The PREV Language Specification

Boštjan Slivnik

August 25, 2016

1 Lexical structure

The programs in the PREV programming language are written is 7-bit ASCII character set (all other characters are invalid). A single character LF denotes the end of line regardless of the text file format.

The (groups of) symbols of the PREV programming language are:

- **Symbols:**

  `+ & = : , } ] ) . / == >= > <= < @ % * != ! { [ ( | - ^`

- **Constants:**
  - *Boolean constants:* `true` `false`
  - *Integer constants:* An integer constant is a sequence of decimal digits optionally prefixed by a sign, i.e., "+" or "−", denoting a 64-bit signed integer, i.e., from the interval $[-2^{63}, 2^{63}-1]$.
  - *Character constants:* A character constant consists of a single character name within single quotes. A character name is either a character with an ASCII code from the interval $[32, 126]$ (but not a backslash, a single or a double quote) or an escape sequence. An escape sequence starts with a backslash character followed by a backslash (denoting a backslash), a single quote (denoting a single quote), a double quote (denoting a double quote), "`t`" (denoting TAB), or "`n`" (denoting LF).
  - *String constants:* A string constant is a possibly empty finite sequence of character names within double quotes. A character name is either a character with an ASCII code from the interval $[32, 126]$ (but not a backslash, a single or a double quote) or an escape sequence. An escape sequence starts with a backslash character followed by a backslash (denoting a backslash), a single quote (denoting a single quote), a double quote (denoting a double quote), "`t`" (denoting TAB), or "`n`" (denoting LF).
  - *Pointer constant:* `null`
  - *Void constant:* `none`

- **Type names:**

  `boolean integer char string void`

- **Keywords:**

  `arr else end for fun if then ptr rec typ var where while`

- **Identifiers:** An identifier is a nonempty finite sequence of letters, digits and underscores that starts with a letter or an underscore.

Additionaly, the source might include the following:

- **White space:** Characters CR, LF, TAB, or space.
- **Comments:** A comment is a sequence of character that starts with an octothorpe character, i.e., "`#`", and ends with the LF character (regardless of the text file format).

To break the source file into individual symbols, the first-match-longest-match rule must be used.

2 Syntactic structure

The concrete syntax of the PREV programming language is defined by an LR(1) grammar. Nonterminal and terminal symbols are written in italic and typewritter fonts, respectivelly. Terminal symbols `IDENTIFIER`, `INTEGER`, `BOOLEAN`, `CHAR` and `STRING` denote (all) identifiers, integer constants, boolean constants, character constants and string constants, respectivelly. The start symbol of the grammar is *Program*. The LR(1) grammar contains the following productions:

Program ⟶ *Expression*

Expression ⟶ *AssignmentExpression*
Expression ⟶ *Expression* `where` *Declarations* `end`

Expressions ⟶ *Expression*
Expressions ⟶ *Expressions* `,` *Expression*

AssignmentExpression ⟶ *DisjunctiveExpression*
AssignmentExpression ⟶ *DisjunctiveExpression* `=` *DisjunctiveExpression*

DisjunctiveExpression ⟶ *ConjunctiveExpression*
DisjunctiveExpression ⟶ *DisjunctiveExpression* `|` *ConjunctiveExpression*

ConjunctiveExpression ⟶ *RelationalExpression*
ConjunctiveExpression ⟶ *ConjunctiveExpression* `&` *RelationalExpression*

RelationalExpression ⟶ *AdditiveExpression*
RelationalExpression ⟶ *AdditiveExpression* `==` *AdditiveExpression*
RelationalExpression ⟶ *AdditiveExpression* `!=` *AdditiveExpression*
RelationalExpression ⟶ *AdditiveExpression* `<` *AdditiveExpression*
RelationalExpression ⟶ *AdditiveExpression* `>` *AdditiveExpression*
RelationalExpression ⟶ *AdditiveExpression* `<=` *AdditiveExpression*
RelationalExpression ⟶ *AdditiveExpression* `>=` *AdditiveExpression*

