gazprea

cmput415

Sep 13, 2022

CONTENTS

Gazprea is derived from a language originally designed at the IBM Hardware Acceleration Laboratory in Markham, ON.

CHAPTER

ONE

CHANGE LOG

- **2020/11/18 19:00**
  - Clarified when memory should be freed. (*Memory Management*)
  - Clarified that typedef type symbol names do not conflict with variable or subroutine symbol names. (*Typedef*, Section 13)
  - Clarified that `null` and `identity` cannot be cast. (*Null and Identity*, Section 11.7)
  - Clarified that "two sided" promotion may occur with tuples. (*Tuple to Tuple*, Section 12.4)
  - Clarified that `return` statements must have values compatible with the type in the `returns` clause. (*Return*, Section 15.7)
  - Clarified that procedures without `returns` clauses do not require explicit `return` statements. (*Return*, Section 15.7)
  - Clarified that scalar `character` values can still take part in the concatenation (`||`) operation with a `string` or vector with type `character`. (*Operations*, Section 8.2.5)
  - Rewrote several sections of `string`:
    - Clarified differences in type description. (*String*, Section 8.8)
    - Clarified declaration methods. (*Declaration*, Section 8.8.1)
    - Clarified concatentation of characters onto strings. (*Operations*, Section 8.8.5)
  - Clarified that whitespace cannot be in a real literal when in the input stream. (*Input Format*, Section 5.2.1)
  - Fixed example of misreading a `boolean` from the `std_input`. (*Input Format*, Section 5.2.1).
  - Added restrictions on input stream rewinding. (*Stream State*, Section 19.4; *Error Handling*, Section 5.2.2)
  - Moved and expanded description of stream rewinding. (*Stream State*, Section 19.4; *Error Handling*, Section 5.2.2)
  - Removed "and subsequent reads" due to ever-expanding nature of `std_input`. (*Error Handling*, Section 5.2.2)
  - Trimmed information in description of `stream_state` which was duplicating the description of streams. (*Stream State*, Section 19.4)
- **2020/10/22 16:20**
  - Cleaned up latex artifacts messing up a code block. (*Return*, Section 15.7)
  - Changed `pythag` function so that the exponent is a real. (*Syntax*, Section 16.1)
  - Fixed latex artifact where `character` literal for `'` was `\'` not `\'`. (*Literals and Escape Sequences*, Section 8.2.4)

  - Removed usages of `std_output` as an assignable value. (*Main*, Section 17.3; *Tuple to Tuple*, Section 12.4; *Operations*, Section 8.8.5; *Scalar to Vector or Matrix*, Section 12.2)
  - Remove mention of `matrix` keyword that no longer exists. (*Declaration*, Section 8.9.1)
  - Clarify lack of `= <stmt>;` format for procedures. (*Syntax*, Section 17.1)
  - Fix usage of `if ... fi` that is not used in *Gazprea*. (*Operations*, Section 8.5.6).
  - Remove usage of `0.0f` which is C syntax. (*Assignment Statements*, Section 15.1)
  - Fix malformed types for vectors and matrices where the size was not attached to the type. (*Operations*, Section 8.7.5; *Declaration*, Section 8.9.1)
  - Clarified format of `real` literals without scientific notation. (*Literals*, Section 8.4.4)
  - Clarified format of `real` literals with scientific notation. (*Literals*, Section 8.4.4)

- **2020/09/01 15:00**
  - Initial release for Fall 2020

CHAPTER

TWO

KEYWORDS

Gazprea has a number of built in keywords that are reserved and should not be used by a programmer.

- and
- as
- boolean
- break
- by
- call
- character
- columns
- const
- continue
- else
- false
- function
- identity
- if
- in
- integer
- interval
- length
- loop
- not
- null
- or
- procedure
- real
- return

- returns
- reverse
- rows
- std_input
- std_output
- stream_state
- string
- true
- tuple
- typedef
- var
- while
- xor

CHAPTER

THREE

IDENTIFIERS

Identifiers in *Gazprea* must start with either an underscore or a letter (upper or lower cased). Subsequent characters can be an underscore, letter (upper or lower case), or number. An identifier may not be any of *Gazprea*'s keywords. Here are some valid identifiers in *Gazprea*:

```
hello
h3ll0
_h3LL0
_Hi
Hi
_3
```

The following are some examples of invalid identifiers. They begin with a number, contain invalid characters, or are a keyword:

```
3d
in
a-bad-variable-name
no@twitter
we.don't.like.punctuation
```

Gazprea imposes no restrictions on the length of identifiers.

CHAPTER

FOUR

COMMENTS

Gazprea supports *C99* style comments.

Single line comments are made using `//`. Anything on the line after the two adjacent forward slashes is ignored. For example:

```
integer x = 2 * 3;  // This is ignored
```

Multi-line block comments are made using **/*** and ***/**. The start of a block comment is marked using **/***, and the end of the block comment is the **first** occurrence of the sequence of characters ***/**. For example:

```
/* This is a block comment. I can span as many lines as we want, and
   only ends when the closing sequence is encountered.
 */
integer x = 2 * 3;  /* Block comments can also be on a single line */
```

Block comments cannot be nested because the comment finishes when it reaches the first closing sequence. For example, this is invalid:

```
/* A comment /* A nested comment */ */
```

CHAPTER

FIVE

STREAMS

Gazprea has two streams: `std_output` and `std_input`, which are used for outputting to stdout and reading from stdin respectively.

5.1 Output Stream

Output streams use the following syntax:

```
<exp> -> std_output;
```

5.1.1 Output Format

Values of the following base types are treated as follows when sent to an output stream:

- *Character*: The character is printed.
- *Integer*: Converted to a string representation, and then printed.
- *Real*: Converted to a string representation, and then printed. This is the same behaviour as the %g specifier in printf.
- *Boolean*: Print T for true, and F for false.

Vectors print their contents according to the rules above, with square braces surrounding its elements and with spaces only *between* values. For example:

```
integer[*] v = 1..3;
v -> std_output;
```

prints the following:

```
[1 2 3]
```

Strings print their contents as a contiguous sequence of characters. For example:

```
string str = "Hello, World!";
str -> std_output;
```

prints the following:

```
Hello, World!
```

Matrices print like a vector of vectors. For example:

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]] -> std_output;
```

prints the following:

```
[[1 2 3] [4 5 6] [7 8 9]]
```

No other type may be sent to a stream. For instance functions, procedures, and tuples cannot be sent to streams.

Note that there is **no automatic new line or spaces printed.** To print a new line, a user must explicitly print the new line or space character. For example:

```
'\n' -> std_output;
' ' -> std_output;
```

5.1.2 Null and Identity

If `null` or `identity` is sent to a stream then the result is a null or identity character being printed.

5.2 Input Stream

Input streams use the following syntax:

```
<l-value> <- std_input;
```

An l-value may be anything that can appear on the left hand side of an assignment statement. Consider reading the discussion of an l-value here.

Input streams may only work on the following base types:

- `character`: Reads a single character from stdin. Note that there can be no *error state* for reading characters.
- `integer`: Reads an integer from stdin. If an integer could not be read, an *error state* is set on this stream.
- `real`: Reads a real from stdin. If a real could not be read, an *error state* is set on this stream.
- `boolean`: Reads a boolean from stdin. If a boolean value could not be read, an *error state* is set on this stream.

5.2.1 Input Format

Whitespace will separate values in stdin, but take note that a whitespace character *can* also be read from stdin and assigned to a character variable.

A `character` from stdin is the first byte that can be read from the stream. If the end of the stream is encountered, then `-1` is returned.

An `integer` from stdin can take any legal format described in the *integer literal* section. It may also be proceeded by a single negative or positive sign.

A `real` input from stdin can take any legal format described in the *real literal* section with the exception that no whitespace may be present. It may also be proceeded by a single negative or positive sign.

A `boolean` input from stdin is either `T` or `F`.

When reading a value, if any other input were to be in the stream during the read then an *error state* is set. For example, the following program:

