Strik Spec

Mizunashi Mana

2023 年 12 月 20 日

概要

Strik, a programming language.

目次

1 Preliminaries

1.1 Grammar Notation

[For02]

2 Syntax

2.1 Lexical

Overview:

```
lexical_program ::= (whitespace / lexeme)* EOS
        lexeme ::= literal_part
                 / literal
                 / special_char
                 / keyword_id
                 / keyword_sym
                 / free_id
                 / var_id
                 / var_sym
                 / con_id
                 / con_sym
```

Identifier:

```
    var_id ::= id_small_char id_char* !id_char
    con_id ::= id_large_char id_char* !id_char
   var_sym ::= sym_normal_char sym_char* !sym_char
   con_sym ::= sym_sp_char sym_char* !sym_char
    free_id ::= keyword_prefix_char string
```

Reserved:

```
keyword_prefixed ::= keyword_prefix_char id_char+ !id_char
                   / keyword_prefix_char sym_char+ !sym_char
                   / keyword_prefix_char ("{" / "[" / "(")
      keyword_id ::= keyword_id_unit !id_char
 keyword_id_unit ::= "_"
     keyword_sym ::= keyword_sym_unit !sym_char
keyword_sym_unit ::= "="
                   / "^"
                   / ":"
                   / "\"
```

Literal Overview:

```
literal_part ::= interp_string_part
     literal ::= string
               / rational
               / integer
```

Number Literal:

```
rational      ::= sign_char? (! zero_char) decimal num_dot_sym_char decimal
integer       ::= number_prefix hexit_prefix_char heximal
                / number_prefix digit_prefix_char decimal
                / sign_char? (! zero_char) decimal
number_prefix ::= sign_char? zero_char
decimal       ::= digit_char (digit_char / num_sep_sym_char)* !(digit_char / num_sep_sym_char)
heximal       ::= hexit_char (hexit_char / num_sep_sym_char)* !(hexit_char / num_sep_sym_char)
sign_char     ::= "+"
                / "-"
zero_char     ::= "\p{Digit=0}"
num_dot_sym_char ::= "."
num_sep_sym_char ::= "_"
hexit_char    ::= digit_char
                / "A" / "B" / ⋯ / "F"
                / "a" / "b" / ⋯ / "f"
hexit_prefix_char ::= "x" / "X"
digit_prefix_char ::= "d" / "D"
```

String Literal:

```
interp_string_part ::= interp_string_start
                     / interp_string_cont
                     / interp_string_end
string             ::= string_sep_char interp_string_graphic* string_sep_char
interp_string_start ::= string_sep_char interp_string_graphic* interp_open
interp_string_cont  ::= interp_close interp_string_graphic* interp_open
interp_string_end   ::= interp_close interp_string_graphic* string_sep_char
interp_open         ::= interp_open_char "{"
interp_close        ::= keyword_prefix_char "}"
escape_open_char    ::= "\"
interp_open_char    ::= keyword_prefix_char
interp_string_graphic ::= uni_escape
                        / bstr_graphic
bstr_graphic        ::= byte_escape
                      / char_escape
                      / bstr_graphic_char
bstr_graphic_char   ::= white_char
                      / (!(escape_open_char / string_sep_char / interp_open_char)) graphic_char
uni_escape          ::= escape_open_char unicode_prefix_char "{" hexit_char+ "}"
byte_escape         ::= escape_open_char hexit_prefix_char hexit_char hexit_char
char_escape         ::= escape_open_char charesc_char
charesc_char        ::= "0" / "a" / "b" / "f" / "n" / "r" / "t" / "v"
                      / escape_open_char / string_sep_char / interp_open_char
unicode_prefix_char ::= "u" / "U"
```

White Space:

```
whitespace ::= whitestuff+
whitestuff ::= white_char
             / comment
```

Comment:

comment ::= *line_comment*
/ *multiline_comment*
line_comment ::= *line_comment_open any_1l_char*∗ (*newline* / **EOS**)
multiline_comment ::= *multiline_comment_open*; *anys* (*multiline_comment_close* / **EOS**)
line_comment_open ::= `"//"`
multiline_comment_open ::= `"/*"`
multiline_comment_close ::= `"*/"`
any_1l_char ::= *graphic_char*
/ *space_char*
anys ::= ((! *multiline_comment_close*) *any_char*)∗
any_char ::= *graphic_char*
/ *white_char*