AdditiveExpression ⟶ *AdditiveExpression* `+` *MultiplicativeExpression*
AdditiveExpression ⟶ *AdditiveExpression* `-` *MultiplicativeExpression*
AdditiveExpression ⟶ *MultiplicativeExpression*

MultiplicativeExpression ⟶ *MultiplicativeExpression* `*` *PrefixExpression*
MultiplicativeExpression ⟶ *MultiplicativeExpression* `/` *PrefixExpression*
MultiplicativeExpression ⟶ *MultiplicativeExpression* `%` *PrefixExpression*
MultiplicativeExpression ⟶ *PrefixExpression*

PrefixExpression ⟶ *PostfixExpression*
PrefixExpression ⟶ `+` *PrefixExpression*
PrefixExpression ⟶ `-` *PrefixExpression*
PrefixExpression ⟶ `!` *PrefixExpression*
PrefixExpression ⟶ `@` *PrefixExpression*
PrefixExpression ⟶ `[` *Type* `]` *PrefixExpression*

PostfixExpression ⟶ *AtomicExpression*
PostfixExpression ⟶ *PostfixExpression* `[` *Expression* `]`
PostfixExpression ⟶ *PostfixExpression* `.` `IDENTIFIER`
PostfixExpression ⟶ *PostfixExpression* `^`

AtomicExpression ⟶ `INTEGER`

AtomicExpression ⟶ `BOOLEAN`

AtomicExpression ⟶ `CHAR`

AtomicExpression ⟶ `STRING`

AtomicExpression ⟶ `null`

AtomicExpression ⟶ `none`

AtomicExpression ⟶ `IDENTIFIER` *ArgumentsOpt*

AtomicExpression ⟶ `(` *Expressions* `)`

AtomicExpression ⟶ `if` *Expression* `then` *Expression* `else` *Expression* `end`

AtomicExpression ⟶ `for` `IDENTIFIER` = *Expression* `,` *Expression* `:` *Expression* `end`

AtomicExpression ⟶ `while` *Expression* `:` *Expression* `end`

ArgumentsOpt ⟶ `(`

ArgumentsOpt ⟶ erm

ArgumentsOpt ⟶ `(` *Expressions* `)`

Declarations ⟶ *Declaration*

Declarations ⟶ *Declarations Declaration*

Declaration ⟶ *TypeDeclaration*

Declaration ⟶ *FunctionDeclaration*

Declaration ⟶ *VariableDeclaration*

TypeDeclaration ⟶ `typ` `IDENTIFIER` `:` *Type*

FunctionDeclaration ⟶ `fun` `IDENTIFIER` `(` *ParametersOpt* `)` `:` *Type FunctionBodyOpt*

ParametersOpt ⟶ erm

ParametersOpt ⟶ *Parameters*

Parameters ⟶ *Parameter*

Parameters ⟶ *Parameters* `,` *Parameter*

Parameter ⟶ `IDENTIFIER` `:` *Type*

FunctionBodyOpt ⟶ erm

FunctionBodyOpt ⟶ = *Expression*

VariableDeclaration ⟶ `var` `IDENTIFIER` `:` *Type*

Type ⟶ `integer`

Type ⟶ `boolean`

Type ⟶ `char`

Type ⟶ `string`

Type ⟶ `void`

Type ⟶ `arr` `[` *Expression* `]` *Type*

Type ⟶ `rec` `{` *Components* `}`

Type ⟶ `ptr` *Type*

Type ⟶ `IDENTIFIER`

Components ⟶ *Component*

Components ⟶ *Components* `,` *Component*

Component ⟶ `IDENTIFIER` `:` *Type*

Note that the LR(1) grammar generates certain sentential forms which are prohibited by semantics.

3 Semantic rules

3.1 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 Scope

1. `where`-expression

$$\textit{expr}\ \texttt{where}\ \textit{decl}_1\ \textit{decl}_2\ \ldots\ \textit{decl}_n\ \texttt{end}$$

   creates a new scope:

   - subexpression *expr* and declarations $\textit{decl}_i$, for $i = 1, 2, \ldots, n$, belong to the new scope;
   - names declared by declarations $\textit{decl}_i$, for $i = 1, 2, \ldots, n$, are visible in the scope (unless they are redeclared within inner scopes).

