCTION ISUMUP(N)
        ISUMUP = 0
        DO 99 I = 1, N
   99   ISUMUP = ISUMUP + I
        RETURN
```

- Implicit types: **I-N** means Integer. No Recursion.

+ Separate compilation, **COMMON** storage blocks

Control Structures

Conditional jumps as PPE of Zuse 22
enabled arbitrary algorithms.
Conditional forward jumps for branching,
Conditional backward jumps for repetition.

FORTRAN I (1956) introduced
Arithmetic IF:

```
IF (expression)  negativeLabel, zeroLabel, positiveLabel
```

Jumps to one of the labels, depending on the value of the expression. Labels were numeric ☹.

Counting Loop:

```
      DO label var = initExpr, limitExpr, IncrExpr
      statements
label lastStatement
```

ALGOL 60 Control Structures

ALGOL 60 (Algorithmic Language) pioneered:

* Definition by formal grammar (Backus-Naur-Form)
* Declaration Principle, explicit types
* Block principle (local variables)
* Recursion
* Modern looking control structures:

Multi-branch cascadable IF:

```
IF condition THEN expr ELSE expr
```

Universal loop with control variable:

```
FOR I := 1 STEP 2 UNTIL 99 DO PRINT(i)
```

prints odd numbers between 0 and 100.

```
FOR I := 1, I*2 WHILE I<2000 DO PRINT(I)
```

prints the powers of **2** from **1** to **1024**.

Pascal Structured Programming Control Structures

Pascal (1970) pioneered :

* User-defined data types (RECORDs)
* Structured Programming (avoid jumps)

Multi-branch cascadable IF like in ALGOL:

IF *condition* **THEN** *expr* **ELSE** *expr*

Pre-testing loop:

WHILE *condition* **DO** *statement*

Post-testing loop:

REPEAT *statements* **UNTIL** *condition*

Counting loop:

FOR *initialValue* **TO** *finalValue* **DO** *statement*

Nevertheless contained GOTO, as non-local termination could not be done otherwise.

Pascal Loop Redundancy Problem

Processing an unknown number of elements.

```
{Get positive numbers and print their sum}
VAR
   x: integer;
   sum: integer := 0;
BEGIN
   getNumber(x); {Sets x by a VAR parameter!}
   WHILE x>0 DO BEGIN
      sum := sum + x;
      getNumber(x);
   END;
   writeln;
   writeln('The sum is ', sum, '.');
```

Problem: `getNumber` has to be called redundantly.

Introducing middle-testing Loop

C (1973), Modula-2 (1978), and Ada (1980) thus enabled a middle-testing loop by a terminating jump instruction:

```
/*Read positive numbers and print their sum in "C"*/
int sum=0, x;
for(;;){
    getNumber(&x); //Enables setting x by passing its address!
    if (x<=0) break;
    sum += x; //Redundancy-free incrementation!
}
printf("The sum is %d.", sum);
```

Hurrah: `getNumber` call coded only once.

Pattern: get, test, process

User-Defined Control Structures: LISP I

Enabled in LISP (1960).

Instructions and data in same format: S-Expression
`(TIMES N 7)` means
a) multiply `N` by `7`
b) a list with elements `TIMES`, `N`, and `7`.

A function, defined as FEXPR (instead of usual EXPR), did not evaluate its arguments.
Left to explicit call of `EVAL` in the function body.
So body could control frequency and order of argument evaluation.

Defining IF-Statement in MacLISP (on PDP-10)

LISP dialects introduced comfortable FEXPRs.
In MacLISP by `defun`. Defining `IF`-expression:

```
(defun IF fexpr (args)
  (let ((predicate (car args))
        (then (cadr args))
        (else (caddr args)))
    (cond
     ((eval predicate) (eval then))
     (t (eval else)))))
```

For calculating absolute value of `n` now can use `if`:

```
(if (greater n 0) n (minus n))
```

Instead of traditional multi-branch `cond`:

```
(cond
   ((greater n 0) n)
   (t (minus n))
)
```

Control Abstraction in Scala (2003)

Middle-Testing loop self-defined:

```
def loopGetTestProcess[Item]
    (getItem: => Item)
    (shouldContinue: Item=>Boolean)
    (processItem: Item=>Unit)
{
    var item = getItem
    while(shouldContinue(item)){
        processItem(item)
        item = getItem
    }
}
```

Usable free of redundancy:

```
val in: java.io.BufferedReader = …
loopGetTestProcess(in.readLine())(_!=null){
    println
}
```

Constants & Dimensioning

Storage cells modifiable
⇒ No constants in Freiburger Code.

In 1960 FORTRAN, ALGOL, and COBOL had literals, but no symbolic constants.

Dimensioning problem:
In COBOL 66 a west-german ZIP code declarable as

```
ZIP PICTURE 9(4).
```

If needed in some places, size was redundant
⇒ In 1993 modifications necessary:

```
ZIP PICTURE 9(5).
```

Dimensioning Constants in Pascal (1970)

Systematic Approach:
Symbolic Constants usable for dimensioning arrays:

```
CONST zipLength: integer := 4;
TYPE  ZipCode = PACKED ARRAY[zipLength] OF character;
VAR   clientZip: ZipCode;
BEGIN
   FOR I := 1 TO zipLength DO write(clientZip[i]);
```

User-defined data types.
Help avoid redundant structural definitions.

