Languages-beta: OC-L-01-Lexical-Conventions [*]

The PLanCompS Project

`OC-L-01-Lexical-Conventions.cbs` | PLAIN | PRETTY

OUTLINE

Language “`OCaml Light`”

1 Lexical conventions

Identifiers

$$
\begin{array}{rrcl}
\textit{Lexis} & I : \mathsf{ident} & ::= & \mathsf{capitalized\text{-}ident} \mid \mathsf{lowercase\text{-}ident} \\
& CI : \mathsf{capitalized\text{-}ident} & ::= & \mathsf{uppercase}\ (\mathsf{uppercase} \mid \mathsf{lowercase} \mid \mathsf{decimal} \mid \text{‘}\texttt{\_}\text{’} \mid \text{‘}\texttt{'}\text{’})^{*} \\
& LI : \mathsf{lowercase\text{-}ident} & ::= & \mathsf{lowercase}\ (\mathsf{uppercase} \mid \mathsf{lowercase} \mid \mathsf{decimal} \mid \text{‘}\texttt{\_}\text{’} \mid \text{‘}\texttt{'}\text{’})^{*} \\
& & \mid & \text{‘}\texttt{\_}\text{’}\ (\mathsf{uppercase} \mid \mathsf{lowercase} \mid \mathsf{decimal} \mid \text{‘}\texttt{\_}\text{’} \mid \text{‘}\texttt{'}\text{’})^{+} \\
& \mathsf{uppercase} & ::= & \text{‘}\texttt{A}\text{’} - \text{‘}\texttt{Z}\text{’} \\
& \mathsf{lowercase} & ::= & \text{‘}\texttt{a}\text{’} - \text{‘}\texttt{z}\text{’} \\
& \mathsf{decimal} & ::= & \text{‘}\texttt{0}\text{’} - \text{‘}\texttt{9}\text{’}
\end{array}
$$

$$
\begin{array}{rl}
\textit{Semantics} & \mathsf{id}[\![\ \_ : \mathsf{ident}\ ]\!] : \mathsf{ids} \\
\textit{Rule} & \mathsf{id}[\![\ I\ ]\!] = \text{“}I\text{”}
\end{array}
$$

[*] Suggestions for improvement: `plancomps@gmail.com`.
Reports of issues: `https://github.com/plancomps/CBS-beta/issues`.

Integer literals

Syntax *IL* : integer-literal ::= '-'?_natural-literal

NL : natural-literal ::= decimal-plus
| ('0x' | '0X') hexadecimal-plus
| ('0o' | '0O') octal-plus
| ('0b' | '0B') binary-plus

Lexis *DP* : decimal-plus ::= decimal^{+}

HP : hexadecimal-plus ::= $(\text{decimal} \mid \text{'A'} - \text{'F'} \mid \text{'a'} - \text{'f'})^{+}$

OP : octal-plus ::= $(\text{'0'} - \text{'7'})^{+}$

BP : binary-plus ::= $(\text{'0'} \mid \text{'1'})^{+}$

Semantics integer-value⟦ _ : integer-literal ⟧ : ⇒ implemented-integers

Rule integer-value⟦ '-' *NL* ⟧ = integer-negate(integer-value⟦ *NL* ⟧)

Rule integer-value⟦ *DP* ⟧ = implemented-integer decimal-natural("*DP*")

Floating-point literals

Syntax *FL* : float-literal ::= '-'?_non-negative-float-literal

NNFL : non-negative-float-literal ::= decimal-plus_'.'_decimal-plus
| decimal-plus_'.'
| decimal-plus_'.'_decimal-plus_float-exponent
| decimal-plus_'.'_float-exponent
| decimal-plus_float-exponent

FE : float-exponent ::= ('e' | 'E')_('+' | '-')?_decimal-plus

Rule ⟦ DP_1 '.' DP_2 ⟧ : non-negative-float-literal = ⟦ DP_1 '.' DP_2 'e' '1' ⟧

Rule ⟦ *DP* '.' ⟧ : non-negative-float-literal = ⟦ *DP* '.' '0' 'e' '1' ⟧

Rule ⟦ *DP* '.' *FE* ⟧ : non-negative-float-literal = ⟦ *DP* '.' '0' *FE* ⟧

Rule ⟦ *DP* *FE* ⟧ : non-negative-float-literal = ⟦ *DP* '.' '0' *FE* ⟧

Rule ⟦ 'e' '+' *DP* ⟧ : float-exponent = ⟦ 'e' *DP* ⟧

Rule ⟦ 'E' '+' *DP* ⟧ : float-exponent = ⟦ 'e' *DP* ⟧

Rule ⟦ 'E' '-' *DP* ⟧ : float-exponent = ⟦ 'e' '-' *DP* ⟧

Semantics float-value⟦ _ : float-literal ⟧ : ⇒ implemented-floats

float-value⟦ _ ⟧ is unspecified if the literal value is not representable in floats(implemented-floats-format).

