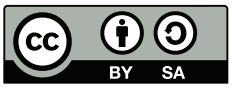
Report of the Astl Programming Language Draft in preparation of version 0.38 11 November 2019



Andreas F. Borchert

Contents

Chapter 1

Lexical elements

1.1 Source files

Astl program source files must end in the suffix ".ast" which is to be omitted when source files are refered to.

1.2 Character set

Astl program sources are to be encoded in UTF-8. While most Astl tokens including identifiers consist of ASCII characters only, arbitrary non-ASCII characters are permitted in string literals (see 1.10.2), program text literals (see 1.10.4), regular expression literals (see 1.10.3), and comments.

Case is significant.

1.3 Lines and columns

To provide locations for error messages, lines and columns are tracked while reading a program source. The newline character (ASCII 10) is interpreted as line terminator and considered otherwise as being equivalent to the space character (ASCII 32). Tabs (ASCII 9) are similarly treated as space characters but cause the current column to be advanced to the next multiply of 8 plus 1. Unicode codepoints are counted as one character independently from the number of bytes required to encode them in UTF-8.

1.4 Comments

Comments can be delimited by "/*" and "*/". Nested comments are not supported. Alternatively, comments start with "//" and are ended by the next line terminator. Comments are handled like space characters.

1.5 Tokens

Each program source is converted into a sequence of tokens during the lexical analysis. Tokens can be delimiters consisting of one or more special characters, reserved keywords, identifiers, or literals. Tokens end if the next character is a space or if the next character cannot be added to it.

Example: The character sequence *a*[*i2*]+=**exists** *f*{*i2*}? 12: *existsf* consists of the following tokens:

a	an identifier
[	a single-character delimiter
i2	an identifier
]	a single-character delimiter
+=	a compound delimiter
exists	a keyword
f	an identifier
{	a single-character delimiter
i2	an identifier
}	a single-character delimiter
?	a single-character delimiter
12	a decimal literal
:	a single-character delimiter
existsf	an identifier

1.6 Single-character delimiters

Following delimiters consist of one character only and cannot be part of a longer token which is not a literal:

```
{   }   [   ]
^   *   .   ,
?   :   ;
```

1.7 Compound delimiters

Following delimiters consist of one or multiple characters:

```
(     )     <(    )>
<     <=    >     >=
−     −>    −−    −=
+     +=    ++
&     &&    &=
!     !=
=     =~    ==
||
```

1.8 Keywords

Following keywords are reserved. They cannot be used as identifiers.

abstract	**div**	**library**	**pre**	**transformation**
and	**else**	**machine**	**print**	**var**
as	**elsif**	**mod**	**private**	**when**
at	**exists**	**nonassoc**	**retract**	**where**
attribution	**foreach**	**null**	**return**	**while**
cache	**if**	**on**	**right**	**x**
close	**import**	**operators**	**rules**	
create	**in**	**opset**	**shared**	
cut	**inplace**	**or**	**state**	
delete	**left**	**post**	**sub**	

1.9 Identifiers

Identifiers begin with a letter, i.e. “A” to “Z” and “a” to “z”, or an underscore “_”, and are optionally continued with a sequence of more letters, underscores, or decimal digits “0” to “9”. Some identifiers are predefined (see 14).

1.10 Literals

Literals can be decimal literals, string literals (see 1.10.2), regular expressions (see 1.10.3, and program text literals (see 1.10.4).

1.10.1 Decimal literals

Decimal literals begin with a decimal digit “0” to “9” and optionally more digits. Decimal constants are unsigned and can be of arbitrary size.

1.10.2 String literals

String literals are delimited by “"”. Backslashes, i.e. “\”, are escape characters, i.e. they remove the special meaning of the following character or allow to insert special characters into a string:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
form feed	FF	\f
alert	BEL	\a
space		\␣
backslash	\	\\
question mark	?	\?
single quote	’	\’
double quote	"	\"
left brace	{	\{
right brace	}	\}
dollar	$	\$
octal number	*ooo*	\ *ooo*
16-bit Unicode codepoint in hex	*hhhh*	\u *hhhh*
32-bit Unicode codepoint in hex	*hhhhhhhh*	\U *hhhhhhhh*

(This includes the set of ⟨simple-escape-sequence⟩ and ⟨octal-escape-sequence⟩ of the C language as defined by the ISO standard 9899-1999.)

Examples: "Hello,␣world" represents the text “Hello, world”, "\"" represents the double quote “"”, "\\" represents the backslash “\”, and "Two\nlines" represents “Two”, followed by a line terminator, and “lines”. Unicode codepoints can be specified using four (using \u) or height hex digits (using \U). Example: "\U0001f37a␣Cheers!".

1.10.3 Regular expressions

A regular expression is a literal that is enclosed by the delimiters “m{” and “}”. Regular expressions are interpreted by the *Perl Compatible Regular Expression* library (PCRE).[1] Reg-

[1] See `http://www.pcre.org` and `http://www.pcre.org/current/doc/html/pcre2syntax.html`. The options *PCRE2_ALT_BSUX*, *PCRE2_UCP*, and *PCRE2_UTF* are set.

ular expressions must be encoded in UTF-8 and expect the text to be in UTF-8.

Example: *m*{[*a*−*zA*−*Z*_][*a*−*zA*−*Z*_*0*−9]∗}

1.10.4 Program text literals

A program text literal begins with "*q*{" and is ended by a balanced "}".[2] These literals are designed to support the generation of program text within the print rules (see 11). They may include embedded references to variables or embedded expressions that are interpolated when the literal is evaluated. As in string literals (see 1.10.2), the backslash "\" may be used to escape following characters. Braces may be used without preceding backslashes within a program text literal as long as they are balanced.

Example: *q*{{}} is equivalent to *q*{\{\}}.

Leading and trailing white spaces are removed unless escape sequences are used to protect them.

1.10.4.1 Embedded variable references

An embedded variable reference begins with the character "*$*"and is followed by an identifier. Spaces between "*$*"and the identifier following it are not permitted.

1.10.4.2 Embedded expressions

An embedded expression begins with the character sequence "*${*", is followed by an arbitrary expression, and closed by "}".

1.10.4.3 Interpretation of multi-line literals

Program text literals may extend over multiple lines. In this case, the indentations of the individual lines are interpreted relatively to each other.

Example: Following program text literal generates an if-else-statement construct that references the variables *condition*, *then_statement*, and *else_statement*. The white space between the opening "{" and "if" is ignored. Similarly, the trailing whitespace between *$else_statement* and the closing "}" is skipped. However, the relative indentations are noted, i.e. "*$then_statement*" is to be indented by three additional spaces.

```
q{
    if ($condition)
        $then_statement
    else
        $else_statement
}
```

[2] This type of literal is perhaps familiar to those knowing Perl. However, its semantics follows that of the "*qq*{...}" construct in Perl which allows interpolation.

Chapter 2

Types

Variables are untyped but values are typed. There exist twelve different types of values: **null**, Boolean values, integers, strings, lists, dictionaries, functions, abstract syntax trees, matching results of a regular expression, control flow graph nodes, input streams, and output streams.

2.1 Null value

Null values can have the value **null** only. Uninitialized variables are initially bound to **null** (see 4.6).

2.2 Boolean values

Boolean values can be either *true* or *false*.

The standard identifiers *true* and *false* may be used to denote the two possible Boolean values. In the following example, a variable named *flag* is declared and set to *false*. In the following statement the Boolean value of a logical expression is assigned to *flag*.

var *flag* = *false*;
flag = *flag* || (*i* == 7);

2.3 Integer values

Integer values can be of arbitrary size as their implementation is based on the GNU Multiple Precision Library[1]. Please note that floating point values are not supported.

2.4 String values

Strings are sequences of Unicode codepoints which can be of arbitrary length.

The following example demonstrates how implicitly strings can be converted to integers or integers to strings. Firstly, the variable *i* is initialized to the integer value 17. Secondly, *j* is initialized to the concatenation of the strings "17" (implicitly converted from the integer value of *i*) and "8". Finally, the sum of *i* and *j* is computed where *j* is converted to an integer first. These implicit conversions are enforced by the operators “&” which expects strings and “+” which expects integers.

[1] `https://gmplib.org/`

```
var i = 17;
var j = i & "8";
var k = i + j; // k is set to 195
```

When strings are converted to integers and vice versa, a decimal representation is expected or generated, respectively. Conversions of strings to integers are implemented using the *mpz_set_str* function of the GNU Multiple Precision Library with base set to 10. This means that initial white space characters are permitted in the string. However, trailing non-whitespace characters are not permitted.[2]

2.5 Lists

Lists are ordered sets of elements which can be accessed by using indices $0..n-1$.

The following example shows how lists can be constructed using the [...]-syntax and how to iterate through lists using a **foreach**- or **while**-statement:

```
var cities = ["Berlin", "Hamburg", "Stuttgart", "Ulm"];
foreach city in (cities) {
    println(city);
}
// alternatively, indices can be used:
var index = 0;
while (index < len(cities)) {
    println(cities[index]);
    ++index;
}
```

Expressions of list type reference the data structure that is actually representing the list. Consequently, an assignment of a list causes a list to be shared (shallow copy). Example:

```
var numbers = [1, 3, 5];
var more_numbers = numbers;
push(more_numbers, 7);
// len(numbers) == 4
```

By using the predefined functions *push* and *pop*, lists can be used as FIFO queues.

2.6 Dictionaries

Dictionaries are data structures that map string-valued keys to values of arbitrary type (including **null**).

The attribution rule in the following example shows how dictionaries can be used to construct symbol tables. Abstract syntax trees that are bound to variables (*id*, *declarator* and *block* in this example) can be used as dictionaries to attach attributes to a node of the syntax tree. In this example, *block.vars* designates the entry of the dictionary *block* selected by the key *vars*. Whenever a key is constant and an identifier (see 1.9), the dot-operator can be used as a selecting operator of dictionaries. Selector expressions can be put into {...} as demonstrated by {*id*} in this example. As shown below, dictionaries can be constructed using the {...} syntax. In this case the keys in front of the "->" delimiter must be identifiers.

```
("direct_declarator" ("identifier" id)) as declarator
        in ("compound_statement" *) as block -> {
    block.vars{id} = {
```

[2] In this regard, the semantics diverts from that of Perl which evaluates "10x" + 1 as 11.

```
        decl −> declarator,
        used −> false,
        minlevel −> −1,
        minblock −> null,
    };
}
```

Individual keys and their associated entries in a dictionary may be deleted through a deletion statement.

Example:

delete *block.vars*{*id*}; *// delete entry for id in block.vars*

Expressions of type dictionary just reference the data structure representing the actual dictionary. Consequently, an assignment of a dictionary causes a dictionary to be shared (shallow copy).