2. Function declaration

$$\texttt{fun ID (ID}_1\texttt{:}\textit{type}_1\texttt{, ID}_2\texttt{:}\textit{type}_2\texttt{, } \ldots\texttt{, ID}_n\texttt{:}\textit{type}_n\texttt{) : } \textit{type}\ (= \textit{expr})_{\text{opt}}$$

   creates a new scope:

   - Function name `ID`, function type *type* and parameter types $\textit{type}_i$, for $i = 1, 2, \ldots, n$, do not belong to the new scope.
   - Parameter names $\texttt{ID}_i$, for $i = 1, 2, \ldots, n$, and function body *expr* (if specified) belong to the new scope and are visible in the entire new scope.

3.3 Constant integer expressions

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

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

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

$$\frac{\textit{lexeme}(\texttt{INTEGER}) \in \mathrm{I}}{[\![\texttt{INTEGER}]\!]_{\text{VAL}} = \textit{lexeme}(\texttt{INTEGER})}$$

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

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

3.4 Type system

A set

$$\begin{aligned}
\mathrm{T} = \{&\textit{boolean}, \textit{integer}, \textit{char}, \textit{string}, \textit{void}\} && \text{(atomic types)}\\
&\cup \{\textit{arr}(n \times \tau) \mid n \in \mathrm{I} \wedge \tau \in \mathrm{T}\} && \text{(arrays)}\\
&\cup \{\textit{rec}(\tau_1, \tau_2, \ldots, \tau_n) \mid n > 0 \wedge \tau_1, \tau_2, \ldots, \tau_n \in \mathrm{T}\} && \text{(records)}\\
&\cup \{\textit{ptr}(\tau) \mid \tau \in \mathrm{T}\} && \text{(pointers)}\\
&\cup \{(\tau_1, \tau_2, \ldots, \tau_n) \to \tau \mid n \geq 0 \wedge \tau_1, \tau_2, \ldots, \tau_n \in \mathrm{T}\} && \text{(functions)}
\end{aligned}$$

denotes a set of all types of PREV. Two types are equal if they share the same structure.

Semantic function $[\![.]\!]_{\text{TYP}}$

$$[\![\cdot]\!]_{\text{TYP}} : \mathcal{P} \to \mathrm{T}$$

maps phrases of PREV to types.

Let function DECL maps a type, function, variable or parameter name to its declaration according to the namespace and scope rules specified above and let DECL_τ map a component name to its declaration within (a record) type τ. If a name is undefined in the current namespace or scope, functions DECL and DECL_τ return $\perp$. The semantic function $[\![.]\!]_{\text{TYP}}$ is defined by the rules in the following subsections. In cases when no rule can be applied, the semantic function $[\![.]\!]_{\text{TYP}}$ is left undefined (denoted by $\perp$).

3.4.1 Type expressions:

$$\frac{}{[\![\,\texttt{boolean}\,]\!]_{\mathrm{TYP}} = \mathit{boolean}} \qquad \frac{}{[\![\,\texttt{integer}\,]\!]_{\mathrm{TYP}} = \mathit{integer}}$$

$$\frac{}{[\![\,\texttt{char}\,]\!]_{\mathrm{TYP}} = \mathit{char}} \qquad \frac{}{[\![\,\texttt{string}\,]\!]_{\mathrm{TYP}} = \mathit{string}} \qquad \frac{}{[\![\,\texttt{void}\,]\!]_{\mathrm{TYP}} = \mathit{void}}$$

$$\frac{[\![\,\mathit{expr}\,]\!]_{\mathrm{VAL}} = n \quad n > 0 \quad [\![\,\mathit{type}\,]\!]_{\mathrm{TYP}} = \tau}{[\![\,\texttt{arr [}\mathit{expr}\texttt{]}\ \mathit{type}\,]\!]_{\mathrm{TYP}} = \mathit{arr}(n \times \tau)} \qquad \frac{[\![\,\mathit{type}\,]\!]_{\mathrm{TYP}} = \tau}{[\![\,\texttt{ptr}\ \mathit{type}\,]\!]_{\mathrm{TYP}} = \mathit{ptr}(\tau)}$$

