Appendices

Appendix A
Syntax of the example programs

The example functional programs in this textbook are generally written in a Miranda-based notation. These examples obey the BNF-style context-free syntax rules given in this appendix.

Conventions used in the syntax specification below

[non-term]	means that the presence of non-term is optional
[non-term]+	means that non-term occurs at least once
{non-term}	means that non-term can occur zero or more times
terminals	are enclosed by single quotes
symbols	are printed in *italic*

Script

script = {declaration} ;
declaration = function_decl | type_decl ;

Definition

function_decl = def_lhs def_rhs ;
def_lhs = pattern | function_form ;
pattern = formal | *constructor* [formal]+
  | '(' pattern ')' [formal]+
  | pattern '$'*constructor* pattern ;
formal = *variable* | *constructor* | *literal*
  | '(' pattern ')' | tuple_formal | list_formal ;
tuple_formal = '(' ')' | '(' pattern [',' pattern]+')' ;
list_formal = '[' ']' | '(' pattern ':' pattern ')'
  | '[' pattern {',' pattern} ']' ;
function_form = *variable* [formal]+
  | pattern '$'*variable* pattern
  | '(' function_form ')' {formal} ;
def_rhs = {def_case} def_lastcase [where_def] *terminator* ;
def_case = '=' expression ',' guard *terminator* ;
expression = prefix_operator [expression]
  | infix_operator
  | [simple_expr]+
  | expression infix_operator expression
  | expression '$'*variable* expression
  | expression '$'*constructor* expression ;
simple_expr = *variable* | *constructor* | *literal*
  | '(' expression ')'
  | '(' infix_operator expression ')'
  | '(' expression infix_operator ')' | tuple | list ;

```
tuple            =  '(' ')'          |  '(' expression [',' expression]+')' ;
list             =  '[' ']'          |  '[' expression {',' expression} ']'
                 |  dotdot_expr      |  zf_expr ;
dotdot_expr      =  '[' expression [',' expression] '..' [expression] ']' ;
zf_expr          =  '[' expression  '|' qualifier {';' qualifier} ']'
                 |  '[' expression '//' qualifier {';' qualifier} ']' ;
qualifier        =  expression       |  generator ;
generator        =  pattern {',' pattern} '<-' expression
                 |  pattern '<-' expression ',' expression '..' ;
prefix_operator  =  '~'  |  '-'  |  '#' ;
infix_operator   =  '++' |  ':'  |  '\/' |  '&'  |  '>'  |  '>='  |  '~='
                 |  '<=' |  '<'  |  '='  |  '+'  |  '*'  |  '/'   |  'div'
                 |  '^'  |  '.'  |  '!'  |  '-'  |  '--' |  'mod' ;
guard            =  [ 'if' ] expression ;
def_lastcase     =  '=' expression ',' guard        |  '=' expression ',' 'otherwise'
                 |  '=' expression ;
where_def        =  'where' [function_decl]+ ;
```

Type declaration and specification

```
type_decl        =  abstype_def  |  typesyn_def  |  type_def     |  type_spec ;
abstype_def      =  'abstype' type_form {',' type_form} 'with' signature terminator ;
type_form        =  type_name {type_variable}
                 |  type_variable '$'type_variable type_variable ;
signature        =  [type_spec]+ ;
type_spec        =  type_variable {',' type_variable} '::' type terminator
                 |  type_form {',' type_form} '::' 'type' terminator ;
type             =  simple_type      |  type_name [simple_type]+
                 |  type '->' type   |  type '$'type_variable type ;
simple_type      =  type_name        |  type_variable  |  '(' type ')'
                 |  tuple_type       |  list_type ;
tuple_type       =  '(' ')'          |  '(' type [',' type ]+')' ;
list_type        =  '[' type ']' ;
typesyn_def      =  type_form '==' type terminator ;
type_def         =  type_form '::=' type_construct {',' type_construct} terminator ;
type_construct   =  '(' type_construct ')' {simple_type}
                 |  constructor {simple_type}      |  type '$'constructor type ;
```

Symbols

```
variable         =  IDENT ;
type_name        =  IDENT ;
type_variable    =  TYPEVAR ;
constructor      =  CONS ;
literal          =  NUMERAL          |  CHAR           |  STRING ;
terminator       =  ';'              |  OFFSIDE ;
```

Lexical classes used in the syntax specification above

IDENT	identifiers starting with a lower case letter, e.g. f, f', longer_name
CONS	identifiers starting with an upper case letter, e.g. Cons, Tree
TYPEVAR	one or more asterisk, e.g. *, **, ***
NUMERAL	numeral denotation, e.g. 1.0, 2, 3.3e7
CHAR	character denotation, e.g. 'c', '\n'
STRING	string denotation, e.g. "hello world\n"
OFFSIDE	not a proper terminal: represents an application of the offside rule