Dictionaries can also be interpreted as sets where the key values represent the members and the associated values are ignored. Following set operators are supported which expect both operands to be of the dictionary type:[3]

"+"	union of two sets
"−"	difference, i.e. taking the first set and removing all keys belonging to the second set
"∗"	intersection of two sets
"^"	symmetric difference, i.e. the result includes keys only which belong either to the first or to the second set but not to both

All set operators work on the keys only. The associated values are taken from the first operand of a set operator, if present, and otherwise taken from the second. In addition, the assignment operators "+=" and "−=" are supported for sets (see 3.19).

2.7 Functions

Function values consist of an optional parameter list, a function body and a closure. A closure of an anonymous function allows a function body to refer to all lexically visible bindings at the point where the function was constructed.

In the following example, the anonymous functions assigned to *incr* and *decr* refer both to the same instance of the local variable *counter* (see 4.6).

```
var counter = 0;
var incr = sub { return ++counter; };
var decr = sub { return --counter; };
```

2.8 Abstract syntax trees

An abstract syntax tree is either a token or a node consisting of an operator represented by a string and an ordered list of subtrees. There exists two kinds of leafs, i.e. token leafs or operator nodes with an empty list of subtrees.

Each node (be it a regular node or a leaf node representing a token) has an associated set of attributes that is organized in the form of a dictionary.

[3] The set operators resemble those of Pascal. Unlike Modula-2 that added an operator for the symmetric difference, however, "/" is not used for this operator. Instead, "^" has been taken because C uses it as operator for XOR which is similar to the symmetric difference.

Tokens provide a textual string in two variants. The token literal value preserves the original text sequence while the token text value provides the processed contents of a token. Example: In case of a string literal "Hello,␣world!", the token literal value returns the entire sequence including the quotes while the token text value returns the contents of the string literal without the quotes. The token literal value and the token text value of a token can be retrieved through *tokenliteral* and *tokentext* standard functions, respectively.

Examples: The following function *traverse* traverses an abstract syntax tree recursively to generate a string representing it. The standard function *isoperator* is used to distinguish between operator and token nodes. An operator node can be examined like a list to retrieve the individual subtrees.

```
sub traverse (node) {
    var result = "";
    if (isoperator(node)) {
        result = "(" & operator(node);
        foreach operand in (node) {
            result &= "␣" & traverse(operand);
        }
        result &= ")";
    } else {
        result = tokenliteral(node);
    }
    return result;
}
```

Alternatively, the individual operands can be indexed:

```
sub traverse (node) {
    var result = "";
    if (isoperator(node)) {
        result = "(" & operator(node);
        var index = 0;
        while (index < len(node)) {
            result &= "␣" & traverse(node[index]);
            ++index;
        }
        result &= ")";
    } else {
        result = tokenliteral(node);
    }
    return result;
}
```

The following attribution rules matches all "compound_statement" operator nodes with an arbitrary number of subnodes. Whenever the rule is executed, the operator node is bound to *block* due to the **as** clause. Abstract syntax trees provide a dictionary that can be used for per-node attributes. The example below initializes the attributes *vars*, *level*, and *up* of *block*:

```
("compound_statement" *) as block -> {
    block.vars = {};
    block.level = 0;
    block.up = null;
}
```

Expressions of type abstract syntax tree reference a node within an abstract syntax tree. Consequently, an assignment of an abstract syntax tree causes the tree to be shared.

2.9 Matching results of regular expressions

If a regular expression matches, a variable can be bound to its result. This result consists of the matched text and a list of captured substrings, if "(...)"-constructs where used within the regular expression. Objects of this type can be interpreted as strings, in this case the whole matched text is delivered. Alternatively, when indices are used, beginning with an index of 0 for the first captured substring, or when objects of this type are put into a list context, the captured substrings are delivered.

Examples: In the following transformation rule (see 9), *dec* is bound to the sequence of digits recognized by the regular expression. Variables bound to matching results can be used like scalars to refer to the matched text.

("decimal_literal" *m*{^\d+$} **as** *dec*) −> ("decimal_literal" {*dec* + 1})

The regular expression of the following example captures one substring which is refered to by *dec*[0].

("decimal_literal" *m*{^\d+(\d)$} **as** *dec*) −> ("decimal_literal" {*dec* & *dec*[0]})

2.10 Control flow graph nodes

Attribution rules can be used to construct a interprocedural control flow graph in conformance to the IFDS framework by [Reps et al. 1995]. In this framework, a program is represented by a set of directed graphs $G = \{G_1, ..., G_n\}$ where each individual graph G_i represents a procedure. Each graph $G_i = (N_i, E_i, n_i)$ consists of nodes N_i, directed edges E_i, and a unique name n_i.

Nodes are created using the *cfg_node* function which takes one or two parameters. The parameters specify a type and/or the associated node of an abstract syntax tree:

var *node1* = *cfg_node*(*node*); *// cfg node associated with an ast node*
var *node2* = *cfg_node*("entry"); *// cfg node created with the type "entry"*
var *cls* = *cfg_node*("close_block", *block*); *// typed cfg node with ast node*

Control flow node types are strings which can be used as identifiers (see 1.9). These types can be queried using the *cfg_type* function and used within conditions of state machines:

/ kill instance when we are leaving our block;*
*node is bound to the current cfg node */*
at *close_block* **where** *node.astnode* == *block* −> **close**

Directed edges are created using the function *cfg_connect* which takes two control flow graph nodes and optionally a third parameter with a label which is a string that can be used as an identifier (see 1.9):

// label "t" represents "true" for conditional paths
cfg_connect(*node1*, *node2*, "t");

Labels can be used within conditions of state machines. Following example matches the tree expression against the abstract syntax node of the current control flow graph node, requires that the node is typed as *conditional_fork* and tests within the **if** constructs for the labels "t" and "f":

("identifier" *varname*) **as** *id* **at** *conditional_fork*
where exists *id.object* && *id.object* == *object*
if *t* **and when** *welldefined* −> *nonnull*
if *t* **and when** *isnull* −> **cut**

if *f* **and when** *welldefined* −> *isnull*
if *f* **and when** *nonnull* −> **cut**

Either all edges from a node have distinct labels or none of the edges are labelled.

Each control flow node has (like syntax tree nodes) a set of attributes that is organized as a dictionary. If a control flow node is associated to a node of an abstract syntax tree, the *astnode* field allows to access it.

The **foreach** loop may be used to iterate through all successor nodes of a control flow graph node. In case of labeled edges, the branches can be examined in the *branch* dictionary.

The set of graphs *G* is represented by the predefined dictionary named *graph*. Per convention this graph shall have the following attributes:

attribute	type	description
entries	dictionary	entry nodes by name
exits	dictionary	exit nodes by name
nodes	list	list of all nodes

Expressions of type control flow graph node reference a node within a control flow graph. Consequently, an assignment of a node causes the node to be shared.

2.11 Streams

Streams exist in two variants, input and output streams. The predefined bindings include *stdin* (standard input), *stdout* (standard output), and *stderr* (standard error output) (see 14). Files can be opened using the *open* function. By default, *open* returns an input stream but files can also be opened for writing using "r" as second parameter. The predefined functions *getline*, *println*, and *prints* allow to read or to write to a stream, respectively. Streams are implicitly closed when the last reference to it is cut.

```
sub main(argv) {
    foreach arg in (argv) {
        var input = open(arg);
        if (input) {
            var line;
            while (defined(line = getline(input))) {
                println(line);
            }
        } else {
            println(stderr, arg, ": could not be opened for reading");
        }
    }
}
```

2.12 Implicit Conversions

In dependence of their context, values may be implicitly converted into another type, if possible. A run time error is raised whenever a conversion cannot be done.

2.12.1 Conversions to string type

Null values are converted to the empty string, integers are converted into their decimal representation, Boolean into "1" or "0", match results into the matched substring, and flow

graph nodes into their type. In case of abstract syntax trees, a conversion delivers either the result of the standard function *operator* in case of operator nodes or *tokentext* otherwise. In case of lists and dictionaries, the size is returned (equivalent to the *len* standard function). Functions without parameter lists are invoked with *args* bound to **null** and their result is converted into string type. An empty string is delivered if **null** is returned.

Streams are converted to their associated file names which were passed to *open*. The predefined standard streams are converted to the respective names, i.e. "stdin", "stdout", and "stderr".

2.12.2 Conversions to integer type

Null values are converted to 0, Boolean values *false* and *true* are converted to 0 and 1, respectively. All other types are converted into a string first and then interpreted as a decimal integer, i.e. an optional minus sign followed by a non-empty sequence of digits “0” to “9” of arbitrary length. If the string does not conform to this format, a run time error is raised.

Streams are converted to 1, if they are in a good state, and 0 otherwise in case of errors or on encountering end of input.

2.12.3 Conversions to Boolean type

Null values are converted to *false*. Non-zero integer values are converted to *true*, zero to *false*. All abstract syntax trees, match results, and flow graph nodes are converted to *true* irrespectively of their value. (Note that even an empty match result is evaluated as *true* to distinguish it from a failed match.)

Streams are converted to *true*, if they are in a good state, and *false* otherwise in case of errors or on encountering end of input.

Values of other types are converted into strings first. The string values "" and "0" are converted to *false*, all other values to *true*.

2.12.4 Conversions to list type

A null value is converted into an empty list. An abstract syntax tree is, if its root is an operator node, converted to a list of subtrees of its root, or, if it is a token, converted to a list of one string-valued element representing the token text. A dictionary is converted into a sorted list of keys. In case of match results, a list of all strings is generated that were captured by the regular subexpressions. (This list is empty if there were no subexpressions.) For control flow graph nodes a list of successor nodes is generated. All other types are converted into a string and from this string a list is contructed with that element.

2.12.5 Conversions to dictionary type

A null value is converted into an empty dictionary. Syntax tree nodes are converted to their set of attributes (see 2.8). All other types are converted to list first (see 2.12.4). Then each element of the list gets converted into a string (see 2.12.1) which is subsequently used as a key. For the associated values the Boolean value *true* is used.

2.12.6 Conversion to abstract syntax tree type

The value is converted to a string and the string value is turned into an abstract syntax tree consisting just of a token whose token text is identical to the string. A null value is treated similarly but an empty string is taken.

