The programming language PINS

in the subject of Compilers and Virtual Machines in the academic year 2022/23.

1 Lexical rules

- Keywords:

  `arr else for fun if then typ var where while`

- Names of atomic data types:

  `logical integer string`

- Constants of atomic data types:

  `logical :` `true false`

  `integer :` Arbitrary signed sequence of digits.

  `string :` Arbitrary (possibly empty) sequence of characters with ASCII codes between 32_{10} and 126_{10}, enclosed in single quotation marks (', ASCII code 39_{10}); the exception is the character ', which is duplicated.

- Names:

  Arbitrary sequence of letters, digits, and underscores that does not start with a digit and is not a keyword, name, or constant of an atomic data type.

- Other symbols:

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

- Comments:

  A comment is an arbitrary text that begins with the "#" symbol (ASCII code 35_{10}) and extends until the end of the line.

- Whitespace:

  Whitespace refers to the characters that represent empty space or formatting in text. This includes the space character (ASCII code 32_{10}), the tab character (ASCII code 9_{10}), and the newline characters (ASCII codes 10_{10} and 13_{10}), which represent the end of a line.

2 Syntax rules

$$source \longrightarrow definitions$$

$$definitions \longrightarrow definition$$
$$definitions \longrightarrow definitions \;\; ; \;\; definition$$

$$definition \longrightarrow type\_definition$$
$$definition \longrightarrow function\_definition$$
$$definition \longrightarrow variable\_definition$$

$$type\_definition \longrightarrow \texttt{typ}\ \text{identifier} \;\; \texttt{:} \;\; type$$

$$type \longrightarrow \text{identifier}$$

type ⟶ logical
type ⟶ integer
type ⟶ string
type ⟶ arr [int_const] type

function_definition ⟶ fun identifier (parameters) : type = expression

parameters ⟶ parameter
parameters ⟶ parameters , parameter

parameter ⟶ identifier : type

expression ⟶ logical_ior_expression
expression ⟶ logical_ior_expression { WHERE definitions }

logical_ior_expression ⟶ logical_ior_expression | logical_and_expression
logical_ior_expression ⟶ logical_and_expression

logical_and_expression ⟶ logical_and_expression & compare_expression
logical_and_expression ⟶ compare_expression

compare_expression ⟶ additive_expression == additive_expression
compare_expression ⟶ additive_expression != additive_expression
compare_expression ⟶ additive_expression <= additive_expression
compare_expression ⟶ additive_expression >= additive_expression
compare_expression ⟶ additive_expression < additive_expression
compare_expression ⟶ additive_expression < additive_expression
compare_expression ⟶ additive_expression

additive_expression ⟶ additive_expression + multiplicative_expression
additive_expression ⟶ additive_expression - multiplicative_expression
additive_expression ⟶ multiplicative_expression

multiplicative_expression ⟶ multiplicative_expression * prefix_expression
multiplicative_expression ⟶ multiplicative_expression / prefix_expression
multiplicative_expression ⟶ multiplicative_expression % prefix_expression
multiplicative_expression ⟶ prefix_expression

prefix_expression ⟶ + prefix_expression
prefix_expression ⟶ - prefix_expression
prefix_expression ⟶ ! prefix_expression
prefix_expression ⟶ postfix_expression

postfix_expression ⟶ postfix_expression [expression]
postfix_expression ⟶ atom_expression

atom_expression ⟶ log_constant
atom_expression ⟶ int_constant
atom_expression ⟶ str_constant
atom_expression ⟶ identifier
atom_expression ⟶ identifier (expressions)
atom_expression ⟶ { expression = expression }
atom_expression ⟶ { if expression then expression }
atom_expression ⟶ { if expression then expression else expression }
atom_expression ⟶ { while expression : expression }
atom_expression ⟶ { for identifier = expression , expression , expression : expression }
atom_expression ⟶ (expressions)

expressions ⟶ expression
expressions ⟶ expressions , expression

variable_definition ⟶ var identifer : type

3 Semantic rules

Scopes

- A name is visible throughout its entire scope (from its declaration to its end) regardless of its definition's location.
- The expression of the form expression `{ WHERE definitions }` creates a new nested scope: the expression and all definitions are within the new nested scope of visibility.
- A function definition creates a new nested scope of visibility, which starts after the function name and extends until the end of the function definition.