Base Unit:

graphic_char ::= *small_char*
/ *large_char*
/ *symbol_char*
/ *digit_char*
/ *other_char*
/ *special_char*
/ *other_special_char*
/ *other_graphic_char*
id_char ::= *id_small_char*
/ *id_large_char*
/ *digit_char*
/ *other_char*
id_small_char ::= *small_char*
id_large_char ::= *large_char*
sym_char ::= *sym_normal_char*
/ *sym_sp_char*
/ *other_char*
sym_normal_char ::= (! *sym_sp_char*) *symbol_char*
sym_sp_char ::= `"~"`
white_char ::= `"\v"`
/ *space_char*
/ *newline_char*
space_char ::= `"\t"`
/ `"\u{200E}"`
/ `"\u{200F}"`
/ `"\p{General_Category=Space_Separator}"`
newline ::= `"\r\n"`
/ *newline_char*
newline_char ::= `"\r"`
/ `"\n"`
/ `"\f"`
/ `"\p{General_Category=Line_Separator}"`
/ `"\p{General_Category=Paragraph_Separator}"`

```
small_char      ::= "\p{General_Category=Lowercase_Letter}"
                  / "\p{General_Category=Other_Letter}"
                  / "_"
large_char      ::= "\p{General_Category=Uppercase_Letter}"
                  / "\p{General_Category=Titlecase_Letter}"
symbol_char     ::= (!(special_char / other_special_char / "_")) symbol_cat_char
symbol_cat_char ::= "\p{General_Category=Connector_Punctuation}"
                  / "\p{General_Category=Dash_Punctuation}"
                  / "\p{General_Category=Other_Punctuation}"
                  / "\p{General_Category=Symbol}"
digit_char      ::= "\p{General_Category=Decimal_Number}"
other_char      ::= (! white_char) other_cat_char
other_cat_char  ::= "\p{General_Category=Modifier_Letter}"
                  / "\p{General_Category=Mark}"
                  / "\p{General_Category=Letter_Number}"
                  / "\p{General_Category=Other_Number}"
                  / "\p{General_Category=Format}"

special_char      ::= "{"
                    / "}"
                    / "["
                    / "]"
                    / "("
                    / ")"
                    / ";"
                    / "."
other_special_char ::= keyword_prefix_char
                    / string_sep_char
                    / "'"
keyword_prefix_char ::= "#"
string_sep_char     ::= "\\"
other_graphic_char  ::= (!(symbol_cat_char / special_char / other_special_char)) other_graphic_cat_char
other_graphic_cat_char ::= "\p{General_Category=Punctuation}"
```

2.2 Layout

Pre-Process:

```
1:  procedure PRE_PARSE
2:      pre_line ← 1
3:      should_open_imp_layout ← ⊤                ▷ Insert the top-level implicit layout.
4:      while (current_token, current_line, current_col) ← consume do
5:          if current_token ∉ lexeme then
6:              continue                          ▷ Skip white-spaces.
7:          end if
8:          if should_open_imp_layout then
9:              yield {open_imp(current_col)}
10:             should_open_imp_layout ← ⊥
11:         end if
12:         if pre_line < current_line then
13:             yield {newline(current_col)}
14:             pre_line ← current_line
15:         end if
```

```
        yield current_token
        if current_token ∈ lb_imp_open ∪ lp_imp_open then
            should_open_imp_layout ← ⊤
        else if current_token ∈ lb_exp_open ∪ lp_exp_open then
            yield {open_exp}
        else if current_token ∈ lb_close ∪ lp_close then
            yield {close}
        end if
    end while
    if should_open_imp_layout then
        yield {open_imp(0)}
        should_open_imp_layout ← ⊥
    end if
    yield {close}
end procedure
```

Add Layout Tokens:

```
procedure WITH_LAYOUT_TOKEN
    layout_stack ← []
    while current_token ← consume do
        if {open_imp(m)} ← current_token then
            layout_stack.push({m})
        else if {open_exp} ← current_token then
            layout_stack.push({−})
        else if {close} ← current_token then
            if layout_stack.is_empty() then
                error                                   ▷ Too many layout closing.
            else
                layout_stack.pop()
            end if
        else if {newline(c)} ← current_token then
            if layout_stack.is_empty() then
                error                                   ▷ Too early layout closing.
            else if {m} ← layout_stack.get() then
                if c < m then
                    error                               ▷ Less indentation in the current implicit layout.
                else if c = m then
                    yield <;>
                else if c > m then
                    continue
                end if
            else if {−} ← layout_stack.get() then
                continue                                ▷ Skip newlines in explicit layouts.
            end if
        else
```

```
            yield current_token
        end if
    end while
end procedure
```