2.13 Examining the type of an object

The predefined function *type* takes an arbitrary object as value and returns its type as string:

object type	returned string
null	"null"
Boolean value	"boolean"
integer value	"integer"
string value	"string"
list	"list"
dictionary	"dictionary"
function	"function"
abstract syntax tree	"tree"
matching results	"match_result"
control flow graph node	"flow_graph_node"
input stream	"istream"
output stream	"ostream"

Chapter 3

Expressions

Expressions serve to compute values. They can have side effects. Following table summarizes all operators along with their priorities and associativity:

Operators	Function	Type	Priority	Associativity
identifiers and literals	simple symbols	primary	14	—
d.f	field selection	postfix	14	left to right
d{*f*}	field selection	postfix	14	left to right
a[*i*]	element selection	postfix	14	left to right
exists *d.f*	existance operator	prefix	13	left to right
++*i*	prefix increment	prefix	13	non-associative
i++	postfix increment	postfix	13	non-associative
−−*i*	prefix decrement	prefix	13	non-associative
i−−	postfix decrement	postfix	13	non-associative
(*e*)	grouping	prefix	13	non-associative
[]	list aggregate	prefix	13	non-associative
{}	dictionary aggregate	prefix	13	non-associative
<()>	tree construction	prefix	13	non-associative
f()	function call	postfix	13	left to right
sub {}	function constructor	prefix	13	non-associative
−	negation	prefix	12	non-associative
!	logical negation	prefix	12	non-associative
^	power	infix	11	left to right
∗ **div mod**	multiplicative operators	infix	10	left to right
+ −	additive operators	infix	9	left to right
< > <= >= == =!	comparison operators	infix	8	left to right
x	repetition	infix	7	left to right
&	concatenation	infix	6	left to right
=~	regexp match	infix	5	non-associative
&&	logical and	infix	4	left ro right
| |	logical or	infix	3	left to right
? :	selection	infix	2	non-associative
= &= += −=	assignment	prefix	1	right to left

3.1 Designators

A designator allows to specify an object that can be assigned to. Possible designators are named variables or selected elements of a variable:

⟨designator⟩ ⟶ ⟨identifier⟩
⟶ ⟨designator⟩ “.” ⟨identifier⟩
⟶ ⟨designator⟩ “{” ⟨expression⟩ “}”
⟶ ⟨designator⟩ “[” ⟨expression⟩ “]”

Identifiers refer to one of the bound or locally declared variables. The visibility of a variable name is determined by its lexical scope. In case of name conflicts, the innermost declaration takes precedence.

Objects of dictionary or abstract syntax tree type accept a selector in the form of a key using the “.”-construct for identifiers or the {...}-construct using arbitrary expressions which are converted into a string. Unused keys may only be used if no further selectors are following and if the designator is used at the left side of an assignment or as parameter to the **exists** operator. Otherwise, the use of a non-existing key causes a run-time error to be raised.

Objects of list, match result, or abstract syntax tree type accept a selector in the form of a list index using the [...]-construct using arbitrary expressions which are converted into an integer (see 2.12.2). (In case of abstract syntax trees, an index is only accepted if the root element of the tree is an operator node.) Indices must fall into the range $[0..n-1]$ where n is the length of the list or match result or the number of operands in case of an abstract syntax tree. Individual elements of a match result that represent captured substrings or subtrees selected out of an abstract syntax tree must not be assigned to. Lists can be extended only through the *push* standard function.

3.2 Increment and decrement operators

Designators can be incremented or decremented using prefix or postfix operators:

⟨prefix-increment⟩ ⟶ “++” ⟨designator⟩
⟨prefix-decrement⟩ ⟶ “−−” ⟨designator⟩
⟨postfix-increment⟩ ⟶ ⟨designator⟩ “++”
⟨postfix-decrement⟩ ⟶ ⟨designator⟩ “−−”

In case of the prefix increment and decrement operators, the resulting value of the expression is the value of the designator *after* it has been incremented or decremented. In case of the postfix increment and decrement operators, the resulting value of the expression is the value the designator *before* it has been incremented or decremented.

3.3 List aggregates

List aggregates construct list values:

⟨list-aggregate⟩ ⟶ “[” “]”
⟶ “[” ⟨expression-list⟩ “]”
⟨expression-list⟩ ⟶ ⟨expression⟩
⟶ ⟨expression-list⟩ “,” ⟨expression⟩

Nested list aggregates construct nested data structures. Example:

var *matrix* = [[1 2 3], [4 5 6], [7 8 9]];

3.4 Dictionary aggregates

Dictionary aggregates construct dictionary values:

⟨dictionary-aggregate⟩ ⟶ “{” “}”
⟶ “{” ⟨key-value-pairs⟩ “}”
⟶ “{” ⟨key-value-pairs⟩ “,” “}”
⟨key-value-pairs⟩ ⟶ ⟨key-value-pair⟩
⟶ ⟨key-value-pairs⟩ “,” ⟨key-value-pair⟩
⟨key-value-pair⟩ ⟶ ⟨identifier⟩ “->” ⟨expression⟩
⟶ ⟨string-literal⟩ “->” ⟨expression⟩

Identifiers on the left-hand-side of a “->” operator do not refer to equally-named variables in the current lexical scope but are interpreted literally. In the following example, *member* is a dictionary with one entry where the key is “name” and its associated value “Andreas”:

var *name* = "Andreas";
var *member* = {*name* −> *name*};

Key values which are not of string type are converted implicitly to strings (see 2.12.1). Null values are not permitted as keys.

3.5 Abstract syntax tree constructors

Tree constructors create new abstract syntax trees:

⟨tree-constructor⟩ ⟶ “<(” ⟨string-literal⟩ “)>”
⟶ “<(” ⟨string-literal⟩ ⟨subnodes-constructor⟩ “)>”
⟨subnode-constructor⟩ ⟶ ⟨identifier⟩
⟶ ⟨identifier⟩ “...”
⟶ ⟨tree-expression-constructor⟩
⟶ ⟨tree-expression-constructor⟩ **as** ⟨identifier⟩
⟶ “{” ⟨expression⟩ “}”
⟶ “{” ⟨expression⟩ “}” **as** ⟨identifier⟩

	⟶	"{" ⟨expression⟩ "}" "..."
	⟶	"{" ⟨expression⟩ "}" "..." **as** ⟨identifier⟩
⟨tree-expression-constructor⟩	⟶	"(" ⟨string-literal⟩ ")"
	⟶	"(" ⟨string-literal⟩ ⟨subnodes-constructor⟩ ")"
⟨subnodes-constructor⟩	⟶	⟨subnode-constructor⟩
	⟶	⟨subnodes-constructor⟩ ⟨subnode-constructor⟩

If an expression in curly braces or a bound variable name is followed by the "..." operator, the expression or variable must be a list of subnodes (see 2.5) whose elements are inserted at the corresponding place.

The syntax is close to tree expressions (see 6) except that the topmost parentheses require angle brackets and that all constructs are omitted that are required for matching a tree expression. Alternatively, the predefined functions *make_node* and *make_token* (see 14.1) can be used.

The following example constructs abstract syntax trees that correspond to $3 + 4 * 5$ and $2 * (3 + 4 * 5)$, respectively:

```
var node = <("+"
    { make_token("3") }
    ("*"
        { make_token("4") }
        { make_token("5") }
    )
)>;
var expr = <("*" { make_token("2") } node)>;
```

Alternatively, *make_node* can be used:

```
var node = make_node("+",
    make_token("3"),
    make_node("*",
        make_token("4"),
        make_token("5")
    )
);
var expr = make_node("*", make_token("2"), node);
```

3.6 Function constructors

Function constructors define local functions. Their lexical closure, i.e. the lexical scope at the location of the function construction including the instances of the local variables visible at the time when the function was constructed, remains visible within the body of the function and continues to exist as long as the function can be refered to.

⟨function-constructor⟩	⟶	**sub** ⟨block⟩
	⟶	**sub** ⟨parameter-list⟩ ⟨block⟩
⟨parameter-list⟩	⟶	"(" ")"
	⟶	"(" ⟨identifier-list⟩ ")"

⟨identifier-list⟩ ⟶ ⟨identifier⟩
⟶ ⟨identifier-list⟩ “,” ⟨identifier⟩

If no parameter list is specified, the number of parameters is variable. All actual arguments are then converted into a list and bound to *args*. Otherwise, if a parameter list is given, the number of parameters is fixed and the formal parameters are bound to the corresponding actual parameters.

3.7 Function calls

Functions without parameter lists can be invoked with an arbitrary number of parameters. All parameters are evaluated left to right and their results bound to the elements of the list *args*. This list is empty if no parameters are passed. Otherwise, if the function has a parameter list, the number of actual arguments must match the number of named parameters.

⟨function-call⟩ ⟶ ⟨primary⟩ “(” “)”
⟶ ⟨primary⟩ “(” ⟨expression-list⟩ “)”

The value of a function call is determined by its return value (see 4.5). If a function has no **return** statement, **null** is returned.

Note that functions without parameter lists can also be implicitly invoked through an implicit conversion of a function value to a string (see 2.12.1). In this case, *args* is bound to **null**.

3.8 Primaries

⟨primary⟩ ⟶ ⟨designator⟩
⟶ **exists** ⟨designator⟩
⟶ **exists** “(” ⟨designator⟩ “)”
⟶ ⟨prefix-increment⟩
⟶ ⟨prefix-decrement⟩
⟶ ⟨postfix-increment⟩
⟶ ⟨postfix-decrement⟩
⟶ “(” ⟨assignment⟩ “)”
⟶ ⟨list-aggregate⟩
⟶ ⟨dictionary-aggregate⟩
⟶ ⟨tree-constructor⟩
⟶ ⟨function-call⟩
⟶ ⟨function-constructor⟩
⟶ ⟨cardinal-literal⟩
⟶ ⟨string-literal⟩
⟶ **null**

The **exists** operator returns true if a given key of a designator is used within a dictionary or not. Example:

var *dict* = {*a* −> "a", *b* −> **null**};
var *b1* = **exists** *dict.a*; *// is true*
var *b2* = **exists** *dict.b*; *// is true, even if the value is null*
var *b3* = *defined*(*dict.b*); *// is false as dict.b is null*
var *b4* = **exists** *dict.c*; *// is false*
var *b5* = **exists** *dict.c.d*; *// not permitted as dict.c does not exist*

3.9 Factors

$$\begin{aligned} \langle \text{factor} \rangle &\longrightarrow \langle \text{primary} \rangle \\ &\longrightarrow \text{“}-\text{”}\ \langle \text{primary} \rangle \\ &\longrightarrow \text{“!”}\ \langle \text{primary} \rangle \end{aligned}$$

The unary minus operator “−” converts its operand into an integer and toggles its sign. The logical negation operator “!” converts its operand into a boolean value and negates it.

3.10 Power expressions

$$\begin{aligned} \langle \text{power-expression} \rangle &\longrightarrow \langle \text{factor} \rangle \\ &\longrightarrow \langle \text{power-expression} \rangle\ \text{“\^{}”}\ \langle \text{factor} \rangle \end{aligned}$$

If both operands are dictionaries, the binary operator “^” is interpreted as set operator implementing the symmetric difference (see 2.6).

Otherwise, the binary operator “^” converts its operands a and b to integer values and returns a^b. Note that the value of b must be non-negative and not larger than $2^{31} - 1$.