Typing

Data types:

- logical, integer, and string describe the types LOGICAL, INTEGER, and STRING, respectively.
- If the value of the constant int_const is equal to n, and type describes the type τ, then

  arr[int_const] type

  describes type ARR(n, τ).

Declarations:

- Type declaration

  typ identifier : type,

  where type describes the type τ, it specifies that the identifier represents the type τ.

- Function declaration

  fun identifier (identifier 1 : **type** $_1$, **identifier** 2 : **type** $_2$, ..., **identifier** $_n$: **type** $_n$ **) : type** $=$ **expression,**

  In a function declaration, where: (a) type_i describes the type τ_i for $i \in 1, 2, \ldots, n$, (b) type describes the type τ, and (c) expression is of type τ, it specifies that the function identifier is of type $\tau_1 \times \tau_2 \times \ldots \times \tau_n \to \tau$.

- Variable declaration

$$\text{var identifier : type,}$$

  In a variable declaration, where type describes the type τ, it specifies that the variable identifier is of type τ.

- Parameter or component declaration

$$\text{identifier : type,}$$

  In a parameter or component declaration, where type describes the type τ, it specifies that the parameter or component identifier is of type τ.

Expressions:

- log_const, int_const, and str_const are of type LOGICAL, INTEGER, and STRING, respectively.
- If expression is of type LOGICAL, then !expression is of type LOGICAL as well.
- If expression is of type INTEGER, then both +expression and -expression are of type INTEGER.
- If *expression*$_1$ and *expression*$_2$ are of type LOGICAL, then

$$\text{expression }_1\text{ op expression }_2\text{ exp } \text{ when } op \in \{\&, |\}$$

  is type LOGICAL. - If *expression*$_1$ and *expression*$_2$ are of type INTEGER, then

$$\text{expression }_1\text{ op expression }_2\text{ when } \text{ op } \in \{+, -, *, /, \%\}$$

  is type INTEGER.

- If *expression*$_1$ and *expression*$_2$ are of type $\tau \in$ {LOGICAL, INTEGER}, then

$$\text{expression }_1\text{ op expression }_2\text{ when } \text{ op } \in \{==, ! =, <=, >=, <, >\}$$

  is type LOGICAL.

- If $expression_1$ is of type ARR(n, τ) and $expression_2$ is of type INTEGER, then

$$\text{expression }_1 [\text{ expression }_2]$$

is type τ.

- If identifier is of type $\tau_1 \times \tau_2 \times \ldots \times \tau_n \to \tau$, and $expression_i$ is of type τ_i for $i \in 1, 2, \ldots, n$, then the expression

$$\text{identifier (expression} \quad \text{expression }_2, \ldots, \text{ expression }_n)$$

is type τ.

- If expression is of type τ, then the expression is of the form:

$$\text{expression } \{ \text{ where definitions } \}$$

is type τ.

- If $expression_1$ and $expression_2$ are of type $\tau \in \{$LOGICAL, INTEGER, STRING$\}$, then

$$\{ \text{ expression }_1 = \text{ expression }_2 \}$$

is type τ.

- If expression is of type LOGICAL, then the expressions

$$\{\ \texttt{while}\ \mathit{expression}\ \texttt{:}\ \mathit{expression}'\ \} \quad ,$$
$$\{\ \texttt{if}\ \mathit{expression}\ \texttt{then}\ \mathit{expression}'\ \} \quad \text{in}$$
$$\{\ \texttt{if}\ \mathit{expression}\ \texttt{then}\ \mathit{expression}'\ \texttt{else}\ \mathit{expression}''\ \}$$

are type VOID.

- If $identifier$, $expression$, $expression_2$, and $expression_3$ are of type INTEGER, then the expression

$$\{ \text{ for identifier} = \text{expression }_1, \text{expression }_2, \text{expression }_3 : \text{expression }'\}$$

is type VOID.

- If $expression_i$ is of type τ_i for $i \in 1, 2, \ldots, n$, then the expression

$$(\text{ expression }_1, \text{expression }_2, \ldots, \text{expression }_n)$$

is type τ_n.