2.3 Grammar

Program:

$$\mathit{program} \quad ::= \quad \mathit{expr}\ \textbf{EOS}$$

Local Declaration:

$$\begin{array}{rcl}
\mathit{local_decl} & ::= & \texttt{"\#let"}\ \mathit{let_body} \\
& / & \texttt{"\#rec"}\ \mathit{let_body} \\
\mathit{local_type_decl} & ::= & \texttt{"\#let"}\ \mathit{let_type_body} \\
\mathit{let_body} & ::= & \mathit{lb_open}\ \mathit{let_body_items}\ \mathit{lb_close} \\
& / & \mathit{let_body_item} \\
\mathit{let_body_items} & ::= & \mathit{lsemis}?\ \mathit{let_body_item}\ (\mathit{lsemis}\ \mathit{let_body_item})*\ \mathit{lsemis}? \\
\mathit{let_body_item} & ::= & \mathit{bind_prom_type} \\
& / & \mathit{bind_expr} \\
\mathit{let_type_body} & ::= & \mathit{lb_open}\ \mathit{let_type_body_items}\ \mathit{lb_close} \\
& / & \mathit{let_type_body_item} \\
\mathit{let_type_body_items} & ::= & \mathit{lsemis}?\ \mathit{let_type_body_item}\ (\mathit{lsemis}\ \mathit{let_type_body_item})*\ \mathit{lsemis}? \\
\mathit{let_type_body_item} & ::= & \mathit{bind_type}
\end{array}$$

Where Declaration:

$$\begin{array}{rcl}
\mathit{where_body} & ::= & \mathit{lb_open}\ \mathit{where_body_items}\ \mathit{lb_close} \\
& / & \mathit{where_body_item} \\
\mathit{where_body_items} & ::= & \mathit{lsemis}?\ \mathit{where_body_item}\ (\mathit{lsemis}\ \mathit{where_body_item})*\ \mathit{lsemis}? \\
& / & \mathit{lsemis}? \\
\mathit{where_body_item} & ::= & \mathit{bind_prom_type} \\
& / & \mathit{bind_expr} \\
& / & \mathit{local_decl} \\
\mathit{bind_expr} & ::= & \mathit{declvar}\ (\texttt{":"}\ \mathit{type})?\ \texttt{"="}\ \mathit{expr} \\
\mathit{bind_type} & ::= & \mathit{declvar}\ (\texttt{":"}\ \mathit{type})?\ \texttt{"="}\ \mathit{type} \\
\mathit{bind_prom_type} & ::= & \texttt{"\^{}"}\ \mathit{declvar}\ (\texttt{":"}\ \mathit{type})?\ \texttt{"="}\ \mathit{type}
\end{array}$$

Expression:

```
expr          ::= expr_ann ("#where" where_body)*
expr_ann      ::= expr_infix ":" type
                / expr_infix
expr_infix    ::= expr_apps (expr_op expr_apps)*
expr_op       ::= "#op" lp_open lsemis? expr lsemis? lp_close
                / sym
expr_apps     ::= expr_struct expr_struct*
expr_struct   ::= "\" expr
                / "#match" expr_tuple_items "#in" expr
                / "#case" case_body
                / "#if" case_body
                / expr_atomic
expr_atomic   ::= expr_block
                / expr_literal
                / con
                / var
expr_literal  ::= literal
                / expr_interp_string
                / expr_tuple

case_body     ::= lb_open case_items lb_close
                / case_item
case_items    ::= lsemis? case_item (lsemis case_item)* lsemis?
                / lsemis?
case_item     ::= view "#>" expr

expr_block    ::= lb_open expr_block_items lb_close
                / lb_open expr_block_stmts lb_close
                / lb_open lsemis? lb_close
expr_block_items ::= lsemis? expr_block_item (lsemis expr_block_item)* lsemis?
expr_block_item  ::= expr_block_pats expr_block_guard? "#>" expr
expr_block_pats  ::= lsemis? pat (lsemis pat)* lsemis?
                / lsemis?
expr_block_guard ::= "#if" view
expr_block_stmts ::= lsemis? expr_block_stmt (lsemis expr_block_stmt)* lsemis?
expr_block_stmt  ::= expr
                / local_decl

expr_interp_string      ::= interp_string_start expr_block_stmts expr_interp_string_item* interp_string_end
expr_interp_string_item ::= interp_string_cont expr_block_stmts

expr_tuple    ::= lp_open expr_tuple_items lp_close
expr_tuple_items ::= lsemis? expr_tuple_item (lsemis expr_tuple_item)* lsemis?
                / lsemis?
expr_tuple_item  ::= bind_prom_type
                / bind_expr
                / "^" type
                / expr
                / local_decl
```