3.11 Multiplicative expressions

$$\begin{aligned} \langle \text{multiplicative-expression} \rangle &\longrightarrow \langle \text{power-expression} \rangle \\ &\longrightarrow \langle \text{multiplicative-expression} \rangle\ \text{“}*\text{”}\ \langle \text{power-expression} \rangle \\ &\longrightarrow \langle \text{multiplicative-expression} \rangle\ \textbf{div}\ \langle \text{power-expression} \rangle \\ &\longrightarrow \langle \text{multiplicative-expression} \rangle\ \textbf{mod}\ \langle \text{power-expression} \rangle \end{aligned}$$

If both operands are dictionaries, the binary operator “∗” is interpreted as intersection operator for sets (see 2.6).

Otherwise, all multiplicative operators convert their operands into integer values (see 2.12.2) and return an integer value.

The operators **div** and **mod** follow Knuth's definition of these operators if $b \neq 0$ [Knuth 1997]:

$$\begin{aligned} a \textbf{ div } b &:= \left\lfloor \frac{a}{b} \right\rfloor \\ a \textbf{ mod } b &:= a - b \left\lfloor \frac{a}{b} \right\rfloor \end{aligned}$$

A runtime error is raised if the second operand is zero.

3.12 Additive expressions

$$\begin{aligned} \langle \text{additive-expression} \rangle &\longrightarrow \langle \text{multiplicative-expression} \rangle \\ &\longrightarrow \langle \text{additive-expression} \rangle \text{ “+” } \langle \text{multiplicative-expression} \rangle \\ &\longrightarrow \langle \text{additive-expression} \rangle \text{ “−” } \langle \text{multiplicative-expression} \rangle \end{aligned}$$

If both operands are dictionaries, the additive operators are interpreted as set operators (see 2.6), “+” delivering the union, and “−” the difference.

Otherwise, all additive operators convert their operands into integer values (see 2.12.2) and return an integer value.

3.13 Comparison operators

$$\begin{aligned} \langle \text{comparison} \rangle &\longrightarrow \langle \text{additive-expression} \rangle \\ &\longrightarrow \langle \text{comparison} \rangle \text{ “==” } \langle \text{additive-expression} \rangle \\ &\longrightarrow \langle \text{comparison} \rangle \text{ “!=” } \langle \text{additive-expression} \rangle \\ &\longrightarrow \langle \text{comparison} \rangle \text{ “<” } \langle \text{additive-expression} \rangle \\ &\longrightarrow \langle \text{comparison} \rangle \text{ “<=” } \langle \text{additive-expression} \rangle \\ &\longrightarrow \langle \text{comparison} \rangle \text{ “>=” } \langle \text{additive-expression} \rangle \\ &\longrightarrow \langle \text{comparison} \rangle \text{ “>” } \langle \text{additive-expression} \rangle \end{aligned}$$

In case of the comparison operators “==” (equality) and “!=” (inequality), **null** is considered equal to **null** only. Non-scalar types, i.e. lists, dictionaries, functions, abstract syntax trees, matching results of regular expressions, and control flow graph nodes, are considered equal only if they refer to the same object.

If one of the operands of integer type (see 2.3), the other operand, if it is not of integer type, is converted to an integer value (see 2.12.2). The comparison is then performed on two integers.

Otherwise, both operands are converted to string type (see 2.12.1) and the comparison is then performed octet by octet in comparing their numerical values in the local encoding.

3.14 String repetition

⟨repetitive-expression⟩ ⟶ ⟨comparison⟩
⟶ ⟨repetitive-expression⟩ **x** ⟨comparison⟩

The repetition operator "x" converts its left operand into a string (see 2.12.1) and its right operand into an integer (see 2.12.2). If the right operand is zero or negative, the empty string is returned. Otherwise, a string is generated which consists of *n* repetitions of the left operand where *n* is the value of the right operand. Example:

var *fred2* = "fred" **x** 2; */* result is "fredfred" */*
var *spaces* = "␣" **x** 80; */* 80 space characters */*

3.15 String and list concatenation

⟨concatenation-expression⟩ ⟶ ⟨repetitive-expression⟩
⟶ ⟨concatenation-expression⟩ "&" ⟨repetitive-expression⟩

If any of the two operands is of list type, the other operand is, if necessary, converted into a list (see 2.12.4) and a concatenated list is returned consisting of all elements of the first operand, followed by those of the second operand.

Otherwise, both operands are converted to strings (see 2.12.1) and the concatenated string of the first and second operand is returned.

3.16 Matching strings against regular expressions

⟨match-expression⟩ ⟶ ⟨concatenation-expression⟩
⟶ ⟨concatenation-expression⟩ "=~" ⟨regular-expression⟩
⟶ ⟨concatenation-expression⟩ "=~" ⟨concatenation-expression⟩

The matching operator converts its left operand into a string (see 2.12.1) and matches its against the regular expression. If any substring of the left operand matches the regular expression, an object of match result type (see 2.9) is constructed and returned. Otherwise, **null** is returned.

The regular expression can be either specified by a literal (see 1.10.3) or by an arbitrary expression. In the latter case, it is converted to a string (see 2.12.1) and then interpreted as an regular expression like the corresponding literals but without the enclosing braces.

The regular expression engine is aware of the encoding in UTF-8:

- A dot in the regular expression does not match a byte but a Unicode codepoint.
- Captures select sequences of Unicode codepoints.

- Unicode extensions for regular expressions are supported.[1]

Invalid regular expressions cause runtime exceptions.

Example: The following function *ast_summary* takes an abstract syntax tree node as argument and returns a text representation of it. If this representation extends over multiple lines, the intermediate lines are replaced by dots:

```
sub ast_summary (tree) {
    var text = gentext(tree);
    var res;
    if (res = text =~ m{([^\n]*)\n.*\n(.*)}) {
        text = res[0] & "␣...␣" & res[1];
    } elsif (res = text =~ m{([^\n]*)\n}) {
        text = res[0] & "␣...";
    }
    return text;
}
```

3.17 Logical expressions

⟨and-expression⟩	⟶	⟨match-expression⟩
	⟶	⟨and-expression⟩ "&&" ⟨match-expression⟩
⟨or-expression⟩	⟶	⟨and-expression⟩
	⟶	⟨or-expression⟩ "||" ⟨and-expression⟩

The logical operators evaluate the first operand and convert it to Boolean type (see 2.12.3). If in case of the logical and operator "&&" the first operand evaluates to *false*, the logical expression returns *false* without evaluating the right operand (*short circuit evaluation*). Similarly, if in case of the logical or operator "||" the first operand evaluates to *true*, the logical expression returns *true* without evaluating the right operand.

Otherwise, the second operand is evaluated and converted to Boolean type and its value is returned.

3.18 Conditional expressions

⟨conditional⟩	⟶	⟨and-expression⟩
	⟶	⟨and-expression⟩ "?" ⟨and-expression⟩ ":" ⟨and-expression⟩

The conditional operator "?" evaluates its first operand and converts it to Boolean type (see 2.12.3). If it is *true*, the second operand is evaluated and returned, otherwise the third.

[1] See `http://www.pcre.org/current/doc/html/pcre2syntax.html` for details.

3.19 Assignments and expressions

⟨assignment⟩ ⟶ ⟨conditional⟩
⟶ ⟨designator⟩ "=" ⟨assignment⟩
⟶ ⟨designator⟩ "&=" ⟨assignment⟩
⟶ ⟨designator⟩ "+=" ⟨assignment⟩
⟶ ⟨designator⟩ "−=" ⟨assignment⟩
⟨expression⟩ ⟶ ⟨assignment⟩

The regular assignment operator "=" implements reference semantics, i.e. values do not get copied or cloned but references are shared. In case of elementary data types like strings or integers, the effect is not seen as values do not change. Instead, even in case of an increment or decrement operator, new values are represented by new objects. The effect, however, can be observed in case of updatable structured data types like lists and dictionaries. Example:

```
var l1 = [1, 2, 3];
var l2 = l1; // both, l1 and l2, refer to the same list
l1[0] = 4;
// l2[0] == 4;
```

The updating assignment operators "&=", "+=", and "-=", if used on elementary data types (i.e. strings or integers), compute a new value from the old value of the left-hand-side and the given right-hand-side and assign it to the designator. In case of lists and the "&=" operator, the referenced list is extended by the right-hand-side.

The assignment operators "+=" and "-=" are interpreted as corresponding set operators if the designator is a dictionary. In this case, if the right operand is converted into a dictionary, if necessary (see 2.12.5). Example:

```
var set = {};
set += 1; set += 2;
// set: {1 -> true, 2 -> true}
set -= 1;
// set: {2 -> true}
```

Note that *set1 += set2* is a short form of *set1 = set1 + set2*, i.e. a reference to a newly created dictionary is assigned to. The old dictionary remains untouched. This behaviour is different from following loop which extends the existing dictionary:

```
foreach (key, value) in (set2) {
    if (!exists set1{key}) {
        set1{key} = value;
    }
}
```

Chapter 4

Statements

4.1 Blocks

A block groups a sequence of instructions and opens a nested scope, i.e. variables declared within a block are not visible outside the block. Whenever a block is executed, a new instance of it is created with its own set of local variables (see 4.6). Blocks and their variables are implicitly refered to through closures (see 2.7).

$$
\begin{array}{rcl}
\langle \text{block} \rangle & \longrightarrow & \text{“\{” “\}”} \\
& \longrightarrow & \text{“\{” } \langle \text{statements} \rangle \text{ “\}”} \\
\langle \text{statements} \rangle & \longrightarrow & \langle \text{statement} \rangle \\
& \longrightarrow & \langle \text{statements} \rangle \ \langle \text{statement} \rangle \\
\langle \text{statement} \rangle & \longrightarrow & \langle \text{expression} \rangle \text{ “;”} \\
& \longrightarrow & \langle \text{delete-statement} \rangle \text{ “;”} \\
& \longrightarrow & \langle \text{if-statement} \rangle \\
& \longrightarrow & \langle \text{while-statement} \rangle \\
& \longrightarrow & \langle \text{foreach-statement} \rangle \\
& \longrightarrow & \langle \text{return-statement} \rangle \text{ “;”} \\
& \longrightarrow & \langle \text{var-statement} \rangle \text{ “;”}
\end{array}
$$

4.2 Deletion statement

A deletion statement allows to delete a key and its entry in a dictionary. If the given key does not exist, the operation has no effect.

$$
\langle \text{delete-statement} \rangle \longrightarrow \textbf{delete} \ \langle \text{designator} \rangle
$$

4.3 Conditional statement

An **if**-statement allows to execute code conditionally. It consists of a sequence of conditions and associated blocks with an optional **else**-part at the end. The conditions are evaluated and converted to boolean type (see 2.12.3) in the given order until one of them evaluates to *true*. Then the associated block is executed. Otherwise, if none of the conditions evaluates to *true*, the **else**-part is executed, if present.