$$\frac{\forall i\colon [\![\,\mathit{type}_i\,]\!]_{\mathrm{TYP}} = \tau_i}{[\![\,\texttt{rec}\ \{\ \mathit{name}_1\texttt{:}\mathit{type}_1\texttt{,}\mathit{name}_2\texttt{:}\mathit{type}_2\texttt{,}\ \ldots\texttt{,}\mathit{name}_n\texttt{:}\mathit{type}_n\ \}\,]\!]_{\mathrm{TYP}} = \mathit{rec}(\tau_1, \tau_2, \ldots, \tau_n)}$$

$$\frac{[\![\,\mathrm{DECL}(\mathit{typ\text{-}name})\,]\!]_{\mathrm{TYP}} = \tau}{[\![\,\mathit{typ\text{-}name}\,]\!]_{\mathrm{TYP}} = \tau}$$

3.4.2 Value expressions:

$$\frac{}{[\![\,\texttt{BOOLEAN}\,]\!]_{\mathrm{TYP}} = \mathit{boolean}} \qquad \frac{}{[\![\,\texttt{INTEGER}\,]\!]_{\mathrm{TYP}} = \mathit{integer}}$$

$$\frac{}{[\![\,\texttt{CHAR}\,]\!]_{\mathrm{TYP}} = \mathit{char}} \qquad \frac{}{[\![\,\texttt{STRING}\,]\!]_{\mathrm{TYP}} = \mathit{string}}$$

$$\frac{}{[\![\,\texttt{none}\,]\!]_{\mathrm{TYP}} = \mathit{void}} \qquad \frac{}{[\![\,\texttt{null}\,]\!]_{\mathrm{TYP}} = \mathit{ptr}(\mathit{void})}$$

$$\frac{[\![\,\mathit{expr}\,]\!]_{\mathrm{TYP}} = \mathit{boolean}}{[\![\ \texttt{!}\ \mathit{expr}\,]\!]_{\mathrm{TYP}} = \mathit{boolean}} \qquad \frac{[\![\,\mathit{expr}\,]\!]_{\mathrm{TYP}} = \mathit{integer} \quad \mathrm{op} \in \{\texttt{+}, \texttt{-}\}}{[\![\,\mathrm{op}\ \mathit{expr}\,]\!]_{\mathrm{TYP}} = \mathit{integer}}$$

$$\frac{[\![\,\mathit{expr}_1\,]\!]_{\mathrm{TYP}} = \mathit{integer} \quad [\![\,\mathit{expr}_2\,]\!]_{\mathrm{TYP}} = \mathit{integer} \quad \mathrm{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{[\![\,\mathit{expr}_1\ \mathrm{op}\ \mathit{expr}_2\,]\!]_{\mathrm{TYP}} = \mathit{integer}}$$

$$\frac{[\![\,\mathit{expr}_1\,]\!]_{\mathrm{TYP}} = \mathit{boolean} \quad [\![\,\mathit{expr}_2\,]\!]_{\mathrm{TYP}} = \mathit{boolean} \quad \mathrm{op} \in \{\texttt{\&}, \texttt{|}\}}{[\![\,\mathit{expr}_1\ \mathrm{op}\ \mathit{expr}_2\,]\!]_{\mathrm{TYP}} = \mathit{boolean}}$$

$$\frac{\begin{array}{c}[\![\,\mathit{expr}_1\,]\!]_{\mathrm{TYP}} = \tau_1 \quad [\![\,\mathit{expr}_2\,]\!]_{\mathrm{TYP}} = \tau_2 \quad \mathrm{op} \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\} \\ \tau_1, \tau_2 \in \{\mathit{boolean}, \mathit{integer}, \mathit{char}\} \cup \{\mathit{ptr}(\tau) \mid \tau \in \mathrm{T}\} \quad \tau_1 = \tau_2\end{array}}{[\![\,\mathit{expr}_1\ \mathrm{op}\ \mathit{expr}_2\,]\!]_{\mathrm{TYP}} = \mathit{boolean}}$$