```
boolean b;
b <- std_input;
```

With the standard input stream containing this:

```
Ta
```

An *error state* would be set on the stream.

5.2.2 Error Handling

When reading `boolean`, `integer`, and `real` from stdin, it is possible that the end of the stream or an error is encountered. In order to handle these situations *Gazprea* provides a built in procedure that is implicitly defined in every file: `stream_state` (see *Stream State*).

Reading a `character` can never cause an error. The character will either be successfully read or the end of the stream will be reached and `-1` will be returned on this read.

Otherwise, when an error or the end of the stream is encountered, the value returned is the type-appropriate `null`.

Only when an error is encountered, the stream must be rewound to where it was when the read started. This rewind includes any whitespace that may have been skipped to in order to encounter the next token. This is because the subsequent read may be for a `character` which should successfully read the rewound whitespace. For example, with this program:

```
integer i;
character c;
i <- std_input;
i <- std_input;
c <- std_input;

i -> std_output;
c -> std_output;
'$' ->
```

and the input stream (with * representing `' '`):

```
5****10a
```

the output should be:

```
0 $
```

and the remaining input stream should be:

```
***10a
```

Because this means you may have to skip a potentially nearly-infinite amount of whitespace this specification, this specification limits the size of the "rewind buffer" to 1024 characters. Therefore, no read from `std_input` will require more than 1KB of characters from the current stream position to the end of the next token. This means that you will only ever need to maintain at most 1024 characters in a buffer (1025 if a `'\0'` character is required). If more characters than that are required to be read then the runtime should emit an error.

This table summarizes an input stream's possible error states after a read of a particular data type.

Type	Situation	Return	`stream_state`
Boolean	error	`false`	1
	end of stream	`false`	2
Character	error	N/A	N/A
	end of stream	`-1`	0
Integer	error	`0`	1
	end of stream	`0`	2
Real	error	`0.0`	1
	end of stream	`0.0`	2

CHAPTER

SIX

DECLARATIONS

Variables must be declared before they are used. Aside from a few *special cases*, declarations have the following formats:

```
<qualifier> <type> <identifier> = <expression>;
<qualifier> <type> <identifier>;
```

Both declarations are creating a variable with an *identifier* of `<identifier>`, with *type* `<type>`, and optionally a *type qualifier* of `<qualifier>`.

The first declaration explicitly initializes the value of the new variable with the value of `<expression>`.

In *Gazprea* all variables must be initialized in a well defined manner in order to ensure functional purity. If the variables were not initialized to a known value their initial value might change depending on when the program is run. Therefore, the second declaration is equivalent to:

```
<qualifier> <type> <identifier> = null;
```

For simplicity *Gazprea* assumes that declarations can only appear at the beginning of a block. For instance this would not be legal in *Gazprea*:

```
integer i = 10;
if (blah) {
  i = i + 1;
  real i = 0;  // Illegal placement of a declaration.
}
```

because the declaration of the real version of `i` does not occur at the start of the block.

The following declaration placement is legal:

```
integer i = 10;
if (blah) {
  real i = 0;  // At the start of the block. All good.
  i = i + 1;
}
```

The declaration of a variable happens after initialization. Thus it is illegal to refer to a variable within its own initialization statement.

```
/* All of these declarations are illegal, they would result in garbage
   values. */
integer i = i;
integer v[10] = v[0] * 2;
```

An error message should be raised about the use of undeclared variables in these cases. If a variable of the same name is declared in an enclosing scope, then it is legal to use that in the initialization of a variable with the same name. For instance:

```
integer x = 7;
if (true) {
  integer y = x;  /* y gets a value of 7 */
  real x = x; /* Refers to the enclosing scope's 'x', so this is legal */

  /* Now 'x' refers to the real version, with a value of 7.0 */
}
```

6.1 Special cases

Special cases of declarations are covered in their respective sections.

1. *Vectors*
2. *Matrices*
3. *Tuples*
4. *Globals*
5. *Functions*
6. *Procedures*

CHAPTER

SEVEN

TYPE QUALIFIERS

Gazprea has two type qualifiers: `const` and `var`. These qualifers can prefix a type to specify its mutability or entirely replace the type to request that it be inferred. Mutability refers to a values ability to be an r-value or l-value. The two qualifiers cannot be combined as they are mutually exclusive.

7.1 Const

A `const` value is immutable and therefore cannot be an l-value but can be an r-value. For example:

```
const integer i;
```

Because a `const` value is not an l-value, it cannot be passed to a `var` argument in a `procedure`.

7.2 Var

A `var` value is mutable and therefore can be an l-value or r-value. For example:

```
var integer i;
```

Note that var is the default *Gazprea* behaviour and is essentially a no-op unless it is entirely replacing the type.

7.3 Type Inference Using Qualifiers

Type qualifiers may be used in place of a type, in which case the real type must be inferred. A variable declared in this manner must be **immediately initialised** to enable inference. For example:

```
var i = 1; // integer
const i = 1; // integer
var r = 1.0; // real
const c = 'a'; // character
var t = (1, 2, 'a', [1, 2, 3]); // tuple(integer, integer, character, integer[3])
const v = ['a', 'b', 'c', 'd']; // character[4]
```

See *Type Inference* for a larger description of type inference, this section only provides the syntax for inference using `const` and `var`.

CHAPTER

EIGHT

TYPES

8.1 Boolean

A `boolean` is either `true` or `false`. A `boolean` can be represented by an `i1` in *LLVM IR*.

8.1.1 Declaration

A `boolean` value is declared with the keyword `boolean`.

8.1.2 Null

`null` is `false` for `boolean`.

8.1.3 Identity

`identity` is `true` for `boolean`.

8.1.4 Literals

The following are the only two valid `boolean` literals:

- `true`
- `false`

8.1.5 Operations

The following operations are defined on `boolean` values. In all of the usage examples `bool-expr` means some `boolean` yielding expression.

Operation	Symbol	Usage	Associativity
parenthesis	`()`	`(bool-expr)`	N/A
negation	`not`	`not bool-expr`	right
logical or	`or`	`bool-expr or bool-expr`	left
logical xor	`xor`	`bool-expr xor bool-expr`	left
logical and	`and`	`bool-expr and bool-expr`	left
equals	`==`	`bool-expr == bool-expr`	left
not equals	`!=`	`bool-expr != bool-expr`	left

Unlike many languages the `and` and `or` operators do not short circuit evaluation. Therefore, both the left hand side and right hand side of an expression must always be evaluated.

This table specifies `boolean` operator precedence. Operators without lines between them have the same level of precedence.

Precedence	Operation
HIGHER	`not`
	`==` `!=`
	`and`
LOWER	`or` `xor`

8.1.6 Type Casting and Type Promotion

To see the types that `boolean` may be cast and/or promoted to, see the sections on *Type Casting* and *Type Promotion* respectively.

8.2 Character

A `characters` is a signed 8-bit value. A `character` can be represented by an `i8` in *LLVM IR*.

8.2.1 Declaration

A `character` value is declared with the keyword `character`.

8.2.2 Null

`null` is ASCII `NUL` (`'\0'`, `0x00`) for `character`.

8.2.3 Identity

`identity` is ASCII `SOH` (`0x01`) for characters. This choice allows the casting of a `character` to an `integer` to yield the `integer` `identity`.

8.2.4 Literals and Escape Sequences

A `character` literal is written in the same manner as *C99*: a single character enclosed in single quotes. You may not use literal newlines. For example:

```
'a'
'b'
'A'
'1'
'.'
'*'
```

As in *C99*, *Gazprea* supports character escape sequences for common characters. For example:

```
'\0'
'\n'
```

The following escape sequences are supported by *Gazprea*:

Description	Escape Sequence	Value (Hex)
Null	`\0`	`0x00`
Bell	`\a`	`0x07`
Backspace	`\b`	`0x08`
Tab	`\t`	`0x09`
Line Feed	`\n`	`0x0A`
Carriage Return	`\r`	`0x0D`
Quotation Mark	`\"`	`0x22`
Apostrophe	`\'`	`0x27`
Backslash	`\\`	`0x5C`

8.2.5 Operations

There are no operations defined between scalar values with type``character``. To operate on a `character` it must first be cast to either a `boolean`, `integer`, or `real`.

However, scalar values with type `character` may still be concatenated onto values with type `string` or vectors with type `character`

8.2.6 Type Casting and Type Promotion

To see the types that `character` may be cast and/or promoted to, see the sections on *Type Casting* and *Type Promotion* respectively.

8.3 Integer

An `integer` is a signed 32-bit value. An `integer` can be represented by an `i32` in *LLVM IR*.

8.3.1 Declaration

A `integer` value is declared with the keyword `integer`.

8.3.2 Null

`null` is `0` for `integer`.

8.3.3 Identity

`identity` is 1 for `integer`.

8.3.4 Literals

An `integer` literal is specified in base 10. For example:

```
1234
2
0
```

8.3.5 Operations

The following operations are defined between `integer` values. In all of the usage examples `int-expr` means some `integer` yielding expression.

Class	Operation	Symbol	Usage	Associativity
Grouping	parentheses	`()`	`(int-expr)`	N/A
Arithmetic	addition	`+`	`int-expr + int-expr`	left
	subtraction	`-`	`int-expr - int-expr`	left
	multiplication	`*`	`int-expr * int-expr`	left
	division	`/`	`int-expr / int-expr`	left
	remainder	`%`	`int-expr % int-expr`	left
	exponentiation	`^`	`int-expr ^ int-expr`	right
	unary negation	`-`	`- int-expr`	right
	unary plus (no-op)	`+`	`+ int-expr`	right
Comparison	less than	`<`	`int-expr < int-expr`	left
	greater than	`>`	`int-expr > int-expr`	left
	less than or equal to	`<=`	`int-expr <= int-expr`	left
	greater than or equal to	`>=`	`int-expr >= int-expr`	left
	equals	`==`	`int-expr == int-expr`	left
	not equals	`!=`	`int-expr != int-expr`	left

Unary negation produces the additive inverse of the `integer` expression. Unary plus always produces the same result as the `integer` expression it is applied to. Remainder mirrors the behaviour of remainder in *C99*.

This table specifies `integer` operator precedence. Operators without lines between them have the same level of precedence. Note that parentheses are not included in this list because they are used to override precedence and create new atoms in an expression.

Precedence	Operations
HIGHER	`unary +` `unary -`
	`^`

	`*`
	`/`
	`%`

	`+` `-`

	`<`
	`>`
	`<=`
	`>=`

LOWER	`==` `!=`

8.3.6 Type Casting and Type Promotion

To see the types that `integer` may be cast and/or promoted to, see the sections on *Type Casting* and *Type Promotion* respectively.

8.4 Real

A `real` is an IEEE 754 32-bit floating point value. A `real` can be represented by a `float` in *LLVM IR*.

8.4.1 Declaration

A `real` value is declared with the keyword `real`.

8.4.2 null

`null` is `0.0` for `real`.

8.4.3 identity

`identity` is `1.0` for `real`.

8.4.4 Literals

A `real` literal can be specified in several ways. A leading zero is not necessary and can be inferred from a leading decimal point. Likewise, a trailing zero is not necessary and can be inferred from a trailing decimal point. However, at least one digit must be present in order to be parsed. For example:

```
42.0
42.
4.2
```

(continues on next page)

(continued from previous page)

```
0.42
.42
.  // Illegal.
```

A `real` literal can also be created by any valid `real` or `integer` literal followed by scientific notation indicated by the letter `e` and another valid `integer` literal. Scientific notation multiplies the first literal by 10^x. For example, $4.2\text{e}{-3} = 4.2 \times 10^{-3}$. For example:

```
4.2e-1
4.2e+9
4.2e5
42.e+37
.42e-7
42e6
```

8.4.5 Operations

Floating point operations and precedence are equivalent to *integer operation and precedence*.

Operations on real numbers should adhere to the IEEE 754 spec with regards to the representation of not-a-number(NaNs), infiity(infs), and zeros. A signaling NaN should cause a runtime error. Floating point errors and semantics can be guaranteed by using the LLVM IR constrained floating point intrinsics. The `round.towardzero` rounding mode should be chosen along with the `fpexcept.strict` exception behaviour.

For more information on why this is necessary, look into the default LLVM IR floating point environment.

8.4.6 Type Casting and Type Promotion

To see the types that `real` may be cast and/or promoted to, see the sections on *Type Casting* and *Type Promotion* respectively.

8.5 Tuple

A `tuple` is a way of grouping multiple values with potentially different types into one type. All types may be stored within tuples except *streams* and other tuples.

8.5.1 Declaration

A `tuple` value is declared with the keyword `tuple` followed by a parentheses-surrounded, comma-separated list of types. The list must contain *at least two types*. Tuples are *mutable*. For example:

```
tuple(integer, real, integer[10]) t1;
tuple(character, real, string[256], real) t2;
```

The fields of a `tuple` may also be named. For example:

```
tuple(integer, real r, integer[10]) t3;
tuple(character mode, real, string[256] id, real) t4;
```

Here, `t3` has a named `real` field named `r` and `t4` has a named `character` field named `mode` and another named `string` field named `id`.

The number of fields in a `tuple` must be known at compile time. The only exception is when a *variable is declared without a type using var or const*. In this case, the variable must be initialised immediately with a literal whose type is known at compile time.

8.5.2 Access

The elements in a `tuple` are accessed using dot notation. Dot notation can only be applied to `tuple` variables and *not* `tuple` literals. Therefore, dot notation is an identifier followed by a period and then either a literal `integer` or a field name. Field indices *start at one*, not zero. For example:

```
t1.1
t2.4
t3.r
t4.mode
```

8.5.3 Null

`null` is every field assigned their type-appropriate `null` for `tuple`.

8.5.4 Identity

`identity` is every field assigned their type-appropriate `identity` for `tuple`.

8.5.5 Literals

A `tuple` literal is constructed by grouping values together between parentheses in a comma separated list. For example:

```
tuple(integer, string[5], integer[3])  my_tuple = (x, "hello", [1, 2, 3]);
var my_tuple = (x, "hello", [1, 2, 3]);
const my_tuple = (x, "hello", [1, 2, 3]);
tuple(integer, real r, integer[10]) tuple_var = (1, 2.1, [i in 1..10 | i]);
```

8.5.6 Operations

The following operations are defined on `tuple` values. In all of the usage examples `tuple-expr` means some `tuple` yielding expression, while `int_lit` is an `integer` literal as defined in *Integer Literals* and `id` is an identifier as defined in *Identifiers*.

Class	Operation	Symbol	Usage	Associativity
Access	dot	`.`	`tuple-expr.int_lit` `tuple-expr.id`	left
Comparison	equals	`==`	`tuple-expr == tuple-expr`	left
	not equals	`!=`	`tuple-expr != tuple-expr`	left

Note that in the above table `tuple-expr` may refer to only a variable for access. Accessing a literal could be replaced immediately with the scalar inside the `tuple` literal. However `tuple-expr` may refer to a literal in comparison operations to enable shorthand like this:

```
if ((a, b) == (c, d)) { }
```

Comparisons are performed pairwise, therefore only `tuple` values of the same type can be compared. This table describes how the comparisons are completed, where t1 and t2 are `tuple` yielding expressions including literals:

Operation	Meaning
`t1 == t2`	`t1.1 == t2.1 and ... and t1.n == t2.n`
`t1 != t2`	`t1.1 != t2.1 or ... or t1.n != t2.n`

8.5.7 Type Casting and Type Promotion

To see the types that `tuple` may be cast and/or promoted to, see the sections on *Type Casting* and *Type Promotion* respectively.

8.6 Interval

An `interval` is used to represent ranges of values. *Gazprea* only has support for an `integer interval`.

8.6.1 Declaration

An `interval` is declared with the keyword `interval`. Only the `integer` base type for intervals is supported by *Gazprea*. For example:

```
integer interval iv;
```

8.6.2 Null

`null` is defined as `null..null`. For the `integer interval` this is `0..0`.

8.6.3 Identity

`identity` is defined as `identity..identity`. For the `integer interval` this is `1..1`.

8.6.4 Literals

An `interval` literal is a range expression, created using two inclusive bounds and the range operator (`..`). For example, a range from one to ten, including the endpoints:

```
integer interval i = 1..10;
```

8.6.5 Operations

Operations on intervals should follow the standard rules of interval arithmetic. In each case integer operations should be used, for instance interval division should use integer division. For another explanation see this website under the heading of "How the operations work".

In the following table `ivl-expr` means any expression that yields an `interval` value and `int-expr` means any `integer` yielding expression.

Class	Operation	Symbol	Usage	Associativity
Arithmetic	addition	+	`ivl-expr + ivl-expr`	left
	subtraction	-	`ivl-expr - ivl-expr`	left
	multiplication	*	`ivl-expr * ivl-expr`	left
	unary negation	-	`- ivl-expr`	right
	unary plus (no-op)	+	`+ ivl-expr`	right
Comparison	equals	==	`ivl-expr == ivl-expr`	left
	not equals	!=	`ivl-expr != ivl-expr`	left
Vector	vector creation	by	`ivl-expr by int-expr`	left

Note there is no division operation for intervals in Gazprea.

Regarding the semantics of some of the operators:

- Comparison checks the bounds of each `interval`.
- Range upper bounds must greater than or equal to the lower bound.
- Both bounds must be `integer` valued.

The precedence and associativity follows that of for the operators defined in the above table, with the addition of the `by` and `..` operators, in the following table. The `.` and `[]` operators are included for clarification but for the full table see .

Precedence	Operations
HIGHER	`.` `[]`
	`..`
	arithmetic ops
	`by`
LOWER	comparison ops

This means that `by` is the lowest priority and so last binding operator, therefore each side of the expression will be evaluated before evaluating the `by` operator. As well, `..` is the highest priority and first binding operator, excluding the `.` and `[]` operators which create atoms, and will bind to atoms before other operators. For example:

```
1 .. 10 by 3
a[1] .. b.3 by 1 + 2
```

Should be parsed as:

```
(((1) .. (10)) by (3))
(((a[1]) .. (b.3)) by ((1) + (2)))
```

Some tricky cases, for example:

```
1 + 1 .. 10
- 1 .. 10
```

Should be parsed as:

```
(1 + (1 .. 10))
(- (1 .. 10))
```

The first of which is *illegal* while the second is legal but potentially *unexpected*. For the first, there is no addition operator defined between an `integer` and an `interval`. Second, the unary `-` will be applied to the entire range not just the first operand. Instead, the desired expressions are likely the following:

```
(1 + 1) .. 10
(-1) .. 10
```

By Operator

The by operator produces an `integer vector` from an `interval`. The `integer` value to the right of the `by` operator represents an increment between elements in the resulting `vector`. Values are selected beginning with and including the lower bound of the `interval`. Values after that are obtained by adding the increment to the previous value and appended to the resulting vector if it is less than or equal to the upper bound of the `interval`. For this reason, increments of zero or less are illegal. Such an increment will infinitely append values or cause an underflow. For example:

by expression	integer vector
`3..6 by 1`	`[3, 4, 5, 6]`
`3..6 by 2`	`[3, 5]`
`3..6 by 3`	`[3, 6]`
`3..6 by 4`	`[3]`

8.6.6 Type Casting and Type Promotion

To see the types that `interval` may be cast and/or promoted to, see the sections on *Type Casting* and *Type Promotion* respectively.

8.7 Vectors

Vectors are arrays that can contain any of the following base types:

- `boolean`
- `integer`
- `real`
- `character`

In *Gazprea* the number of elements in the vector also determine its type. A 3 element vector of any base type is always considered a different type from a 2 element vector.

8.7.1 Declaration

Aside from any type specifiers, the element type of the vector is the first portion of the declaration. A vector is then declared using square brackets immediately after the element type.

If possible, initialization expressions may go through an implicit type conversion. For instance, when declaring a real vector if it is initialized with an integer value the integer will be promoted to a real value, and then used as a scalar initialization of the vector.

1. Explicit Size Declarations

   When a vector is declared it may be explicitly given a size. This size can be given as any integer expression, thus the size of the vector may not be known until runtime.

   ```
   <type>[<int-expr>] <identifier>;
   <type>[<int-expr>] <identifier> = <type-expr>;
   <type>[<int-expr>] <identifier> = <type-vector>;
   ```

   The size of the vector is given by the integer expression between the square brackets.

   If the vector is given a scalar value of the same element type then the scalar value is duplicated for every single element of the vector.

   A vector may also be initialized with another vector. If the vector is initialized using a vector that is too small then the vector will be null padded. However, if the vector is initialized with a vector that is too large then a type error will occur.

2. Inferred Size Declarations

   If a vector is assigned an initial value when it is declared, then its size may be inferred. There is no need to repeat the size in the declaration because the size of the vector on the right-hand side is known.

   ```
   <type>[*] <identifier> = <type-vector>;
   ```

3. Inferred Type and Size

   It is also possible to declare a vector with an implied type and length using the var keyword. This type of declaration can only be used when the variable is initialized in the declaration, otherwise the compiler will not be able to infer the type or the size of the vector.

   ```
   integer[*] v = [1, 2, 3];
   var w = v + 1;
   ```

   In this example the compiler can infer both the size and the type of w from v. The size may not always be known at compile time, so this may need to be handled during runtime.

8.7.2 Null

Vector of `null` elements.

When initializing a vector to a value of `null` an explicit size must be given. Such initialization is equivalent to promoting a `null` value of the element type to the vector.

8.7.3 Identity

Vector of `identity` elements.

When initializing a vector to a value of `identity` an explicit size must be given. Such initialization is equivalent to promoting a `identity` value of the element type to the vector.

8.7.4 Construction

A vector value in *Gazprea* may be constructed using the following notation:

```
[expr1, expr2, ..., exprN]
```

Each `expK` is an expression with a compatible type. In the simplest cases each expression is of the same type, but it is possible to mix the types as long as all of the types can be promoted to a common type. For instance it is possible to mix integers and real numbers.

```
real[*] v = [1, 3.3, 5 * 3.4];
```

It is also possible to construct a single-element vector using this method of construction.

```
real[*] v = [7];
```

Gazprea **DOES** support empty vectors.