⟨if-statement⟩ ⟶ **if** “(” ⟨expression⟩ “)” ⟨block⟩
⟶ **if** “(” ⟨expression⟩ “)” ⟨block⟩ **else** ⟨block⟩
⟶ **if** “(” ⟨expression⟩ “)” ⟨block⟩ ⟨elsif-chain⟩
⟶ **if** “(” ⟨expression⟩ “)” ⟨block⟩ ⟨elsif-chain⟩ **else** ⟨block⟩
⟨elsif-chain⟩ ⟶ ⟨elsif-statement⟩
⟶ ⟨elsif-chain⟩ ⟨elsif-statement⟩
⟨elsif-statement⟩ ⟶ **elsif** “(” ⟨expression⟩ “)” ⟨block⟩

4.4 Loop statements

⟨while-statement⟩ ⟶ **while** “(” ⟨expression⟩ “)” ⟨block⟩
⟨foreach-statement⟩ ⟶ **foreach** ⟨identifier⟩ **in** “(” ⟨expression⟩ “)” ⟨block⟩
⟶ **foreach** “(” ⟨identifier⟩ “,” ⟨identifier⟩ “)” **in** “(” ⟨expression⟩ “)” ⟨block⟩

A **while** statement evaluates first the condition, converts it to boolean (see 2.12.3). If the condition is *true* the associated block is executed. This process repeats until the condition evaluates to *false*.

The first variant of the **foreach** statement converts its expression into a list (see 2.12.4) and binds the given identifier to each list member in turn and executes the associated block for it.

The second variant with two variable names expects a dictionary (see 2.6) and binds the given variables to each key and its associated value in the dictionary and executes the associated block for it. The order of the keys is implementation-dependent.

4.5 Return statement

A return statement is valid within functions only (see 2.7 and 5). In its first variant, **null** is returned (see 2.1). The second variant evaluates the given expression and returns its result. In each case, the execution of the current function is terminated and the execution continues after the function call.

⟨return-statement⟩ ⟶ **return**
⟶ **return** ⟨expression⟩

4.6 Variable declarations

Variable declarations bind the given variable name to the value of the given expression or, if no expression has been given, to **null** in the current instance of the local block. Multiple declarations of the same variable name within the same block are not permitted. Variables must not be used before being declared and references to equally-named variables of outer blocks in the same block are prohibited.

⟨var-statement⟩ ⟶ **var** ⟨identifier⟩
⟶ **var** ⟨identifier⟩ "=" ⟨expression⟩

The binding of a variable can be changed through one of the assignment operators in a expression statement (see 3.19).

Chapter 5

Function definitions

5.1 Regular global functions

Functions can be defined at the global level. They are globally visible and their closures are restricted to the set of predefined bindings and the other global functions.

⟨function-definition⟩ ⟶ **sub**⟨identifier⟩ ⟨block⟩
⟶ **sub**⟨identifier⟩ ⟨parameter-list⟩ ⟨block⟩

Parameter passing is handled as for locally constructed functions (see 3.6 and 3.7):

- If a parameter list is given, the number of actual arguments must match the length of the parameter list and each of the named parameters is bound to the corresponding actual argument on invocation.
- Otherwise, if no parameter list is given, any number of actual parameters is permitted, and passed through the list *args*.

Example: The following function computes the greatest common divisor of the two arguments *a* and *b*:

```
sub gcd (a, b) {
    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}
```

Each function returns a value. If the function ends without executing **return** or if the **return**-statement is without value, **null** is returned. Otherwise, the value of the **return** expression is returned (see 4.5).

Local functions can be defined using function constructors (see 3.6) and within state machines (see 8.1).

5.2 Main function

The global *main* function, if it exists, is invoked implicitly within the regular execution order (see 13.1) or is controlling the entire execution in case of a free-standing script (see 13.2). The remaining command line arguments are passed as list to *main*, either through an explicitly named parameter or, if the parameter list has been omitted, through the variable *args* which is locally bound within *main* (see 3.7):

```
/* variant with explicit parameter list */
sub main (argv) {
    foreach arg in (argv) {
        /* process arg... */
    }
}

/* variant without explicit parameter list */
sub main {
    foreach arg in (args) {
        /* process arg... */
    }
}
```

The argument list passed to *main* includes neither the binary of the Astl interpreter nor the name of the script or any other options that are consumed by the runtime environment. The name of the script can be obtained through the predefined binding *cmdname*.

The return value of the *main* function is ignored.

5.3 Other global functions

Named attribution rules (see 7.2), named generating transformation rules (see 9.2), named in-place transformation rules (see 9.3), and named print rules (see 11.2) can be used like regular global functions. They all expect an abstract syntax tree as argument (see 2.8).

Chapter 6

Tree expressions

Tree expressions are used in attribution rules (see 7), state machines (see 8), transformation rules (see 9), and print rules (see 11). A tree expression matches or fails for a particular node in an abstract syntax tree. Whenever a tree expression matches and the associated rule gets executed, the bindings of the tree expression are visible in the lexical scope of the rule. In addition, it is possible to construct trees using a notation that is close to the tree expressions (see 3.5).

6.1 Simple tree expressions

A simple tree expression consists of a operator set (see 12.2) and an arbitrary number of subnodes, specified by nested tree expressions, variable names, variable-length lists using the "..." operator, or by using the wildcard operator "*". Tree expressions containing the operator "..." or the wildcard operator "*" are of variable arity. A tree expression matches a node of an abstract syntax tree if and only if

- the corresponding node is an operator node with an operator that is included in the operator set, and where
- in case of a fixed arity, the number of subnodes matches the arity, and where
- each of the subtree expressions matches the corresponding subtree.

Regular expressions match token nodes only where the regular expression matches the token literal. String literals match tokens only where the token literal is equal to the string literal.

Whenever a named variable is used within ⟨subnode⟩, it is, if defined, compared against the corresponding subtree, or, if undefined, bound to the corresponding subtree. In the former case the variable can be either an node of an abstract syntax tree or an arbitrary value that can be converted into a string. In case of a node of an abstract syntax tree, a comparison covering operators, operands and tokens but not attributes is done recursively. In case of a string, the match succeeds if and only if the corresponding subtree represents a token and if the texts are identical.

Within a list of subnodes, one (and at maximum only one) named variable may be followed by a "..." operator. In this case the variable is either bound to a list consisting of the remaining subnodes or the list of remaining subnodes is matched against the already existing list by comparing the list length and by matching each list element against the corresponding subnode. This list can be empty if there are no additional subnodes.

⟨tree-expression⟩ ⟶ "(" ⟨operator-expression⟩ ")"
⟶ "(" ⟨operator-expression⟩ ⟨subnodes⟩ ")"
⟶ "(" ⟨operator-expression⟩ "*" ")"
⟨subnodes⟩ ⟶ ⟨subnode⟩
⟶ ⟨subnodes⟩ ⟨subnode⟩
⟨subnode⟩ ⟶ ⟨named-tree-expression⟩
⟶ ⟨regular-expression⟩
⟶ ⟨string-literal⟩
⟶ ⟨identifier⟩
⟶ ⟨identifier⟩ "..."
⟨regular-expression⟩ ⟶ ⟨regexp-literal⟩
⟶ ⟨regexp-literal⟩ **as** ⟨identifier⟩

6.2 Operator sets

Operator sets include at least one operator and are defined through operator expressions:

⟨operator-expression⟩ ⟶ ⟨string-literal⟩
⟶ ⟨identifier⟩
⟶ "[" ⟨operators⟩ "]"
⟨operators⟩ ⟶ ⟨operator-term⟩
⟶ ⟨operators⟩ ⟨operator-term⟩
⟨operator-term⟩ ⟶ ⟨string-literal⟩
⟶ ⟨identifier⟩

Named operator sets can be defined through the **opset** clause (see 12.2) which must be declared before it can be used.

6.3 Named tree expressions

Named tree expressions allow to bind a named variable to a matched subnode within the tree expression.

⟨named-tree-expression⟩ ⟶ ⟨tree-expression⟩
⟶ ⟨tree-expression⟩ **as** ⟨identifier⟩

6.4 Contextual tree expressions

Contextual tree expressions allow to specify patterns which must match for one of the parent nodes. If multiple parent nodes match, the innermost is taken for the bindings. If multiple contexts are given in an "**and in**"-chain, the next context is always taken relatively to the previously matched context node. Negated context expressions succeed if no matching parent node is found. No bindings result from negated context expressions and every negated context expression resets the context to the original node.

⟨contextual-tree-expression⟩ ⟶ ⟨named-tree-expression⟩
⟶ ⟨named-tree-expression⟩ **in** ⟨context-expression⟩
⟶ ⟨named-tree-expression⟩ "!" **in** ⟨context-expression⟩
⟨context-expression⟩ ⟶ ⟨context-match⟩
⟶ ⟨context-match⟩ **and in** ⟨context-expression⟩
⟶ ⟨context-match⟩ **and** "!" **in** ⟨context-expression⟩
⟨context-match⟩ ⟶ ⟨named-tree-expression⟩

If a variable named *here* is used within a context expression, it matches only if the original node or one of its parent nodes is equal to the node the variable *here* is bound to. If multiple such special variables are needed in multiple consecutive context expressions, *here1*, *here2* etc. may be used in this order from left to right.

Example: Following tree expression matches all identifiers that are used on the right-hand-side of an assignment expression in C:

```
("identifier" id) in (assignment_operators lhs here) as assignment -> {
    println(id, "␣used␣on␣rhs␣of␣", gentext(assignment));
}
```

Following example matches all cases where an identifier is within the same expression once on the left-hand-side of an expression and then on the right-hand-side:

```
("identifier" id)
        in (assignment_operators here1 rhs) as inner_assignment and
        in (assignment_operators lhs here2) as outer_assignment -> {
    println(id, "␣is␣assigned␣to␣and␣used␣in␣", gentext(outer_assignment));
}
```

6.5 Conditional tree expressions

Conditional tree expressions allow to augment a tree expression with an expression which may use any of the variables bound in the tree expression. The conditional expression is evaluated only if the contextual tree expression matches and before any of the associated rules are executed.

⟨conditional-tree-expression⟩ ⟶ ⟨contextual-tree-expression⟩
⟶ ⟨contextual-tree-expression⟩ **where** ⟨expression⟩

Chapter 7

Attribution rules

7.1 Regular attribution rules

An attribution rule consists of a tree expression and a block. When attribution rules are executed, the abstract syntax tree is traversed depth-first. For each visited node of the abstract syntax tree, all attribution rules are executed whose tree expressions match the visited node.

Attributes rules are executed in pre- or postorder in respect to the traversal of the subnodes. By default, rules are executed in preorder. The keywords **pre** and **post** can be used to specify the order.

When multiple rules match, they are ordered first by arity (in preorder those rules with varying arity are executed first, in postorder rules with varying arity come last) and second by the order of appearance in the source.