$$\frac{\begin{array}{c}[\![\,\mathit{expr}_1\,]\!]_{\mathrm{TYP}} = \tau_1 \quad [\![\,\mathit{expr}_2\,]\!]_{\mathrm{TYP}} = \tau_2 \quad [\![\,\mathit{expr}_1\,]\!]_{\mathrm{MEM}} = \mathsf{true} \\ \tau_1, \tau_2 \in \{\mathit{boolean}, \mathit{integer}, \mathit{char}, \mathit{string}\} \cup \{\mathit{ptr}(\tau) \mid \tau \in \mathrm{T}\} \quad \tau_1 = \tau_2\end{array}}{[\![\,\mathit{expr}_1\ \texttt{=}\ \mathit{expr}_2\,]\!]_{\mathrm{TYP}} = \mathit{void}}$$

$$\frac{[\![\,\mathrm{DECL}(\mathit{var\text{-}name})\,]\!]_{\mathrm{TYP}} = \tau}{[\![\,\mathit{var\text{-}name}\,]\!]_{\mathrm{TYP}} = \tau} \qquad \frac{[\![\,\mathrm{DECL}(\mathit{par\text{-}name})\,]\!]_{\mathrm{TYP}} = \tau}{[\![\,\mathit{par\text{-}name}\,]\!]_{\mathrm{TYP}} = \tau}$$

$$\frac{\forall i\colon [\![\,\mathit{expr}_i\,]\!]_{\mathrm{TYP}} = \tau_i \quad [\![\,\mathrm{DECL}(\mathit{fun\text{-}name})\,]\!]_{\mathrm{TYP}} = (\tau_1, \tau_2, \ldots, \tau_n) \to \tau}{[\![\,\mathit{fun\text{-}name}\ \texttt{(}\mathit{expr}_1\texttt{,}\mathit{expr}_2\texttt{,}\ \ldots\texttt{,}\mathit{expr}_n\texttt{)}\,]\!]_{\mathrm{TYP}} = \tau}$$

$$\frac{\llbracket\, \mathit{expr} \,\rrbracket_{\mathrm{TYP}} = \tau \quad \llbracket\, \mathrm{DECL}_{\tau}(\mathit{comp\text{-}name}) \,\rrbracket_{\mathrm{TYP}} = \tau'}{\llbracket\, \mathit{expr}\,\texttt{.}\,\mathit{comp\text{-}name} \,\rrbracket_{\mathrm{TYP}} = \tau'}$$

$$\frac{\llbracket\, \mathit{expr}_1 \,\rrbracket_{\mathrm{TYP}} = \mathit{arr}(n \times \tau) \quad \llbracket\, \mathit{expr}_2 \,\rrbracket_{\mathrm{TYP}} = \mathit{integer}}{\llbracket\, \mathit{expr}_1\,\texttt{[}\,\mathit{expr}_2\,\texttt{]} \,\rrbracket_{\mathrm{TYP}} = \tau}$$

$$\frac{\llbracket\, \mathit{expr} \,\rrbracket_{\mathrm{TYP}} = \tau \quad \llbracket\, \mathit{expr} \,\rrbracket_{\mathrm{MEM}} = \mathsf{true}}{\llbracket\, \texttt{@}\mathit{expr} \,\rrbracket_{\mathrm{TYP}} = \mathit{ptr}(\tau)} \qquad \frac{\llbracket\, \mathit{expr} \,\rrbracket_{\mathrm{TYP}} = \mathit{ptr}(\tau)}{\llbracket\, \mathit{expr}\,\texttt{\^{}} \,\rrbracket_{\mathrm{TYP}} = \tau}$$

$$\frac{\llbracket\, \mathit{type} \,\rrbracket_{\mathrm{TYP}} = \mathit{ptr}(\tau) \quad \llbracket\, \mathit{expr} \,\rrbracket_{\mathrm{TYP}} = \mathit{ptr}(\mathit{void})}{\llbracket\, \texttt{[}\,\mathit{type}\,\texttt{]}\,\mathit{expr} \,\rrbracket_{\mathrm{TYP}} = \mathit{ptr}(\tau)}$$

