

# The PREV programming language

(academic year 2016/17)

Boštjan Slivnik

## 1 Lexical structure

Programs in the PREV programming language are written in ASCII character set (no additional characters denoting post-alveolar consonants are allowed).

Programs in the PREV programming language consist of the following lexical elements:

- *Literals:*
  - literals of type void: **none**
  - literals of type bool: **true false**
  - literals of type char:  
An character with the character code in decimal range 32...126 (from space to ~) enclosed in single quotes (').
  - literals of type int:  
A nonempty finite string of digits (0...9) optionally preceded by a sign (+ or -).
  - literals of pointer types: **null**
- *Symbols:*  
`! | ^ & == != <= >= < > + - * / % $ @ = . , : ; [ ] ( ) { }`
- *Keywords:*  
`arr bool char del do else end fun if int new ptr rec then typ var void where while`
- *Identifiers:*  
A nonempty finite string of letters (A...Z and a...z), digits (0...9), and underscores (\_) that (a) starts with either a letter or an underscore and (b) is not a keyword.
- *Comments:*  
A string of characters starting with a hash (#) and extending to the end of line.
- *White space:*  
Space, horizontal tab (HT), line feed (LF) and carriage return (CR). Line feed alone denotes the end of line within a source file.

Lexical elements should be recognised from left to right using the longest match approach.

## 2 Syntax structure

The concrete syntax of the PREV programming language is defined by context free grammar with the start symbol *expr* and productions

(literal)	$expr \rightarrow literal$
(unary expression)	$expr \rightarrow unop\ expr$
(binary expression)	$expr \rightarrow expr\ binop\ expr$
(variable access)	$expr \rightarrow identifier$
(function call)	$expr \rightarrow identifier([expr\{, expr\}])$
(element access)	$expr \rightarrow expr[expr]$
(component access)	$expr \rightarrow expr.identifier$
(type cast)	$expr \rightarrow [type]\ expr$
(memory allocation)	$expr \rightarrow new\ type$
(memory deallocation)	$expr \rightarrow del\ expr$
(compound expression)	$expr \rightarrow \{ stmt\{; stmt\} : expr\ [where\ decl\{; decl\}] \}$
(enclosed expression)	$expr \rightarrow (expr)$
(atomic type)	$type \rightarrow void\  \ bool\  \ char\  \ int$
(array type)	$type \rightarrow arr\ [expr]\ type$
(record type)	$type \rightarrow rec\ (identifier : type\{, identifier : type\})$
(pointer type)	$type \rightarrow ptr\ type$
(named type)	$type \rightarrow identifier$
(expression)	$stmt \rightarrow expr$
(assignment)	$stmt \rightarrow expr = expr$
(conditional)	$stmt \rightarrow if\ expr\ then\ stmt\{, stmt\}\ [else\ stmt\{, stmt\}]\ end$
(loop)	$stmt \rightarrow while\ expr\ do\ stmt\{, stmt\}\ end$
(type declaration)	$decl \rightarrow typ\ identifier : type$
(variable declaration)	$decl \rightarrow var\ identifier : type$
(function declaration)	$decl \rightarrow fun\ identifier([identifier : type\{, identifier : type\}]) : type\ [=expr]$

where *literal* denotes any literal, *unop* denotes an unary operator (any of `!`, `+`, `-`, `$` and `@`) and *binop* denotes a binary operator (any of `|`, `^`, `&`, `==`, `!=`, `<=`, `>=`, `<`, `>`, `+`, `-`, `*`, `/` and `%`).

Relational operators are non-associative, all other binary operators are left associative.

The precedence of operators is as follows:

<code>  ^</code>	THE LOWEST PRECEDENCE
<code>&amp;</code>	
<code>== != &lt;= &gt;= &lt; &gt;</code>	
<code>+ -</code>	(binary <code>+</code> and <code>-</code> )
<code>* / %</code>	
<code>! + - \$ @ new del type-cast</code>	(unary <code>+</code> and <code>-</code> )
<code>array-access component-access</code>	THE HIGHEST PRECEDENCE