Rule float-value⟦ '-' *NNFL* ⟧ =
float-negate(implemented-floats-format, float-value⟦ *NNFL* ⟧)

Rule float-value⟦ DP_1 '.' DP_2 'e' DP_3 ⟧ =
decimal-float(
implemented-floats-format, "DP_1", "DP_2", "DP_3")

Rule float-value⟦ DP_1 '.' DP_2 'e' '-' DP_3 ⟧ =
decimal-float(
implemented-floats-format, "DP_1", "DP_2", cons('-', "DP_3"))

Character literals

$$\begin{array}{lrcl}
\textit{Syntax} & CL : \textsf{char-literal} & ::= & \text{‘}\texttt{'}\text{’}\_\textsf{regular-char}\_\text{‘}\texttt{'}\text{’} \\
 & & | & \text{‘}\texttt{'}\text{’}\_\textsf{escape-sequence}\_\text{‘}\texttt{'}\text{’} \\
 & ES : \textsf{escape-sequence} & ::= & \text{‘}\backslash\text{’}\_\textsf{escaped-char} \\
 & & | & \text{‘}\backslash\text{’}\_\textsf{escaped-char-code} \\
\textit{Lexis} & RC : \textsf{regular-char} & ::= & \sim(\text{‘}\texttt{'}\text{’} \mid \text{‘}\backslash\text{’}) \\
 & EC : \textsf{escaped-char} & ::= & \text{‘}\backslash\text{’} \mid \text{‘}\texttt{"}\text{’} \mid \text{‘}\texttt{'}\text{’} \mid \text{‘}\texttt{n}\text{’} \mid \text{‘}\texttt{t}\text{’} \mid \text{‘}\texttt{b}\text{’} \mid \text{‘}\texttt{r}\text{’} \mid \text{‘}\ \text{’} \\
 & ECC : \textsf{escaped-char-code} & ::= & \textsf{decimal decimal decimal}
\end{array}$$

$$\begin{array}{rl}
\textit{Semantics} & \textsf{character-value}\llbracket\ \_ : \textsf{char-literal}\ \rrbracket : \Rightarrow \textsf{implemented-characters} \\
\textit{Rule} & \textsf{character-value}\llbracket\ \text{‘}\texttt{'}\text{’}\ RC\ \text{‘}\texttt{'}\text{’}\ \rrbracket = \textsf{ascii-character}(\text{“}RC\text{”}) \\
\textit{Rule} & \textsf{character-value}\llbracket\ \text{‘}\texttt{'}\text{’}\ ES\ \text{‘}\texttt{'}\text{’}\ \rrbracket = \textsf{capture}\llbracket\ ES\ \rrbracket
\end{array}$$

$$\begin{array}{rl}
\textit{Semantics} & \textsf{capture}\llbracket\ \_ : \textsf{escape-sequence}\ \rrbracket : \textsf{implemented-characters} \\
\textit{Rule} & \textsf{capture}\llbracket\ \text{‘}\backslash\text{’}\ \text{‘}\backslash\text{’}\ \rrbracket = \textsf{backslash} \\
\textit{Rule} & \textsf{capture}\llbracket\ \text{‘}\backslash\text{’}\ \text{‘}\texttt{'}\text{’}\ \rrbracket = \text{‘}\texttt{'}\text{’} \\
\textit{Rule} & \textsf{capture}\llbracket\ \text{‘}\backslash\text{’}\ \text{‘}\texttt{n}\text{’}\ \rrbracket = \textsf{line-feed} \\
\textit{Rule} & \textsf{capture}\llbracket\ \text{‘}\backslash\text{’}\ \text{‘}\texttt{t}\text{’}\ \rrbracket = \textsf{horizontal-tab} \\
\textit{Rule} & \textsf{capture}\llbracket\ \text{‘}\backslash\text{’}\ \text{‘}\texttt{b}\text{’}\ \rrbracket = \textsf{backspace} \\
\textit{Rule} & \textsf{capture}\llbracket\ \text{‘}\backslash\text{’}\ \text{‘}\texttt{r}\text{’}\ \rrbracket = \textsf{carriage-return} \\
\textit{Rule} & \textsf{capture}\llbracket\ \text{‘}\backslash\text{’}\ ECC\ \rrbracket = \\
 & \quad \textsf{checked implemented-character unicode-character decimal-natural}(\text{“}ECC\text{”})
\end{array}$$