$$\frac{\forall i\colon \llbracket\, \mathit{expr}_i \,\rrbracket_{\mathrm{TYP}} \neq \bot \quad \llbracket\, \mathit{expr}_n \,\rrbracket_{\mathrm{TYP}} = \tau}{\llbracket\, \texttt{(}\mathit{expr}_1\texttt{,}\,\mathit{expr}_2\texttt{,}\,\ldots\texttt{,}\,\mathit{expr}_n\texttt{)} \,\rrbracket_{\mathrm{TYP}} = \tau}$$

$$\frac{\llbracket\, \mathit{cond\text{-}expr} \,\rrbracket_{\mathrm{TYP}} = \mathit{boolean} \quad \llbracket\, \mathit{then\text{-}body} \,\rrbracket_{\mathrm{TYP}} \neq \bot \quad \llbracket\, \mathit{else\text{-}body} \,\rrbracket_{\mathrm{TYP}} \neq \bot}{\llbracket\, \texttt{if}\ \mathit{cond\text{-}expr}\ \texttt{then}\ \mathit{then\text{-}body}\ \texttt{else}\ \mathit{else\text{-}body}\ \texttt{end} \,\rrbracket_{\mathrm{TYP}} = \mathit{void}}$$

$$\frac{\llbracket\, \mathit{var\text{-}name} \,\rrbracket_{\mathrm{TYP}} = \mathit{integer} \quad \llbracket\, \mathit{lo\text{-}expr} \,\rrbracket_{\mathrm{TYP}} = \mathit{integer} \quad \llbracket\, \mathit{hi\text{-}expr} \,\rrbracket_{\mathrm{TYP}} = \mathit{integer} \quad \llbracket\, \mathit{body} \,\rrbracket_{\mathrm{TYP}} \neq \bot}{\llbracket\, \texttt{for}\ \mathit{var\text{-}name}\,\texttt{:}\,\mathit{lo\text{-}expr}\,\texttt{,}\,\mathit{hi\text{-}expr}\,\texttt{:}\,\mathit{body}\ \texttt{end} \,\rrbracket_{\mathrm{TYP}} = \mathit{void}}$$

$$\frac{\llbracket\, \mathit{cond\text{-}expr} \,\rrbracket_{\mathrm{TYP}} = \mathit{boolean} \quad \llbracket\, \mathit{body} \,\rrbracket_{\mathrm{TYP}} \neq \bot}{\llbracket\, \texttt{while}\ \mathit{cond\text{-}expr}\,\texttt{:}\,\mathit{body}\ \texttt{end} \,\rrbracket_{\mathrm{TYP}} = \mathit{void}}$$

$$\frac{\forall i\colon \llbracket\, \mathit{decl}_i \,\rrbracket_{\mathrm{TYP}} \neq \bot \quad \llbracket\, \mathit{expr} \,\rrbracket_{\mathrm{TYP}} = \tau}{\llbracket\, \mathit{expr}\ \texttt{where}\ \mathit{decl}_1\texttt{,}\,\mathit{decl}_2\texttt{,}\,\ldots\texttt{,}\,\mathit{decl}_n\ \texttt{end} \,\rrbracket_{\mathrm{TYP}} = \tau}$$

3.4.3 Declarations:

$$\frac{\llbracket\, \mathit{type} \,\rrbracket_{\mathrm{TYP}} = \tau}{\llbracket\, \texttt{typ}\ \mathit{typ\text{-}name}\,\texttt{:}\,\mathit{type} \,\rrbracket_{\mathrm{TYP}} = \tau}$$

