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

3 Semantic structure

3.1 Namespaces

There are two kinds of a namespaces:

1. Names of types, functions, variables and parameters belong to one single global namespace.
2. Names of record components belong to record-specific namespaces, i.e., each record defines its own namespace containing names of its components.

3.2 Scopes

A new scope is created in two ways:

1. Compound expression

   `{` *stmt* {`;`*stmt*} `:` *expr* [`where` *decl* {`;`*decl*}] `}`

   creates a new scope. The scope starts right after `{` and ends just before `}`.

2. Function declaration

   `fun` *identifier*`(`[*identifier*`:`*type* {`,`*identifier*`:`*type*}]`)`:*type* [`=`*expr*]

   creates a new scope. The name of a function, the types of parameters and the type of a result belong to the outer scope while the names of parameters and the expression denoting the function body belong to the scope created by the function declaration.

All names declared within a given scope are visible in the entire scope unless hidden by a declaration in the nested scope.

3.3 Constant subexpressions

Let $\mathrm{I} = \{(-2^{63}) \ldots (2^{63}-1)\}$. Semantic function

$$[\![\cdot]\!]_{\mathrm{VAL}} \colon \mathcal{P} \to \mathrm{I}$$

maps phrases of PREV to the integer values they denote. It is defined by the following rules:

$$\frac{\mathit{lexeme}(\mathrm{INTEGER}) \in \mathrm{I}}{[\![\mathrm{INTEGER}]\!]_{\mathrm{VAL}} = \mathit{lexeme}(\mathrm{INTEGER})} \qquad \frac{[\![\mathit{expr}]\!]_{\mathrm{VAL}} = n}{[\![\texttt{(}\mathit{expr}\texttt{)}]\!]_{\mathrm{VAL}} = n}$$

$$\frac{[\![\mathit{expr}]\!]_{\mathrm{VAL}} = n \quad \mathrm{op} \in \{\texttt{+},\texttt{-}\}}{[\![\mathrm{op}\ \mathit{expr}]\!]_{\mathrm{VAL}} = \mathit{op}\ n} \qquad \frac{[\![\mathit{expr}_1]\!]_{\mathrm{VAL}} = n_1 \quad [\![\mathit{expr}_2]\!]_{\mathrm{VAL}} = n_2 \quad \mathrm{op} \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\}}{[\![\mathit{expr}_1\ \mathrm{op}\ \mathit{expr}_2]\!]_{\mathrm{VAL}} = n_1\ \mathit{op}\ n_2}$$

In all other cases the value of $[\![\cdot]\!]_{\mathrm{VAL}}$ is undefined (denoted by $\perp$).