⟨attribution-rules⟩ ⟶ **attribution rules** “{” ⟨attributions⟩ “}”
⟨attributions⟩ ⟶ ⟨attribution⟩
⟶ ⟨attributions⟩ ⟨attribution⟩
⟨attribution⟩ ⟶ ⟨conditional-tree-expressions⟩ “->” ⟨block⟩
⟶ ⟨conditional-tree-expressions⟩ “->” **pre** ⟨block⟩
⟶ ⟨conditional-tree-expressions⟩ “->” **post** ⟨block⟩

Example: In the following example we have three attribution rules that focus on operator nodes with the operator "compound_statement". The first rule is executed first. Afterwards, the second rule is executed whenever a compound statement is nested within another compound statement. Then all the attribution rules are executed that match the subnodes. Finally, the third rule is executed last as it is a postfix rule.

```
("compound_statement" *) as block -> {
    block.vars = {};
    block.level = 0;
    block.up = null;
}
("compound_statement" *) as inner_block
        in ("compound_statement" *) as outer_block -> {
    inner_block.level = outer_block.level + 1;
```

```
    inner_block.up = outer_block;
}
("compound_statement" *) as block in ("translation_unit" *) as root -> post {
    foreach (varname, entry) in (block.vars) {
    if (entry.used) {
        if (entry.minblock != block) {
            println(location(entry.decl) & ": variable " & varname &
                " should be moved into " & location(entry.minblock));
        }
    } else {
        println(location(entry.decl) & ": unused variable " & varname);
    }
}
```

7.2 Named attribution rules

Named attribution rules are similar to regular attribution rules but they are not implicitly executed (see 13.1). The given name is bound to a function which executes the associated attribute rule set. The function takes one optional parameter that specifies the root node of the to be traversed tree. Otherwise, if no parameter is given, *root* is taken (see 14). Within this rule set, *root* is bound to the to be attributed abstract syntax tree.

⟨attribution-rules⟩ ⟶ **attribution rules** ⟨identifier⟩ “{” ⟨attributions⟩ “}”

Chapter 8

State machines

8.1 Regular state machines

A state machine consists of a finite number of named states, a set of variables, and a set of rules. Rules can create instances of a state machine, change their states, execute arbitrary code, or stop their execution. State machines without creation rules are called *global* state machines. All other state machines are called *local*.[1]

State machines may be derived from an abstract state machine. In this case, the name of the abstract state machine is to be given after a colon.

⟨state-machine⟩ ⟶ **state machine** ⟨identifier⟩ "(" ⟨states⟩ ")"
"{" ⟨sm-vars⟩ ⟨sm-functions⟩ ⟨sm-rules⟩ "}"
⟶ **state machine** ⟨identifier⟩ ":" ⟨identifier⟩ "(" ⟨states⟩ ")"
"{" ⟨sm-vars⟩ ⟨sm-functions⟩ ⟨sm-rules⟩ "}"

State machine instances are executed during a recursive traverse of the control flow graph (see 13.1). Initially, all global state machines are instantiated. Local state machines are instantiated whenever a **create** clause fires for the first time for a control flow graph node. State machines instances continue to run through the traverse until they are explicitly stopped. If a node of the control flow graph has multiple exits, a copy of a state machine is created for each of the possible paths.

All global state machine instances derived from the same state machine are *related* to each other. All local state machine instances that are derived from the same instance that was created at a particular control flow graph node are also *related* to each other.

Variables declared within a state machine are either *shared* or *private*. Shared variables are shared among all state machine instances that are related to each other. Private variables are never shared among instances of a state machine but kept private for each path taken by a state machine.

⟨sm-vars⟩ ⟶ [⟨sm-var-declarations⟩]
⟨sm-var-declarations⟩ ⟶ ⟨sm-var-declaration⟩
⟶ ⟨sm-var-declarations⟩ ⟨sm-var-declaration⟩

[1] This property conforms to the chosen abstract state machine (see below). However, it is well possible to have a local state machine which is derived from *global* or *global_tracker*. In this case, you will get an interprocedural analysis for a local object.

⟨sm-var-declaration⟩ ⟶ **shared var** ⟨identifier⟩
⟶ **shared var** "=" ⟨expression⟩
⟶ **private var** ⟨identifier⟩
⟶ **private var** ⟨identifier⟩ "=" ⟨expression⟩

Local functions can be declared within a state machine. They inherit the scope from their state machine instance including all shared and private variables.

⟨sm-functions⟩ ⟶ [⟨sm-function-definitions⟩]
⟨sm-function-definitions⟩ ⟶ ⟨function-definition⟩
⟶ ⟨sm-function-definitions⟩ ⟨function-definition⟩

State machines are implicitly executed in the course of a regular execution order (see 13.1) if a control flow graph has been constructed through the attribution rules and if *graph.root* exists and points to a node of the control flow graph. In case of a free-standing execution enviroment (see 13.2), the execution of state machines can be initiated through the *run_state_machines* function.

State machines instances, when executed, have a state and begin initially with the very first state of their list of named states. State machines traverse through the control flow graph. The execution of a state machine instance stops if a control flow graph node has already been visited by a related state machine instance with the same state. For every edge leaving the actual control flow graph node, a new instance of a state machine is constructed which inherits the state and the values of the private variables from its predecessor and continues to share the shared variables with its predecessors. All these new instances are related to each other and to the instance they have been derived from.

While traversing a control flow graph, the rules of a state machine can consider

(1) the node of the abstract syntax tree associated with the control flow graph node including its associated data structures, if existant (see ⟨conditional-tree-expression⟩),

(2) the current node of the control flow graph includes its type and its associated data structures (using the **at** keyword, see ⟨cfg-node-expression⟩),

(3) the label of the edge leaving the current node of the control flow graph (using the **if** keyword, see ⟨cfg-edge-condition⟩), and

(4) the current state (using the **when** keyword, see ⟨sm-state-condition⟩).

Combinations of (3) and (4) can be grouped (see ⟨sm-alternatives⟩) and associated to a combination of (1) and (2) (see ⟨sm-condition⟩):

⟨sm-rule⟩ ⟶ ⟨sm-condition⟩ "->" ⟨sm-block⟩
⟶ ⟨sm-condition⟩ ⟨sm-alternatives⟩
⟶ ⟨sm-condition⟩ "->" **create** ⟨block⟩
⟶ **on close** ⟨sm-handler⟩
⟨sm-condition⟩ ⟶ ⟨conditional-tree-expression⟩
⟶ **at** ⟨cfg-node-expression⟩
⟶ ⟨conditional-tree-expression⟩ **at** ⟨cfg-node-expression⟩

⟨cfg-node-expression⟩ ⟶ ⟨cfg-node-types⟩
⟶ ⟨cfg-node-types⟩ **where** ⟨expression⟩
⟶ "*"
⟶ "*" **where** ⟨expression⟩
⟨cfg-node-types⟩ ⟶ ⟨identifier⟩
⟶ ⟨cfg-node-types⟩ **or** ⟨identifier⟩
⟨sm-alternatives⟩ ⟶ ⟨sm-alternative⟩
⟶ ⟨sm-alternatives⟩ ⟨sm-alternative⟩
⟨sm-alternative⟩ ⟶ **if** ⟨cfg-edge-condition⟩ "->" ⟨sm-block⟩
⟶ **when** ⟨sm-state-condition⟩ "->" ⟨sm-block⟩
⟶ **if** ⟨cfg-edge-condition⟩ **and**
when ⟨sm-state-condition⟩ "->" ⟨sm-block⟩
⟨cfg-edge-condition⟩ ⟶ ⟨identifier⟩
⟶ ⟨cfg-edge-condition⟩ **or** ⟨identifier⟩
⟨sm-state-condition⟩ ⟶ ⟨identifier⟩
⟶ ⟨sm-state-condition⟩ **or** ⟨identifier⟩

If a star is given in a ⟨cfg-node-expression⟩, all control flow graph nodes are matched. This allows to write a rule that fires in all cases:

```
at * -> {
    // ...
}
```

The expressions that are given as conditions to ⟨conditional-tree-expression⟩ or ⟨cfg-node-expression⟩ have access to all shared and private variables of the state machine. In addition, the name *node* is bound to the control flow graph node in the condition of ⟨cfg-node-expression⟩ (but not in the condition of ⟨conditional-tree-expression⟩). (The node of the abstract syntax tree, if defined, can be accessed through *node.astnode*.)

When a rule fires, i.e. if its condition is true, an optional action gets performed and an optional block is executed. If multiple rule conditions are true, the associated actions and blocks are executed in the order of appearance.

⟨sm-block⟩ ⟶ ⟨block⟩
⟶ ⟨sm-action⟩
⟶ ⟨sm-action⟩ ⟨block⟩
⟨sm-action⟩ ⟶ **cache** "(" ⟨expression⟩ ")"
⟶ **close**
⟶ **cut**
⟶ **retract**
⟶ ⟨identifier⟩

The actions **close**, **cut**, and **retract** stop the execution of a state machine instance. In addition, **close** causes the close handler to be invoked, and **retract** undoes the visit of that particular control flow graph node. Whenever a state machine instance stops, the stop is immediate, i.e. no further rules are evaluated.

The **cache** action expects a parameter designating a control flow graph node and checks if the successor node has already been visited. If this has been the case, the state machine continues at the given node with the resulting states of the state machine instance with the same entry state. This allows an interprocedural execution of state machines. For this to work, a function call requires at least two special nodes, one for the call and one for the return position. The function call is linked to the entry node of the function which must be used by all function calls to that function. The exit node of the function is expected to be linked to all return nodes that are paired with a call node to that function. Note that the state machine must explicitly cut off all the return nodes that are not paired with the corresponding call node (see the example below). The close handler, if defined, is invoked if the instance that hit the function first is unable to find a path leading to the exit node. As **cache** considers the actual state when deciding how to go on, the current state must not be changed by any of the rules matching the particular control flow graph node.

If a state machine stops at the end of a path, i.e. if the action **close** is executed or if the current node of the control flow graph has no edges departing from it, all close handlers are invoked, possibly in dependence of the current state.