```
TYPE Person = RECORD
   name: Name;
   age: integer;
END;
```

Dimensioning in C (1973)

- Constants only as preprocessor macros
+ **sizeof**-operator for redundancy-free iterating

```
char clientZip[4]; …
int i =0;
for(; i<sizeof(clientZip); i++){
   putchar(clientZip[i]);
}
```

C++ (1983) enabled to declare frozen variables by **`const`**.

Beginning with Java (1995) it became normal to dimension arrays only at runtime or to use growable collections.

Preprocessor Features, C (1973)

Often preprocessor used for avoiding redundancies.
E.g. include common declaration into several compilation units.

COPY in COBOL, **#include** in C.

Used for data abstraction module in C:
* declaration of exported functions in `.h` file
* function definitions + managed variables in `.c` file

See next slide.

Stack Module in C (1973)

Header file `stack.h` for a stack of characters:

```
#ifndef stack_h
#define stack_h

#define MAX_STACK_SIZE 10
void stack_push();
char stack_pop();

#endif
```

Every client of this stack had to include `stack.h`.
Then call functions:

```
#include "stack.h"
…
stack_push('X');
```

About 1985-1995 predominant modularization technique in industry before OOP.

Stringize Operator in C/C++

operator in preprocessor macro delivers the passed argument as string.

```
#define PROMPT_READ(var){cout << #var << "? "; cin >> var;}
```

Enables redundancy-free console dialog program:

```
string name;
int age;
PROMPT_READ(name);
PROMPT_READ(age);
cout << name << " is " << age << " years old. " << endl;
```

Typical dialog:

```
name? Ulf
age? 22
Ulf is 22 years old.
```

Missing Preprocessor Features in Java

* determine name of a variable.
  Reflection not applicable to variables.
* delegate method abortion
  and accompanying action, e.g.
  `errno=FAILURE; return;`
* delegate pattern of exception handling:
  If we had C-like macros in Java we could write:
  `TRANSACTION(action1(); action2();)`
  to perform stmt sequence as database transaction
  by defining the following macro:

```
#define TRANSACTION(stmts) \
{try{stmts; commit();}catch(Exception e){rollback(e);}}
```

Array Initialization and Dimensioning

* Pascal: Redundancy-free array dimensioning
* But initialization was left redundant.

See array for 4 allowed action codes:

```
VAR actions: ARRAY[1..4] OF char = ('C', 'R', 'U', 'D');
```

Size is redundant to number of elements.

C derived array size from initialization list:

```
static char actions[] = {'C', 'R', 'U', 'D'};
```

C derived array size from initialization string as well:

```
char message[] = "Hello";
assert(sizeof(message) == 6) //including terminating '\0'
```

Type Inference & Initialization

Java: Redundant type indication in declaration:

```
final StringBuilder out = new StringBuilder();
```

Avoidable by *type inference* in Scala:

```
val out = new StringBuilder
```

But if we need a collection:

```
val students = new HashMap[Integer,String]
```

Bad style. Better to program against interface:

```
val students: Map = new HashMap[Integer,String]
```

Here Scala infers not the reference type (`Map`), but its type parameters avoiding redundancy.

Summary

- Early Programming Languages introduced extraction and parameterization of a redundant code snippet.
- Sometimes by keywords. See loops, **sizeof**
- In the better case freely nameable. See addresses, constants, subroutines, types, include files, macros, Scala loops.
- Language dissemination powered by features for redundancy elimination.

Publications

Der ewige Kampf gegen Redundanzen
Eine redundante Geschichte der Programmierung
JavaMagazin 1.2012 und 2.2012:
http://jaxenter.de/Der-ewige-Kampf-gegen-Redundanzen-4415.html
http://jaxenter.de/Eine-redundante-Geschichte-der-Programmierung%3A-Von-Parnas-bis-Scala-4497.html

The Eternal Battle Against Redundancies
A Redundant History of Programming
OVERLOAD (accu.org) Dec. 2011 und Feb. 2012:
http://accu.org/var/uploads/journals/overload106.pdf
http://accu.org/var/uploads/journals/overload107.pdf

Topics for Part II

- Information Hiding
- Genericity
- Exception Handling
- Object-Oriented Programming
- Aspect-Oriented Programming
- Functional Programming
- Domain Specific Languages
- Database Normalization

And how all can help to avoid redundancies.

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Thank You