String literals

$$\begin{array}{lrcl}
\textit{Syntax} & SL : \textsf{string-literal} & ::= & \text{‘}\texttt{"}\text{’}\_\textsf{string-character-star}\_\text{‘}\texttt{"}\text{’} \\
 & SCS : \textsf{string-character-star} & ::= & \textsf{string-character}\_\textsf{string-character-star} \\
 & & | & () \\
 & SC : \textsf{string-character} & ::= & \textsf{regular-string-char} \\
 & & | & \textsf{escape-sequence} \\
\textit{Lexis} & RSC : \textsf{regular-string-char} & ::= & \sim(\text{‘}\texttt{"}\text{’} \mid \text{‘}\backslash\text{’})
\end{array}$$

$$\begin{array}{rl}
\textit{Semantics} & \textsf{string-value}\llbracket\ \_ : \textsf{string-literal}\ \rrbracket : \Rightarrow \textsf{implemented-strings} \\
\textit{Rule} & \textsf{string-value}\llbracket\ \text{‘}\texttt{"}\text{’}\ SCS\ \text{‘}\texttt{"}\text{’}\ \rrbracket = \\
 & \quad \textsf{checked implemented-string}\ [\textsf{string-chars}\llbracket\ SCS\ \rrbracket]
\end{array}$$

$$\begin{array}{rl}
\textit{Semantics} & \textsf{string-chars}\llbracket\ \_ : \textsf{string-character-star}\ \rrbracket : \Rightarrow \textsf{implemented-characters}^{*} \\
\textit{Rule} & \textsf{string-chars}\llbracket\ \ \rrbracket = \\
\textit{Rule} & \textsf{string-chars}\llbracket\ SC\ SCS\ \rrbracket = \textsf{string-capture}\llbracket\ SC\ \rrbracket, \textsf{string-chars}\llbracket\ SCS\ \rrbracket
\end{array}$$

$$\begin{array}{rl}
\textit{Semantics} & \textsf{string-capture}\llbracket\ \_ : \textsf{string-character}\ \rrbracket : \textsf{implemented-characters} \\
\textit{Rule} & \textsf{string-capture}\llbracket\ RSC\ \rrbracket = \textsf{ascii-character}(\text{“}RSC\text{”}) \\
\textit{Rule} & \textsf{string-capture}\llbracket\ ES\ \rrbracket = \textsf{capture}\llbracket\ ES\ \rrbracket
\end{array}$$

Prefix and infix symbols

```
Lexis   PS : prefix-symbol  ::=  ‘!’ operator-char*
                              |  (‘?’ | ‘~’) operator-char+
                operator-char  ::=  ‘!’ | ‘$’ | ‘%’ | ‘&’ | ‘*’ | ‘+’ | ‘-’ | ‘.’ | ‘/’
                              |  ‘:’ | ‘<’ | ‘=’ | ‘>’ | ‘?’ | ‘@’ | ‘^’ | ‘|’ | ‘~’
  operator-char-not-asterisk  ::=  ‘!’ | ‘$’ | ‘%’ | ‘&’ | ‘+’ | ‘-’ | ‘.’ | ‘/’
                              |  ‘:’ | ‘<’ | ‘=’ | ‘>’ | ‘?’ | ‘@’ | ‘^’ | ‘|’ | ‘~’
       operator-char-not-bar  ::=  ‘!’ | ‘$’ | ‘%’ | ‘&’ | ‘*’ | ‘+’ | ‘-’ | ‘.’ | ‘/’
                              |  ‘:’ | ‘<’ | ‘=’ | ‘>’ | ‘?’ | ‘@’ | ‘^’ | ‘~’
 operator-char-not-ampersand  ::=  ‘!’ | ‘$’ | ‘%’ | ‘*’ | ‘+’ | ‘-’ | ‘.’ | ‘/’
                              |  ‘:’ | ‘<’ | ‘=’ | ‘>’ | ‘?’ | ‘@’ | ‘^’ | ‘|’ | ‘~’
```

Keywords

```
Lexis   keyword  ::=  ‘and’ | ‘as’ | ‘assert’ | ‘asr’ | ‘begin’ | ‘class’
                   |  ‘constraint’ | ‘do’ | ‘done’ | ‘downto’ | ‘else’ | ‘end’
                   |  ‘exception’ | ‘external’ | ‘false’ | ‘for’ | ‘fun’ | ‘function’
                   |  ‘functor’ | ‘if’ | ‘in’ | ‘include’ | ‘inherit’ | ‘initializer’
                   |  ‘land’ | ‘lazy’ | ‘let’ | ‘lor’ | ‘lsl’ | ‘lsr’
                   |  ‘lxor’ | ‘match’ | ‘method’ | ‘mod’ | ‘module’ | ‘mutable’
                   |  ‘new’ | ‘nonrec’ | ‘object’ | ‘of’ | ‘open’ | ‘or’
                   |  ‘private’ | ‘rec’ | ‘sig’ | ‘struct’ | ‘then’ | ‘to’
                   |  ‘true’ | ‘try’ | ‘type’ | ‘val’ | ‘virtual’ | ‘when’
                   |  ‘while’ | ‘with’
```