⟨sm-handler⟩ ⟶ "->" ⟨block⟩
⟶ **when** ⟨sm-state-condition⟩ "->" ⟨block⟩

Within a ⟨cfg-node-expression⟩, the blocks, and close handlers all local variables and functions of the state machine including those inherited from the abstract state machines are visible. In addition, the variables bound by a ⟨conditional-tree-expression⟩ and the following list of bindings is visible within a ⟨cfg-node-expression⟩ or a block:

name	type	description
current_state	string	textual representation of the current state when the current control flow graph node was entered
label	string	string representation of the label of the current edge in the control flow graph; this string is empty if no such label has been defined
node	control flow graph node	refers to the actual control flow graph node; note that *node.astnode*, if it exists, refers to the associated node of the abstract syntax tree

Example: The following state machine is derived from *global_tracker*, an abstract state machine which is itself derived from *global* (see 8.2) and which supports the *err* method that delivers a backtrace documenting the path that lead to an error. This state machine tracks all global interprocedural paths and checks that the invocations of *lock* and *unlock* are properly balanced:

```
state machine lock_checker: global_tracker (unlocked, locked, broken) {
  ("function_call" ("identifier" "lock")) at call
    when unlocked -> locked
    when locked -> broken { err("lock␣possibly␣called␣twice"); }
  ("function_call" ("identifier" "unlock")) at call
    when locked -> unlocked
    when unlocked -> broken { err("unlock␣possibly␣called␣unbalanced"); }

  on close when locked -> {
```

```
        if (!endless_recursion) {
            err("missing␣invocation␣of␣unlock␣at␣end␣of␣path");
        }
    }
    on close −> {
        if (endless_recursion) {
            err("endless␣recursion␣detected");
        }
    }
}
```

8.2 Abstract state machines

Abstract state machines allow to factorize common rule sets out of regular state machines. Abstract state machines are never instantiated. If a state machine (be it regular or abstract) is derived from an abstract state machine, all its variables and rules are inherited.

All outer blocks associated to a rule of a state machine share the same scope. This includes all outer blocks of a regular state machine and all inherited outer blocks. In consequence, a block within an abstract state machine can use variables that are defined in a regular state machine that is derived from the abstract state machine.

⟨abstract-state-machine⟩ ⟶ **abstract state machine** ⟨identifier⟩
“{” ⟨sm-vars⟩ ⟨sm-functions⟩ ⟨sm-rules⟩ “}”
⟶ **abstract state machine** ⟨identifier⟩ “:” ⟨identifier⟩
“{” ⟨sm-vars⟩ ⟨sm-functions⟩ ⟨sm-rules⟩ “}”

Example: The follow abstract state machine serves as common rule set of global state machines. Global state machines do not follow the local links at a function call but follow the extern links to the entry node of the called function. Whenever they return from the exit node of a called function back to the rtn node paired to the call node they cut all paths that are not properly nested. The nesting is checked through the private variable *chain*.

```
abstract state machine global {
    private var chain = {nestlevel −> 0};
    at actual_call
        if local −> cut
        if extern −> cache(node.branch.local) {
            chain = {
                nestlevel −> chain.nestlevel + 1,
                next −> chain,
                callid −> node.callid
            };
        }
    at rtn where chain.nestlevel == 0 || chain.callid != node.callid −> retract
    at rtn where chain.nestlevel > 0 && chain.callid == node.callid −> {
        chain = chain.next;
    }
}
```

Chapter 9

Transformations

Transformation rules allow to generate new abstract syntax trees on base of a given tree, or to modify an abstract syntax tree in-place. Like attribution rules, transformation rules can be collected in named rule sets which are executed only by invoking them.

9.1 Regular transformation rules

Regular transformation rules do not belong to a named rule set and do not operate in-place. Instead, if specified, they allow to generate mutants in conjunction with the printing rules. They are executed in the course of a regular execution order implicitly at the end if print rules exist (see 13.1). If a *main* function is defined (see 5.2), they are processed as soon as the execution of the *main* function is finished. Regular transformation rules are not executed in case of a free-standing script (see 13.2).

In dependence of the command line arguments, a set of files is generated where each file contains a variant of the original source where exactly one transformation rule at one point in the abstract syntax tree was executed. The print rules are free to use the special bindings *location* and *rulename* to insert the information of the transformed location and the rulename, if given, in the generated output (see 14).

If the set of regular transformation rules is executed and none of them matches, a runtime exception is raised ("no matching transformation rule found").

⟨transformation-rules⟩ ⟶ **transformation rules** "{" ⟨transformations⟩ "}"
⟨transformations⟩ ⟶ ⟨transformation⟩
⟶ ⟨transformations⟩ ⟨transformation⟩
⟨transformation⟩ ⟶ [⟨identifier⟩ ":"] ⟨conditional-tree-expression⟩ "->" ⟨transformation-instructions⟩
⟶ [⟨identifier⟩ ":"] ⟨conditional-tree-expression⟩ "->" **pre** ⟨transformation-instructions⟩
⟶ [⟨identifier⟩ ":"] ⟨conditional-tree-expression⟩ "->" **post** ⟨transformation-instructions⟩

The right-hand side of a transformation rule consists of the replacement tree construct and optional blocks that are executed before and after the actual transformation takes place.

⟨transformation-instructions⟩ ⟶ [⟨pre-transformation-block⟩]
⟨subnode-constructor⟩
[⟨post-transformation-block⟩]
⟨pre-transformation-block⟩ ⟶ ⟨block⟩
⟨post-transformation-block⟩ ⟶ ⟨block⟩

Example: Following simple mutating transformation rule replaces an addition by a multiplication and exchanges the two operands:

transformation rules {
("+" *op1 op2*) −> ("∗" *op2 op1*)
}

9.2 Named sets of generating transformation rules

Named sets of generating transformation rules are not implicitly executed unlike the regular transformation rules (see 13.1). Instead the given name is bound to a function that

- expects an abstract syntax tree as parameter or, if no parameter is given, uses *root*,
- generates a clone for each instance where a transformation rule matches a node of the abstract syntax tree,
- applies the matching transformation rule on the matching node, and
- returns a list of the transformed clones.

If no transformation rule matches, an empty list is returned.
Within this rule set, *root* is bound to the to be transformed abstract syntax tree.

⟨named-transformation-rules⟩ ⟶ **transformation rules** ⟨identifier⟩ "{" ⟨transformations⟩ "}"

9.3 Named sets of in-place transformation rules

Named sets of in-place transformation rules are not implicitly executed like the regular transformation rules (see 13.1). Instead the given name is bound to a function that

- expects a to be transformed abstract syntax tree as parameter or, if no parameter is given, uses *root*,
- performs all the transformations in-place,
- suppresses the execution of any possibly conflicting rule, and
- returns the number of transformations performed.

Please note that in-place transformation rules never match the given root node to preserve the referential integrity. Within this rule set, *root* is bound to the to be transformed abstract syntax tree.

⟨named-inplace-transformation-rules⟩ ⟶ **inplace transformation rules** ⟨identifier⟩
"{" ⟨transformations⟩ "}"

Due to the suppression of possible conflicts, in-place transformation functions are usually embedded in a loop that apply the transformations repeatedly until none of the transformation rules fire:

```
inplace transformation rules simplification_rules {
    ("->" op name) -> ("." ("pointer_dereference" op) name)
    ("[]" array index) -> ("pointer_dereference" ("+" array index))
    /* ... */
}

sub simplify(tree) {
    while (simplification_rules() > 0) {}
}
```

Chapter 10

Operator rules

Operator rules, if present, are processed before a program text is generated using the print rules (see 11). Whenever an associativity is expressed by the abstract syntax tree which would get lost by the print rules without parenthesizing everything, an operator node is inserted with "LPAREN" as operator to override the precedence and associativity of the language. The operator used for parenthesizing can be changed by defininining an operator set named *parentheses* which consists of one operator only. This means that, whenever operator rules are employed, an additional rule is required in the print rules that generates the required parenthesis, e.g. by

("LPAREN" *expr*) −> *q*{($*expr*)};

or by defining an operator set first (see 12.2):

opset *parentheses* = ["()"];

// ...

print rules {
    ("()" *expr*) −> *q*{($*expr*)}
}

Operators are to be grouped and sorted by precedence in ascending order. For each group of operators, the associativity has be to specified by one of the keywords **left**, **right**, or **nonassoc**. The keyword is followed by a list of strings representing the corresponding operators of the abstract syntax tree.

Named operator sets can be defined through the **opset** clause (see 12.2) which must be declared before it can be used.

⟨operator-rules⟩	⟶	**operators** "{" ⟨operator-lists⟩ "}"
⟨operator-lists⟩	⟶	⟨operator-list⟩ ";"
	⟶	⟨operator-lists⟩ ⟨operator-list⟩ ";"
⟨operator-list⟩	⟶	**left** ⟨operators⟩
	⟶	**right** ⟨operators⟩
	⟶	**nonassoc** ⟨operators⟩
⟨operators⟩	⟶	⟨operator-term⟩
	⟶	⟨operators⟩ ⟨operator-term⟩
⟨operator-term⟩	⟶	⟨string-literal⟩

⟶ ⟨identifier⟩

The following example shows the operator rules for the C programming language:

```
opset assignment_operators = [
    "=" "+=" "−=" "*=" "/=" "%=" "<<=" ">>=" "&=" "^=" "|="
];

operators {
    left ",";
    right assignment_operators;
    right "conditional_expression";
    left "||";
    left "&&";
    left "|";
    left "^";
    left "&";
    left "==" "!=";
    left "<" ">" "<=" ">=";
    left "<<" ">>";
    left "+" "−";
    left "*" "/" "%";

    right "cast_expression";
    right "pointer_dereference" "address_of"
          "unary+" "unary−" "!" "~" "sizeof"
          "prefix++" "prefix−−";
    left "{}";
    left "postfix++" "postfix−−" "function_call"
          "−>" "." "[]";
}
```

Chapter 11

Print rules

Print rules define how abstract syntax trees are converted into strings.

11.1 Regular print rules

Regular print rules do not belong to a named set of print rules. They are implicitly used by the *gentext* function (see 14) and for generating the results of the regular transformation rules (see 9.1).

Print rules consist of individual rules giving a tree expression (see 6) and a program text literal (see below). The generation of texts starts with the top-level node passed to *gentext* or at the root node of the transformed syntax tree. This node is matched against the available rules. If none matches, a run-time error is raised, specifying the operator and its arity. If multiple rules match, the last one is taken. The program text literal of the matching rule specifies the to be generated text (see 1.10.4). This literal may include placeholders which are interpolated. If one of these placeholders refers to a bound subnode, the print rules are executed recursively to generate the to be interpolated text for the subnode.

⟨print-rules⟩ ⟶ **print rules** “{” ⟨sequence-of-print-rules⟩ “}”
⟨sequence-of-print-rules⟩ ⟶ ⟨print-rule⟩
⟶ ⟨sequence-of-print-rules⟩ ⟨print-rule⟩
⟨print-rule⟩ ⟶ ⟨conditional-tree-expression⟩ “->” ⟨print-expression⟩

A print expression is a program text literal enclosed in “q{” and “}” with to be interpolated placeholders that begin with the character “$”. If a placeholder references a variable bound to a subnode, it is expanded by the recursively generated text of that subnode. In case of expressions, the result is converted into a string and inserted at the corresponding position.

In case of multiline program text literals, indentations are interpreted relatively to each other. Leading and trailing white space is removed unless protected by escape sequences. Tab stops (i.e. ASCII 9) cause the current column to be interpreted by advancing to the next multiply of 8 plus 1 (see 1.3). Whenever leading spaces are required to indent generated text, leading tabs are used whenever possible.