Type Expression:

```
type          ::= type_ann ("#where" where_body)*
type_ann      ::= type_infix ":" type
                / type_infix
type_infix    ::= type_apps (type_op type_apps)*
type_op       ::= "#op" lp_open lsemis? type lsemis? lp_close
                / sym
type_apps     ::= type_atomic type_atomic*
type_atomic   ::= type_block
                / type_literal
                / con
                / var
type_literal  ::= literal
                / type_tuple
                / type_tuple_sig

type_block    ::= lb_open type_block_stmts lb_close
                / lb_open type_block_stmts lb_close
                / lb_open lsemis? lb_close
type_block_stmts ::= lsemis? type_block_stmt (lsemis type_block_stmt)* lsemis?
type_block_stmt  ::= type
                / local_type_decl

type_tuple    ::= lp_open type_tuple_items lp_close
type_tuple_items ::= lsemis? type_tuple_item (lsemis type_tuple_item)* lsemis?
                / lsemis?
type_tuple_item  ::= bind_prom_type
                / bind_type
                / "^" type_infix
                / type_infix
                / local_type_decl

type_tuple_sig ::= lp_open type_tuple_sig_items lp_close
type_tuple_sig_items ::= lsemis? type_tuple_sig_item (lsemis type_tuple_sig_item)* lsemis?
                / lsemis?
type_tuple_sig_item ::= "^" declvar ":" type
                / declvar ":" type
                / "^" type_infix
                / type_infix
```

Pattern:

```
pat           ::= pat_ann
pat_ann       ::= pat_infix ":" type
                / pat_infix
pat_infix     ::= pat_apps (pat_op pat_apps)*
pat_op        ::= "#op" lp_open lsemis? con lsemis? lp_close
                / con_sym
pat_apps      ::= con pat_atomic*
pat_atomic    ::= lb_open lsemis? pat lsemis? lb_close
                / pat_literal
                / var
pat_literal   ::= literal
                / pat_tuple
```

```
pat_tuple       ::= lp_open pat_tuple_items lp_close
pat_tuple_items ::= lsemis? pat_tuple_item (lsemis pat_tuple_item)* lsemis?
                  / lsemis?
pat_tuple_item  ::= declvar "=" pat
                  / pat
```

View:

```
view           ::= lb_open view_and_items lb_close
                 / "#let" let_pat_body
                 / expr
view_and_items ::= lsemis? view (lsemis view)* lsemis?
                 / lsemis?
let_pat_body   ::= lb_open let_pat_body_items lb_close
                 / let_pat_body_item
let_pat_body_items ::= lsemis? let_pat_body_item (lsemis let_pat_body_item)* lsemis?
let_pat_body_item  ::= pat "=" expr
```

Base Unit:

```
declvar ::= "#id" lp_open lsemis? (var_sym  / var_id) lsemis? lp_close
          / var_id
          / free_id
sym     ::= con_sym
          / var_sym
con     ::= "#id" lp_open lsemis? (con_sym  / con_id) lsemis? lp_close
          / con_id
var     ::= declvar
```

Layout Unit:

```
lb_open     ::= lb_imp_open
              / lb_exp_open
lb_imp_open ::= "{"
lb_exp_open ::= "#{"
lb_close    ::= "}"
lp_open     ::= lp_imp_open
              / lp_exp_open
lp_imp_open ::= "("
lp_exp_open ::= "#("
lp_close    ::= ")"
lsemis      ::= (<;> / ";")+
```

2.4 Fixity Resolution

2.5 Abstract Tree

Program:

```
program ::= expr
```

3 Type System

3.1 Declarative

3.2 Bidirectional

3.3 Algorithmic Bidirectional

参考文献

[For02] Bryan Ford. Packrat Parsing : a Practical Linear-Time Algorithm with Backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.