```
real[*] v = []; /* Should create an empty vector */
```

8.7.5 Operations

1. Vector Operations and functions
    a. length

       The number of elements in a vector is given by the built-in functions `length`. For instance:

       ```
       integer[*] v = [8, 9, 6];
       integer numElements = length(v);
       ```

       In this case `numElements` would be 3, since the vector v contains 3 elements.

    b. Concatenation

       Two vectors with the same element type may be concatenated into a single vector using the concatenation operator, `||`. For instance:

       ```
       [1, 2, 3] || [4, 5] // produces [1, 2, 3, 4, 5]
       [1, 2] || [] || [3, 4] // produces [1, 2, 3, 4]
       ```

       Concatenation is also allowed between vectors of different element types, as long as one element type is coerced automatically to the other. For instance:

       ```
       integer[3] v = [1, 2, 3];
       real[3] u = [4.0, 5.0, 6.0];
       real[6] j = v || u;
       ```

would be permitted, and the integer vector v would be promoted to a real vector before the concatenation.

Concatenation may also be used with scalar values. In this case the scalar values are treated as though they were single element vectors.

```
[1, 2, 3] || 4 // produces [1, 2, 3, 4]
1 || [2, 3, 4] // produces [1, 2, 3, 4]
```

c. Dot Product

Two vectors with the same size and a numeric element type(types with the +, and * operator) may be used in a dot product operation. For instance:

```
integer[3] v = [1, 2, 3];
integer[3] u = [4, 5, 6];

/* v[1] * u[1] + v[2] * u[2] + v[3] * u[3] */
/* 1 * 4 + 2 * 5 + 3 * 6 &=&  32 */
integer dot = v ** u;  /* Perform a dot product */
```

d. Indexing

A vector may be indexed in order to retrieve the values stored in the vector. A vector may be indexed using integers, integer vectors, and integer intervals. *Gazprea* is 1-indexed, so the first element of a vector is at index 1 (as opposed to index 0 in languages like *C*). For instance:

```
integer[3] v = [4, 5, 6];

integer x = v[2]; /* x == 5 */
integer[*] y = v[2..3]; /* y == [5, 6] */
integer[*] z = v[[3, 1, 2]]; /* z == [6, 4, 5] */
```

When indexed with a scalar integer the result is a scalar value, but when indexed with an interval or a vector the result is another vector.

Out of bounds indexing should cause an error.

e. by

The by operator is also defined for vectors of any element type. It produces a vector with every value with the given offset. For instance:

```
integer[*] v = 1..5 by 1; /* [1, 2, 3, 4, 5] */
integer[*] u = v by 1; /* [1, 2, 3, 4, 5] */
integer[*] w = v by 2; /* [1, 3, 5] */
integer[*] l = v by 3; /* [1, 4] */
```

2. Operations of the Element Type

Unary operations that are valid for the Element type of a vector may be applied to the vector in order to produce a vector whose result is the equivalent to applying that unary operation to each element of the vector. For instance:

```
boolean[*] v = [true, false, true, true];
boolean[*] nv = not v;
```

`nv` would have a value of `[not true, not false, not true, not true] = [false, true, false, false]`.

Similarly most binary operations that are valid to the element type of a vector may be also applied to two vectors. When applied to two vectors of the same size, the result of the binary operation is a vector formed by the element-wise application of the binary operation to the vector operands.

```
[1, 2, 3, 4] + [2, 2, 2, 2] // results in [3, 4, 5, 6]
```

Attempting to perform a binary operation between two vectors of different sizes should result in a type error.

When one of the operands of a binary operation is a vector and the other operand this a scalar value, then the scalar value must first be promoted with a vector of the same size as the vector operand and with the value of each element equal the scalar value. For example:

```
[1, 2, 3, 4] + 2 // results in [3, 4, 5, 6]
```

Additionally the element types of vectors may be promoted, for instance in this case the integer vector must be promoted to a real vector in order to perform the operation:

```
[1, 2, 3, 4] + 2.3 // results in [3.3, 4.3, 5.3, 6.3]
```

The equality operation is the exception to the behavior of the binary operations. Instead of producing a boolean vector, an equality operation checks whether or not all of the elements of two vectors are equal, and return a single boolean value reflecting the result of this comparison.

```
[1, 2, 3] == [1, 2, 3]
```

yields `true`

```
[1, 1, 3] == [1, 2, 3]
```

yields `false`

The != operation also produces a boolean instead of a boolean vector. The result is the logical negation of the result of the == operator.

8.7.6 Type Casting and Type Promotion

To see the types that `vector` may be cast and/or promoted to, see the sections on *Type Casting* and *Type Promotion* respectively.

8.8 String

A `string` is fundamentally a vector of `character`. However, there exists several differences between the two types: an *extra declaration style*, an *extra literal style*, the *result of a concatenation* and *behaviour when sent to an output stream*.

8.8.1 Declaration

A string may be declared with the keyword `string`. The same rules of *vector declarations* also apply to strings, allowing for both explicit and inferred size declarations:

```
string[*] <identifier> = <type-string>;
string[int-expr] <identifier> = <type-string>;
```

However, `string` variables have an extra method of writing an inferred size declaration:

```
string <identifier> = <type-string>;
```

8.8.2 Null

Same behaviour as `null` for vectors. The string is filled with `null` characters.

8.8.3 Identity

Same behaviour as `identity` for vectors. The string is filled with `identity` characters.

8.8.4 Literals

Strings can be constructed in the same way as vectors using character literals. *Gazprea* also provides a special syntax for string literals. A string literal is any sequence of character literals (including escape sequences) in between double quotes. For instance:

```
string cats_meow = "The cat said \"Meow!\"\nThat was a good day.\n"
```

8.8.5 Operations

Strings have all of the same operations defined on them as the other vector data types, but with one extra addition. Because a `string` and vector of `character` are fundamentally the same, the concatenation operation may be used to concatenate values of the two types. As well, a scalar `character` may be concatenated onto a `string` in the same way as it would be concatenated onto a vector of `character`.

This operation should always result in a value with type `string`. Again, because a `string` is always able to be converted to a vector of `character`, this is only apparent when printing the result. For example:

```
['a', 'b'] || "cd" -> std_output;
"ef" || 'g' -> std_output;
```

prints the following:

```
abcdefg
```

8.8.6 Type Casting and Type Promotion

To see the types that `string` may be cast and/or promoted to, see the sections on *Type Casting* and *Type Promotion* respectively.

8.9 Matrix

Gazprea supports two dimensional matrices. A matrix can have all of the same element types a vector can:

- `boolean`
- `integer`
- `real`
- `character`

8.9.1 Declaration

Matrix declarations are similar to vector declarations, the difference being that matrices have two dimensions instead of one. The following are valid matrix declarations:

```
integer[*, *] A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
integer[3, 2] B = [[1, 2], [4, 5], [7, 8]];
integer[3, *] C = [[1, 2], [4, 5], [7, 8]];
integer[*, 2] D = [[1, 2], [4, 5], [7, 8]];
integer[*, *] E = [[1, 2], [4, 5], [7, 8]];
```

8.9.2 Null

Matrix of `null` elements.

8.9.3 Identity

Matrix of `identity` elements.

8.9.4 Construction

To construct a matrix the programmer may use nested vectors. Each vector element represents a single row of the matrix. All rows with fewer elements than the row of maximum row length are padded with `null` values on the right. Similarly, if the matrix is declared with a column length larger than the number of rows provided, the bottom rows of the matrix are `null`. If the number of rows or columns exceeds the amounts given in a declaration an error is to be produced.

