The PREV'19 programming language

(academic year 2018/19)

Boštjan Slivnik

1 Lexical structure

Programs in the PREV'19 programming language are written in ASCII character set (no additional characters denoting post-alveolar consonants are allowed).

Programs in the PREV'19 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 a character code in decimal range $\{32 \ldots 126\}$ (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`
  - string literals:
    A possibly empty finite string of characters with character codes in decimal range $\{32 \ldots 126\} \setminus \{34\}$ (from space to `~` but excluding double quote) enclosed in double quotes (`"`).
- *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 or a literal.
- *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. Horizontal tab is 8 spaces wide.

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 *source* and productions

(*program*) $\quad source \longrightarrow decl\,\{decl\}$

(*type declaration*) $\quad decl \longrightarrow$ `typ` *identifier* `:` *type* `;`
(*variable declaration*) $\quad decl \longrightarrow$ `var` *identifier* `:` *type* `;`
(*function declaration*) $\quad decl \longrightarrow$ `fun` *identifier* `(`$[$*identifier* `:` *type* $\{$ `,` *identifier* `:` *type*$\}$`]) :` *type* $[$`=`*expr*$]$ `;`

(*atomic type*) $\quad type \longrightarrow$ `void` | `bool` | `char` | `int`
(*array type*) $\quad type \longrightarrow$ `arr [`*expr*`]` *type*
(*record type*) $\quad type \longrightarrow$ `rec (`*identifier* `:` *type* $\{$ `,` *identifier* `:` *type*$\}$`)`
(*pointer type*) $\quad type \longrightarrow$ `ptr` *type*
(*named type*) $\quad type \longrightarrow$ *identifier*
(*enclosed type*) $\quad type \longrightarrow$ `(`*type*`)`

(*literal*) $\quad expr \longrightarrow literal$
(*unary expression*) $\quad expr \longrightarrow unop\ expr$
(*binary expression*) $\quad expr \longrightarrow expr\ binop\ expr$
(*variable access*) $\quad expr \longrightarrow$ *identifier*
(*function call*) $\quad expr \longrightarrow$ *identifier* `(`$[$*expr* $\{$ `,` *expr*$\}$`]`
(*element access*) $\quad expr \longrightarrow expr$ `[`*expr*`]`
(*component access*) $\quad expr \longrightarrow expr$ `.` *identifier*
(*memory allocation*) $\quad expr \longrightarrow$ `new (`*type*`)`
(*memory deallocation*) $\quad expr \longrightarrow$ `del (`*expr*`)`
(*compound expression*) $\quad expr \longrightarrow$ `{` *stmt* $\{$*stmt*$\}$ `:` *expr* $[$`where` *decl* $\{$*decl*$\}]$`}`
(*typecast*) $\quad expr \longrightarrow$ `(`*expr* `:` *type*`)`
(*enclosed expression*) $\quad expr \longrightarrow$ `(`*expr*`)`

(*expression*) $\quad stmt \longrightarrow expr$ `;`
(*assignment*) $\quad stmt \longrightarrow expr$ `=` *expr* `;`
(*conditional*) $\quad stmt \longrightarrow$ `if` *expr* `then` *stmt* $\{$*stmt*$\}$ $[$`else` *stmt* $\{$*stmt*$\}]$ `end;`
(*loop*) $\quad stmt \longrightarrow$ `while` *expr* `do` *stmt* $\{$*stmt*$\}$ `end;`

where *literal* denotes any literal, *unop* denotes an unary operator (any of `!`, `+`, `-`, `$` and `@`) and *binop* denotes a binary operator (any of `|`, `^`, `&`, `==`, `!=`, `<=`, `>=`, `<`, `>`, `+`, `-`, `*`, `/` and `%`). In the grammar above, braces typeset as {} enclose sentential forms that can repeated zero or more times, brackets typeset as [] enclose sentential forms that can be present or not while braces and brackets typeset as `{}` and `[]` denote characters that are a part of the program text.

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

The precedence of operators is as follows:

`| ^`	THE LOWEST PRECEDENCE
`&`	
`== != <= >= < >`	
`+ -`	(binary `+` and `-`)
`* / %`	
`! + - $ @ new del` *type-cast*	(unary `+` and `-`)
array-access component-access	THE HIGHEST PRECEDENCE