If a placeholder referencing a variable bound to a list of subnodes is followed by the “$...” placeholder, the sequence is replaced by the empty text if the list is empty, by the generated text for the first subnode of the list, if the list has just one subnode, and otherwise expanded by the generated texts for all subnodes with the text between the list

variable and the "$..." operator inserted between each of the generated text sequences for the individual subnodes.

⟨program-text-literal-placeholder⟩ ⟶ "$" ⟨identifier⟩
⟶ "$" "{" ⟨expression⟩ "}"
⟶ "$..."

Examples: The following print rule generates the text for an if-statement in C:

```
("if" expression then_statement else_statement) -> q{
    if ($expression)
        $then_statement
    else
        $else_statement
}
```

This print rules supports compound statements with an arbitrary number of subnodes, each of them representing a statement within the compound statement:

```
("compound_statement" stmt...) -> q{
    {
        $stmt
        $...
    }
}
```

11.2 Named sets of print rules

Named sets of print rules are not implicitly executed. Instead the given identifier is bound to a function that

- expects an abstract syntax tree as parameter or, if no parameter is given, uses *root*,
- recursively applies the print rules as in the case of regular print rules, and
- returns the generated text as *gentext* for the regular print rules.

A run-time error is raised whenever during the traverse a node is found without a matching print rule within the set.

⟨named-print-rules⟩ ⟶ **print rules** ⟨identifier⟩ "{" ⟨sequence-of-print-rules⟩ "}"

Chapter 12

Units

A compilation unit begins with library and import clauses and provides an arbitrary number of rules and definitions:

⟨unit⟩ ⟶ [⟨clauses⟩] [⟨rules⟩]
⟨clauses⟩ ⟶ ⟨clause⟩
⟶ ⟨clauses⟩ ⟨clause⟩
⟨clause⟩ ⟶ ⟨import-clause⟩
⟶ ⟨library-clauses⟩
⟶ ⟨operator-set-clause⟩
⟨import-clause⟩ ⟶ **import** ⟨identifier⟩ ";"
⟨library-clause⟩ ⟶ **library** ⟨string-literal⟩ ";"
⟨rules⟩ ⟶ ⟨rule⟩
⟶ ⟨rules⟩ ⟨rule⟩
⟨rule⟩ ⟶ ⟨function-definition⟩
⟶ ⟨attribution-rules⟩
⟶ ⟨state-machine⟩
⟶ ⟨abstract-state-machine⟩
⟶ ⟨transformation-rules⟩
⟶ ⟨named-transformation-rules⟩
⟶ ⟨named-inplace-transformation-rules⟩
⟶ ⟨operator-rules⟩
⟶ ⟨print-rules⟩
⟶ ⟨named-print-rules⟩

12.1 Libraries

Libraries are source files ending in the suffix ".ast" that can be loaded through an import clause and are looked for in all directories of the library path. The library path consists initially of the current directory only. Library clauses add the given directories to the end of the library path. Libraries must conform to the same syntax, i.e. they are considered as

⟨unit⟩. All their rules and definitions are added to the global pool of rules and definitions. Multiple attempts to import the same library unit are permitted. In this case, just the first import clause is executed.

Multiple operator rules (see 10) are not permitted.

12.2 Operator set clauses

Operator sets can be defined through operator set clauses:

⟨operator-set-clause⟩ ⟶ **opset** ⟨identifier⟩ "=" ⟨operator-expression⟩ ";"

Operator expressions (see 6.2) can be used within tree expressions (see 6) and operator rules (see 10). Operator set clauses must textually precede the tree expressions or operator rules using them.

12.3 Order of appearance

In case of attribution, transformation, and print rules the order of appearance is significant. As all libraries can contribute to the global rule sets, their order is defined as follows:

1. The first source file (usually presented at the command line) is considered first.
2. Within a source file, the order is preserved.
3. Import directives are executed after a source file has been loaded.
4. For each imported unit which has not been loaded yet, the same process starts recursively.

12.4 Regular rule sets

Regular rule sets, i.e. regular attribution rules (see 7.1), regular transformation rules (see 9.1), and regular print rules (see 11.1), can be spread over multiple units and are implicitly united in the order of appearance (see 12.3). This allows, for example, the main program which is loaded first to override a particular print rule by defining a print rule within its unit.

12.5 Global scope

The global scope includes all predefined bindings (see 14), global functions (see 5), named attribution rules (see 7.2), named generating transformation rules (see 9.2), named in-place transformation rules (see 9.3), and named print rules (see 11.2). In case of the function definitions, the order of appearance does not matter as all global functions see all other global functions. The name of state machines do not belong to the global bindings, i.e. a global function name can conflict with the name of a state machine but this is not recommended.

Chapter 13

Execution

The order of execution depends on the actual implementation. But there exist some variants which are supported by the Astl library. Custom implementations can divert from this.

13.1 Standard execution order

This is the most common order of execution where execution starts with the construction of the abstract syntax tree which is not controlled by the Astl program but by its environment. In case of syntax errors, the execution is aborted.

Following steps are performed if a valid and non-empty abstract syntax tree is present:

1. Regular attribution rules (see 7.1) are executed, if they exist.
2. State machines (see 8) are executed, if they exist and a control flow graph has been constructed using attribution rules.
3. The *main* function is executed, if it exists (see 5.2).
4. The regular transformation rules are executed, provided that regular print rules exist (see 11.1). For each transformation, a new abstract syntax tree is generated and printed, in dependence of the command line arguments, either to standard output or into individual output files.

13.2 Free-standing execution order

Alternatively, some implementations support a so-called free-standing execution order where no abstract syntax tree is generated at the beginning. Instead, the entire execution is controlled by the *main* function (see 5.2).

The main function can process its command line arguments, open file arguments, and parse them using the *parse* function (which does not belong to the standard set of predefined functions). Parsing, if successful, delivers an abstract syntax tree. Otherwise, a string is returned with an error message.

The regular attribution rules can be executed using the builtin function *run_attribution_rules*. The execution of state machines can be initiated through *run_state_machines*.

Both is not done automatically to give the opportunity to construct a tree from multiple input sources:

```
sub main(argv) {
    var trees = [];
    foreach arg in (argv) {
        var input = open(arg);
        var res = parse(input);
        if (type(res) == "string") {
            println(stderr, "parse␣of␣", input, "␣failed:\n", res);
        } else {
            push(trees, res);
        }
    }
    var root = <("super−root" { trees }...)>;
    run_attribution_rules(root);
    run_state_machines(root);
}
```

Note that the variable *root* can be redefined in the free-standing execution order while it is read-only in the regular execution model.

Regular transformation rules (see 9.1) will never fire in case of free-standing scripts.

Chapter 14

Predefined bindings

Some bindings are provided at a global scope. None of these values may be redefined at the global level nor modified (i.e. by referencing them on the left side of an assignment) but it is possible to hide them by local declarations. The only exception is the variable *root* in the free-standing execution model (see 13.2) which is initially **null** but may be redefined.

name	type	description
assert	function	aborts the execution if its operand is false
cfg_connect	function	creates a directed edge between the two given control flow nodes; a label can be optionally specified through a third parameter
cfg_node	function	creates a control flow node and expects a control flow node type in form of a string, or an abstract syntax tree, or a node type as first and an abstract syntax tree as second parameter
cfg_type	function	returns the type of a control flow node
chr	function	returns a string consisting of one Unicode codepoint with the value of the first argument converted to an integer; an exception is thrown if the integer value is negative or too large
clone	function	creates a clone of dictionaries and lists (one level deep copy, not done recursively); in all other cases it is similar to a regular assignment
clone_ast	function	creates a clone of an abstract syntax tree whereas all attributes are copied as well as in regular assignments
cmdname	string	basename of the Astl script
copy	function	accepts a target and a source argument which must be both non-null and of the same type; the contents of the source is then copied to the target
defined	function	returns true if the value is non-**null**
env	dictionary	dictionary of environment variables
exit	function	expects an argument that is converted to an integer value which is taken modulo 256 and interpreted as exit code; this function does not return but terminates execution with the given exit code
false	boolean	boolean value of false

Continued on the next page

name	type	description
getline	function	expects an input stream and returns the next line read from the stream without the line terminator; **null** is returned when the input stream has ended or in case of errors
gentext	function	requires the print rules to be available and converts an ast node into a string
graph	dictionary	is predefined as an empty dictionary which is used as data structure for the construction of the control flow graph (see 2.10) and during the execution of the state machines (see 8)
integer	function	converts its argument into an integer value
isoperator	function	its operand must be a node of an abstract syntax tree; true is returned if it is an operator node
isstring	function	returns true if the value is of type string
len	function	returns the length of the list, or the number of keys in a dictionary, or the number of Unicode codepoints within a string (with linear complexity), or the number of captured substrings in a match result
location	function	its operand must be a node of an abstract syntax tree; returns a string representing its source location
make_node	function	creates an abstract syntax operator node where the first parameter specifies the operator and the remaining arguments the subnodes; lists of subnodes are supported and get expanded
make_token	function	returns an abstract syntax tree leaf node consisting of a token; the token text is derived from the first argument which is converted to a string
open	function	expects one or two arguments, the first specifying a filename, the second a mode where "r" (for reading) and "w" (for writing) are accepted; the file name is opened in respect to the given mode (for reading if just the filename is given) and a corresponding stream is returned, if successful; **null** is returned in case of errors
operator	function	its operand must be an operator node of an abstract syntax tree; returns a string representing its operator
ord	function	returns the first Unicode codepoint value of the first argument which is converted to a string
pop	function	removes and returns the first element of a list
push	function	its first operand is a list which is extended by appending all the remaining arguments to it
println	function	prints all arguments and a line terminator to standard output
prints	function	prints all arguments to standard output
root	tree	points to the root node of the abstract syntax tree preloaded in course of the regular execution model (see 13.1); the free-standing execution model (see 13.2) sets *root* initially to **null** but allows it to be redefined; within named attribution rules (see 7.2) and transformation rules (see 9.2 and 9.3) *root* is locally bound to the abstract syntax tree passed to the corresponding function
stdin	istream	standard input stream

Continued on the next page

name	type	description
stdout	ostream	standard output stream
stderr	ostream	standard error output stream
string	function	converts its argument into a string value
tokenliteral	function	returns the literal text of a token; this usually includes the delimiters, e.g. the string quotes in case of a string
tokentext	function	returns the processed text of a token; this usually does not include the delimiters or the escape characters
true	boolean	boolean value of true
type	function	returns the type of its argument as a string (see 2.13)

Whenever print rules are executed for a transformed program text, some additional predefined bindings are added:

name	type	description
location	string	the string representation of the location of the transformed program text
rulename	string	the name of the applied transformation rule, if defined and **null** otherwise.

Index

Bibliography

Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, section 1.2.4, pages 39–40. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: http://doi.acm.org/10.1145/199448.199462.