$$\frac{\begin{array}{c} \forall i\colon \llbracket\, \mathit{decl}_i \,\rrbracket_{\mathrm{TYP}} = \tau_i \quad \forall i\colon \tau_i \in \{\mathit{boolean}, \mathit{integer}, \mathit{char}, \mathit{string}\} \cup \{\mathit{ptr}(\tau) \mid \tau \in \mathrm{T}\} \\ \llbracket\, \mathit{type} \,\rrbracket_{\mathrm{TYP}} = \tau \quad \tau \in \{\mathit{boolean}, \mathit{integer}, \mathit{char}, \mathit{string}, \mathit{void}\} \cup \{\mathit{ptr}(\tau) \mid \tau \in \mathrm{T}\} \\ \llbracket\, \mathit{expr} \,\rrbracket_{\mathrm{TYP}} = \tau \end{array}}{\llbracket\, \texttt{fun}\ \mathit{fun\text{-}name}\,\texttt{(}\mathit{decl}_1\texttt{,}\,\mathit{decl}_2\texttt{,}\,\ldots\texttt{,}\,\mathit{decl}_n\texttt{)}\,\texttt{:}\,\mathit{type}\ (=\mathit{expr})_{\mathrm{opt}} \,\rrbracket_{\mathrm{TYP}} = (\tau_1, \tau_2 \ldots, \tau_n) \to \tau}$$

$$\frac{\llbracket\, \mathit{type} \,\rrbracket_{\mathrm{TYP}} = \tau}{\llbracket\, \texttt{var}\ \mathit{var\text{-}name}\,\texttt{:}\,\mathit{type} \,\rrbracket_{\mathrm{TYP}} = \tau} \qquad \frac{\llbracket\, \mathit{type} \,\rrbracket_{\mathrm{TYP}} = \tau}{\llbracket\, \mathit{par\text{-}name}\,\texttt{:}\,\mathit{type} \,\rrbracket_{\mathrm{TYP}} = \tau} \qquad \frac{\llbracket\, \mathit{type} \,\rrbracket_{\mathrm{TYP}} = \tau}{\llbracket\, \mathit{comp\text{-}name}\,\texttt{:}\,\mathit{type} \,\rrbracket_{\mathrm{TYP}} = \tau}$$

3.5 Memory objects

Semantic function $\llbracket \cdot \rrbracket_{\mathrm{MEM}}$

$$\llbracket \cdot \rrbracket_{\mathrm{MEM}} \colon \mathcal{P} \to \{\mathsf{true}, \mathsf{false}\}$$

maps phrases of PREV to boolean values: it maps a phrase to true if and only if it is a value expression and denotes an object in memory. It is defined by the following rules:

$$\frac{[\![\,\mathrm{DECL}(\mathit{var\text{-}name})\,]\!]_{\text{TYP}} = \tau}{[\![\,\mathit{var\text{-}name}\,]\!]_{\text{MEM}} = \mathsf{true}} \qquad \frac{[\![\,\mathrm{DECL}(\mathit{par\text{-}name})\,]\!]_{\text{TYP}} = \tau}{[\![\,\mathit{par\text{-}name}\,]\!]_{\text{MEM}} = \mathsf{true}}$$

$$\frac{[\![\,\mathit{expr}\,]\!]_{\text{TYP}} = \tau \quad [\![\,\mathrm{DECL}_{\tau}(\mathit{comp\text{-}name})\,]\!]_{\text{TYP}} = \tau'}{[\![\,\mathit{expr}\,\texttt{.}\,\mathit{comp\text{-}name}\,]\!]_{\text{MEM}} = \mathsf{true}}$$

$$\frac{[\![\,\mathit{expr}_1\,]\!]_{\text{TYP}} = \mathit{arr}(n \times \tau) \quad [\![\,\mathit{expr}_2\,]\!]_{\text{TYP}} = \mathit{integer}}{[\![\,\mathit{expr}_1\,\texttt{[}\,\mathit{expr}_2\,\texttt{]}\,]\!]_{\text{MEM}} = \mathsf{true}}$$

$$\frac{[\![\,\mathit{expr}\,]\!]_{\text{TYP}} = \mathit{ptr}(\tau)}{[\![\,\mathit{expr}\,\texttt{\^{}}\,]\!]_{\text{MEM}} = \mathsf{true}}$$

In cases when no rule can be applied, the value of semantic function $[\![\,.\,]\!]_{\text{MEM}}$ is false.