```
integer[*] v = [1, 2, 3];
integer[*, *] A = [v, [1, 2]];
/* A == [[1, 2, 3], [1, 2, 0]] */
```

Similarly, we can have:

```
integer[*] v = [1, 2, 3];
integer[3, 3] A = [v, [1, 2]];
/* A == [[1, 2, 3], [1, 2, 0], [0, 0, 0]] */
```

Also matrices can be initialized with a scalar value, `null`, or `identity`. `null` and `identity` behave as previously described. Initializing with a scalar value makes every element of the matrix equal to the scalar.

8.9.5 Operations

Matrices have binary and unary operations of the element type defined in the same manner as vectors. Unary operations are applied to every element of the matrix, and binary operations are applied between elements with the same position in two matrices.

The operators ==, and != also have the same behaviors that vectors do. These operations compare whether or not **all** elements of two matrices are equal.

In addition to this matrices have several special operations defined on them. If the element type is numeric (supports addition, and multiplication), then matrix multiplication is supported using the operator **. Matrix multiplication is only defined between matrices with compatible element types, and the dimensions of the matrices must be valid for performing a matrix multiplication. If this is not the case then an error should be raised.

All matrices support the built in functions `rows` and `columns`, which when passed a matrix yields the number of rows and columns in the matrix respectively. For instance:

```
integer[*, *] M = [[1, 1, 1], [1, 1, 1]];

integer r = rows(M);  /* This has a value of 2 */
integer c = columns(M);  /* This has a value of 3 */
```

Matrix indexing is done similarly to vector indexing, however, two indices must be used. These indices are separated using a comma.

```
M[i, j] -> std_output;
```

The first index specifies the row of the matrix, and the second index specifies the column of the matrix. The result is retrieved from the row and column. Both the row and column indices can be either integers, integer intervals, or integer vectors. When both indices are scalar integers the result is the scalar value in the row and column specified.

```
integer[*, *] M = [[11, 12, 13], [21, 22, 23]];

/* M[1, 2] == 12 */
```

If one of the indices is an interval or a vector, and the other index is a scalar, then the result is a vector. For example:

```
integer[*, *] M = [[11, 12, 13], [21, 22, 23]];

/* Select from row 2 */
/* M[2, 2..3] == [22, 23] */
/* M[2, [3, 2]] == [23, 22] */

/* Select from column 1 */
/* M[1..2, 1] == [11, 21] */
/* M[[2, 1], 1] == [21, 11] */
```

Finally, both of the indices may be intervals or vectors, in which case the result is another matrix.

```
integer[*, *] M = [[11, 12, 13], [21, 22, 23]];

/* Makes a matrix consisting of [[M[2, 1], M[2, 3]], [M[1, 1], M[1, 3]]] */
integer[*, *] K = M[[2, 1], [1, 3]];
```

As with vectors, out of bounds indexing is an error on Matrices.

8.9.6 Type Casting and Type Promotion

To see the types that `matrix` may be cast and/or promoted to, see the sections on *Type Casting* and *Type Promotion* respectively.

CHAPTER

NINE

TYPE INFERENCE

In many cases the compiler can figure out what a variable's type, or a function's return type should be without an explicit type being provided. For instance, instead of writing:

```
integer x = 2;
const integer y = x * 2;
```

Gazprea allows you to just write:

```
var x = 2;
const y = x * 2;
```

This is allowed because the compiler knows that the initialization expression, 2, has the type integer. Because of this the compiler can automatically give x an integer type. A *Gazprea* programmer can use `var` or `const` for any declaration with an initial value expression, as long as the compiler can guess the type for the expression.

One case where `var` or `const` will lead to an error is if the initial value is a polymorphic constant such as `null`, or `identity`. *Gazprea*'s type inference is simple, and only relies upon a single expression, so it can not discover the type of x in:

```
var x = null;  /* Can't tell what type this is */
const y = identity; /* Can't tell what type this is either */
integer z = x;
```

Clearly the type that makes the most sense for x here would by `integer`, but *Gazprea* only checks the initialization expression, and does not see how the variable x is used across statements. As a result an error should be raised in this situation.

Gazprea employs a very simple type inference algorithm on expressions. It finds the type that makes sense across binary expressions in a bottom up fashion. For instance:

```
/* The type for 'x' can be infered to be an integer. This expression
   can be rewritten as (null + 1) + null. The type inference algorithm
   checks (null + 1), and decides that since 1 is an integer 'null'
   must also be an integer because '+' can only be applied to numbers
   of the same type. Similarly it then infers that since (null + 1) is
   an integer (null + 1) + null is an integer as well, thus 'x' must
   be an integer. */

var x = null + 1 + null;

/* The simple type inference algorithm can not handle this case, and a
   type ambiguity error should be raised. Since this expression is
```

(continues on next page)

(continued from previous page)

```
   (null + null) + 1 the type inference algorithm will try to figure
   out the type for (null + null), but since it doesn't know what
   either of the null values types are it can't. */

var y = null + null + 1;
```

CHAPTER

TEN

VECTOR/MATRIX TYPE CHECKING

While the size of vectors and matrices may not always be known at compile time, there are instances where the compiler can perform length checks at compile time. For instance:

```
integer v[3] = 1..10;
```

In cases like these the compiler is always able to catch the size mismatch, since the vector `1..10` is known at compile time.

Your compiler should only handle the case where the initialization expression consists of an expression with only literal values (thus, it can be evaluated at compile time). Similarly the size of the declared vector must either be given with an expression of literal values, or not be provided. If a size mismatch is detected here the compiler should throw an error. The compiler should also be able to detect cases such as:

```
integer[*] v = 1;
```

where the length of the vector can not be determined at all.

CHAPTER

ELEVEN

TYPE CASTING

Gazprea provides explicit type casting. A value may be converted to a different type using the following syntax where `value` is an expression and `toType` is our destination type:

```
as<toType>(value)
```

Conversions from one type to another is not always legal. For instance converting from an `integer matrix` to an `integer` has no reasonable conversion.

11.1 Scalar to Scalar

This table summarizes all of the conversion rules between scalar types where N/A means no conversion is possible, id means no change is necessary, and anything else describes how to convert the value to the new type:

	To type				
From type		boolean	character	integer	real
	boolean	id	'\0' if false, 0x01 otherwise	1 if true, 0 otherwise	1.0 if true, 0.0 otherwise
	character	false if '\0', true otherwise	id	*ASCII* value as integer	*ASCII* value as real
	integer	false if 0, true otherwise	unsigned integer value mod 256	id	real version of integer
	real	N/A	N/A	truncate	id

11.2 Scalar to Vector/Matrix

A scalar may be promoted to either a `vector` or `matrix` with an element type that the original scalar can be cast to according to the rules in *Scalar to Scalar*. A scalar to vector cast *must* include a size with the type to cast to as this cannot be inferred from the scalar value. For example:

```
// Create a vector of reals with length three where all values are 1.0.
real[*] v = as<real[3]>(1);

// Create a vector of booleans with length 10 where all values are true.
var u = as<boolean[10]>('c');
```

11.3 Interval to Vector

An `integer interval` may be explicitly cast to an `integer` or `real vector` as in the *type promotion rules*, but the explicit cast can cause the interval to be truncated or `null` padded.

11.4 Vector to Vector

Conversions between `vector` types are also possible. First, the values of the original are casted to the destination type's element type according to the rules in *Scalar to Scalar* and then the destination is padded with destination element type's `null` or truncated to match the destination type size. Note that the size is not required for vector to vector casting; if the size is not included in the cast type, the new size is assumed to be the old size. For example:

```
real[3] v = [i in 1..3 | i + 0.3 * i];

// Convert the real vector to an integer vector.
integer[3] u = as<integer[*]>(v);

// Convert to integers and null pad.
integer[5] x = as<integer[5]>(v);

// Truncate the vector.
real[2] y = as<real[2]>(v);
```

11.5 Matrix to Matrix

Conversions between matrix types are also possible. The process is exactly like *Vector to Vector* except padding and truncation can occur in both dimensions. For example:

```
real[2, 2] a = [[1.2, 24], [-13e2, 4.0]];

// Convert to an integer matrix.
integer[2, 2] b = as<integer[2, 2]>(a);

// Convert to integers and pad in both dimensions.
integer[3, 3] c = as<integer[3, 3]>(a);

