

## ADT BoolFun2

**Domain:** the set of all 16 boolean functions of two variables

### Operations:

and (f1,f2 : boolfun) : boolfun

pre: f1 and f2 are boolean functions

post: 'and' is the boolean function r such that for all x, y :  $r(x,y) = f1(x,y) \wedge f2(x,y)$ .

or(f1,f2 : boolfun) : boolfun

pre : f1 and f2 are boolean functions

post : 'or' is the boolean function r such that for all x, y :  $r(x,y) = f1(x,y) \vee f2(x,y)$ .

not (f : boolfun) : boolfun

pre: f is a boolean function

post: 'not' is a boolean function r such that for all x,y :  $r(x,y) = \neg f(x,y)$ .

false() : boolfun

pre: none

post: 'false' is boolean function r such that for all x,y :  $r(x,y) = \text{false}$

true() : boolfun

pre: none

post: 'true' is boolean function r such that for all x,y :  $r(x,y) = \text{true}$

## ADT LetterString

**Domain:** all possible strings of the letters 'A', ..., 'Z', 'a', ..., 'z' and ' '. There is a linear relationship between the letters in each string and the strings are of bounded length.

### Operations:

length (s : letterstring) : integer

pre: none

post: 'length' is the number of characters in the string

leftletter (s: letterstring) : letter

pre: length (s-pre) > 0

post: leftletter is the leftmost letter of s-pre and s-post is s-pre without the first letter

append (l: letter; s: letterstring) : letterstring

pre: numbers of letters in s-pre < bound

post: s-post is s-pre with the letter l added at the righthand end

empty (s : letterstring) : boolean

pre: none

post: 'empty' is true if s has 0 characters otherwise it is false

+ full, reverse, concatenate, .....

## Lessons from recent history

The first high-level languages (FORTRAN, Algol, APL) began to simplify the coding of numerical algorithms. They also inspired new applications such as commercial tasks, compiler design and artificial intelligence. These, in turn, provoked the need for richer data types.

COBOL (1960) introduced records. For example,

{sample record declaration and use in Pascal}

```
type rtype = record
    no:      1000..9999;
    name: packed array[1..20] of char;
    price: real
end;
```

```
var r : rtype;
    x : real;
```

```
    r.price := x;
```

COBOL allowed records to occur as fields within records producing a hierarchical structure.

A compiler translates a high-level language program into machine code. The process involves several phases: lexical analysis, parsing of the expressions and statements, code generation. These operations require more elaborate data structures than arrays and records.

The UNIX parser generator *bison* involves constructions of a combinatorial graph to represent a finite state machine and trees to represent algebraic expressions. The theory of formal languages explains how to connect graphs and trees with the parsing process. Initially the parsing process was not well understood and the syntax of FORTRAN is ill-conceived and the compiler hard and messy to program.

Moral: the right data structures can greatly simplify a programming task.

Early applications in AI such as chess playing and natural language translation made huge demands on data representation, storage and operations.

The need to restructure data during run-time led to dynamic data types and garbage collection. LISP was the first language to contain these features and others suited for symbolic list processing.

Later languages in the 1960s such as Algol68, Pascal included facilities for user-defined types and pointer variables.