// Truncate in one dimension and pad in the other.
real[1, 3] d = as<real[1, 3]>(a);
real[3, 1] e = as<real[3, 1]>(a);
```

11.6 Tuple to Tuple

Conversions between `tuple` types are also possible. The original type and the destination type must have an equal number of internal types and each element must be pairwise castable according to the rules in *Scalar to Scalar*. For example:

```
tuple(integer, integer) int_tup = (1, 2);
tuple(real, boolean) rb_tup = as<tuple(real, boolean)>(int_tup);
```

11.7 Null and Identity

The `null` and `identity` values cannot be cast. For example, the following is illegal:

```
real r = as<real>(null);
```

CHAPTER

TWELVE

TYPE PROMOTION

Type promotion is a sub-problem to and refers to casts that happen implicitly without extra syntax such as using `as`.

12.1 Scalars

The only automatic type promotion for scalars is `integer` to `real`. This promotion is one way - a `real` cannot be automatically converted to `integer`.

Automatic type conversion follows this table where N/A means no implicit conversion possible, id means no conversion necessary, `as<toType>(var)` means var of type "From type" is converted to type "toType" using semantics from .

	To type				
From type		boolean	character	integer	real
	boolean	id	N/A	N/A	N/A
	character	N/A	id	N/A	N/A
	integer	N/A	N/A	id	as<real>(var)
	real	N/A	N/A	N/A	id

12.2 Scalar to Vector or Matrix

All scalar types can be promoted to `vector` or `matrix` types that have an internal type that the scalar can be *converted to implicity*. This can occur when a `vector` or `matrix` is used in an operation with a scalar value.

The scalar will be implicitly converted to a `vector` or `matrix` of equivalent dimensions and equivalent internal type. For example:

```
integer i = 1;
integer[*] v = [1, 2, 3, 4, 5];
integer[*] res = v + i;

res -> std_output;
```

would print the following:

```
[2 3 4 5 6]
```

12.3 Interval to Vector

An `interval` can be implicitly converted to an identically-sized `vector` of any type that `integer` can be *converted to implicity*. For example:

```
integer interval i = 1..5;
integer[5] iv = i;
real[*] rv = i;
```

12.4 Tuple to Tuple

Tuples may be promoted to another tuple type if it has an equal number of internal types and the original internal types can be implicitly converted to the new internal types. For example:

```
tuple(integer, integer) int_tup = (1, 2);
tuple(real, real) real_tup = int_tup;

tuple(char, integer, boolean[2]) many_tup = ('a', 1, [true, false]);
tuple(char, real, boolean[2]) other_tup = many_tup;
```

Field names of tuples are overwritten by the field names of the left-hand side in assignments and declarations when promoted. For example:

```
tuple(integer a, real b) foo = (1, 2);
tuple(real c, real) bar = foo;

foo.a -> std_output; // 1
foo.b -> std_output; // 2

bar.a -> std_output; // error
bar.b -> std_output; // error
bar.c -> std_output; // 1
```

If initializing a variable with a tuple via *Type Inference*, the variable is assumed to be the same type. Therefore, field names are also copied over accordingly. For example:

```
tuple(real a, real b) foo = (1, 2);
tuple(real c, real d) bar = (3, 4);

var baz = foo;
baz.a -> std_output; // 1
baz.b -> std_output; // 2

baz = bar;
baz.a -> std_output; // 3
baz.b -> std_output; // 4
```

It is possible for a two sided promotion to occur with tuples. For example:

```
boolean b = (1.0, 2) == (2, 3.0);
```

12.5 Character Vector to/from String

A `string` can be implicitly converted to a `vector` of `character`s and vice-versa (two-way type promotion).

```
string str1 = "Hello"; /* str == "Hello" */
character[*] chars = str; /* chars == ['H', 'e', 'l', 'l', 'o'] */
string str2 = chars || [' ', 'W', 'o', 'r', 'l', 'd']; /* str2 == "Hello World" */
```

CHAPTER

THIRTEEN

TYPEDEF

Custom names for types can be defined using `typedef`. Typedefs may only appear at global scope, they may not appear within functions or procedures. A typedef may use any valid identifier for the name of the type. After the typedef has been defined any global declaration or function defined may use the new name to refer to the old type. For instance:

```
typedef integer int;
const int a = 0;
```

Additionally, these new type names should not conflict with variable names. The following is therefore legal code:

```
typedef integer a;
const integer a = 0;
```

We can also typedef vectors and matrices with sizes for easy reusability:

```
typedef integer[10] ten_ints;
const ten_ints a = [i in 1..10 | 7];

typedef integer[2,3] two_by_three_matrix;
two_by_three_matrix m = [i in 1..2, j in 1..3 | i + j];
```

Typedefs of vectors and matrices with inferred sizes are allowed, but declarations of variables using the typedef must be initialized appropriately.

CHAPTER

FOURTEEN

EXPRESSIONS

14.1 Table of Operator precedence

The following is a table containing all of the precedences and associativities of the operators in *Gazprea*.

Precedence	Operators	Associativity
(Highest) 1	`.`	left
2	`[]` (indexing)	left
3	`..`	N/A
4	unary +, unary -, `not`	right
5	`^`	right
6	*, /, %, **	left
7	+, -	left
8	`by`	left
9	<, >, <=, >=	left
10	==, !=	left
11	`and`	left
12	`or`, `xor`	left
(Lowest) 13	`||`	right

14.2 Generators

A generator may be used to construct either a vector or a matrix. A generator creates a value of a vector type when one domain variable is used, and a generator creates a value of a matrix type when two domain variables are used. Any other number of domain variables will yield an error.

A generator consists of either one or two domain expression. An additional expression is used on the right hand side in order to create the generated values. For example:

```
integer[10] v = [i in 1..10 | i * i];
/* v[i] == i * i */

integer[2, 3] M = [i in 1..2, j in 1..3 | i * j];
/* M[i, j] == i * j */
```

The expression to the right of the bar "|", is used to generate the value at the given index, and must result in a value with the same type as the element type for the matrix or vector. Generators may be nested, and may be used within domain expressions. For instance, the generator below is perfectly legal:

```
integer i = 7;

/* The domain expression should use the previously defined i */
integer[*] v = [i in [i in 1..i | i] | [i in 1..10 | i * i][i]];

/* v should contain the first 7 squares. */
```

14.3 Filters

Filters are used to accumulate elements into vectors. Each filter contains a single domain expression, and a list of comma-separated predicates.

The result of a filter operation is a tuple. This tuple contains a field for each of the predicates in order. Each field is a vector containing only the elements from the domain which satisfied the predicate expressions. Each filter result has an additional field which is a vector containing all of the values in the domain which did not satisfy any of the predicates. For example:

```
/* x == ([3], [2], [2, 4], [1, 5]) */
var x = [i in 1..5 & i == 3, i == 2, i % 2 == 0];

/* y == ([1, 3, 5], [2, 4]) */
var y = [i in 1..5 & i % 2 == 1);
```

There must be at least one predicate expression

14.4 Domain Expressions

Domain expressions can only appear within iterator loops, generators, and filters. A domain expression is a way of declaring a variable that is local to the loop, generator, or filter, that takes on values from intervals, and vectors in order.

Domain expressions are essentially declarations, and so they follow the same scoping rules. For instance:

```
integer i = 7;

/* This will print 1234567 */
loop i in 1..i {
  i -> std_output;
}
```

Domain variables are not initialized when they are declared. For instance in loops they are initialized at the start of each execution of the loop's body statement. However, we may chain domain variables using commas, like in iterator loops, or matrix generators. Thus it is illegal to use a domain variable declared in the same chain of domain expressions, since the value may be uninitialized.

```
integer i = 7;

/* This is illegal because the i in "j in 1..i" refers to the domain
   variable i. An error should be raised in this case. */
loop i in 1..i, j in 1..i {
   i * j -> std_output;
```

(continues on next page)

(continued from previous page)

```
}

/* This is legal since i will be initialized whenever the inner loop
   is executed */
loop i in 1..i {
  loop j in 1..i {
    i * j -> std_output;
  }
}
```

The domain for the domain expression is only evaluated once. For instance:

```
integer x = 1;

/* 1..x is only evaluated the first time the loop executes, so it is
   simply 1..1, and not an infinite loop. */
loop i in 1..x {
  x = x + 1;
}
```

This is true for domain expressions within generators and filters as well.

CHAPTER

FIFTEEN

STATEMENTS

15.1 Assignment Statements

In *Gazprea* a variable may have different values throughout the execution of the program. Variables may have their values changed with an assignment statement. In the simplest case an assignment statement contains an identifier on the left hand side of an equals sign, and an expression with a compatible type on the right hand side.

```
integer x = 7;

x -> std_output;  /* Prints 7 */

/* Give 'x' a new value */
x = 2 * 3;  /* This is an assignment statement */

x -> std_output;  /* Prints 6 */
```

Type checking must be performed on assignment statements. The expression on the right hand side must have a type that can be automatically promoted to the type of the variable. For instance:

```
integer int_var = 7;
real real_var = 0.0;
boolean bool_var = true;

/* Since 'x' is an integer it can be promoted to a real number */
real_var = int_var;  /* Legal */

/* Real numbers can not be turned into boolean values automatically. */
bool_var = real_var; /* Illegal */
```

Assignments can also be more complicated than this with vectors, matrices, and tuples. With matrices and vectors indices may be provided in order to change the value of a portion of the matrix or vector. For instance, with vectors:

```
integer[*] v = [0, 0, 0];

/* Can assign an entire vector value -- change 'v' to [1, 2, 3] */
v = [1, 2, 3];

/* Change 'v' to [1, 0, 3] */
v[2] = 0;

/* Can also use vector indexing */
```

(continues on next page)

(continued from previous page)

```
v[[1, 3]] = [4, 5];  /* 'v' is now [4, 0, 5] */

integer[*] w = [3, 2, 1];

/* Also note this special case */
w[w] = [2, 2, 2]; /* 'w' is now [3, 2, 2] */
// The above assignment is semantically equivalent to the following loop
loop i in 1..3 {
    w[w[i]] = 2;
}
```

Matrices can be treated similarly.

```
integer[*, *] M = [[1, 1], [1, 1]];

/* Change the entire matrix M to [[1, 2], [3, 4]] */
M = [[1, 2], [3, 4]];

/* Change a single position of M */
M[1, 2] = 7;  /* M is now [[1, 7], [3, 4]] */

/* Can use vector indexing on rows or columns.
   Uses all combinations of row / column coordinates */
```

Tuples also have a special unpacking syntax in *Gazprea*. A tuple's field may be assigned to comma separated variables instead of a tuple variable. For instance:

```
integer x = 0;
real y = 0;
real z = 0;

tuple(integer, real) tup = (1, 2.0);

/* x == 1, and y == 2.0 now */
x, y = tup;

/* Types can be promoted */

/* z == 1.0, y == 2.0 */
z, y = tup;

/* Can swap: z == 2.0, y == 1.0 */
z, y = (y, z);
```

The types of the variables must match the types of the tuple's fields, or the tuple's fields must be able to be automatically promoted to the variable's type. The number of variables in the comma separated list must match the number of fields in the tuple, if this is not the case an error should be raised.

Assignments and initializations must perform a deep copy. It should not be possible to cause the aliasing of memory locations with an assignment. For instance:

```
integer[*] v = [1, 2, 3];
integer[*] w = v;
```

(continues on next page)

(continued from previous page)

```
w[2] = 0;  /* This must not affect 'v' */

/* v has the value [1, 2, 3] */
/* w has the value [1, 0, 3] */

/* If you are not careful, you might copy the pointer of 'v' to 'w',
   which would cause them to be stored in the same location in memory. If
   this happens modifying 'w' would change 'v' as well.
 */
```

The above is a simple example using vectors. You must ensure that values can not be aliased with an assignment between any types, including vectors, matrices, and tuples.

Variables may be declared as const, and in this case it is illegal for them to appear on the left hand side of an assignment expression. The compiler should raise on error when this is detected, since it does not make sense to change a constant value.

The right hand side of an assignment statement is always evaluated before the left hand side. This is important for cases where procedures may change variables, for instance:

```
v[x] = p(x);
/* If p changes x then it is important that p(x) is executed before v[x] */
```

15.2 Block Statements

A list of statements may be grouped into one statement using curly braces. This is called a block statement, and is similar to block statements in other languages such as *C/C++*. As an example:

```
{
  x = 3;
  z = 4;
  x -> std_output; "\n" -> std_output; z -> std_output; "\n" -> std_output;
}
```

Is a block statement. Declarations can only appear at the start of a block. Each block statement introduces a new scope that new variables may be declared in. For instance this is perfectly valid:

```
integer x = 3;
integer y = 0;
real z = 0;

{
  real x = 7.1;
  z = x;
}

y = x;
```

After execution this `y = 3` and `z = 7.1`.

15.3 If/Else Statements

An if statement takes a boolean value as a conditional expression, and a statement for the body. If the conditional expression evaluates to true, then the body is executed. If the conditional expression evaluates to false then the body of the if statement is not executed. If statements in *Gazprea* do not require the conditional expression to be enclosed in parenthesis.

```
integer x = 0;
integer y = 0;

/* Compute some value for x */

if (x == 3) {
   y = 7;
}

/* At this point y will only be 7 if x == 3, and otherwise y will be
   0, assuming it did not change throughout the rest of the program.
 */
```

If statements are often paired with block statements, like in the above example. The if statement above could also be written as:

```
if x == 3
  y = 7;
```

Since `y = 7;` is a statement it can be used as the body statement. All statements after this point are not in the body of the if statement. For instance:

```
if x == 3
  y = 7;
  z = 32;
```

is actually equivalent to the following:

```
if (x == 4) {
  y = 7;
}

z = 32;
```

Gazprea is not sensitive to whitespace, so we could even write something like:

```
if x == 3 y = 7;
```

An if statement may also be followed by an else statement. The else has a body statement just like the if statement, but this is only run if the conditional expression on the if statement fails.

```
if x == 3
  y = 7;
else
  y = 32;
```

Now if `x` does not have a value of 3, `y` is assigned a value of 32. This can be paired with if statements as well.

```
y = 0;

if (x < 0) {
  y = -1;
}
else if (x > 0) {
  y = 1;
}

/* y is negative if x is negative, positive if x is positive,
  and 0 if x is 0. */
```

15.4 Loop

15.4.1 Infinite Loop

Gazprea provides an infinite loop, which continuously executes the body statement given to it. For instance:

```
loop "hello!\n" -> std_output;
```

Would print "hello!" indefinitely. This is often used with block statements.

```
/* Infinite counter */
integer n = 0;

loop {
  n -> std_output; "\n" -> std_output;
  n = n + 1;
}
```

15.4.2 Predicated Loop

A loop may also be provided with a control expression. The control expression automatically breaks from the loop if it evaluates to false when it is checked.

The loop can be pre-predicated, which means that the control expression is tested before the body statement is executed. This is the same behaviour as while loops in most languages, and is written using the while token after the loop, followed by a boolean expression for the predicate. For example:

```
integer x = 0;

/* Print 1 to 10 */
loop while x < 10 {
  x = x + 1;
  x -> std_output; "\n" -> std_output;
}
```

A post-predicated loop is also available. In this case the control expression is tested after the body statement is executed. This also uses the while token followed by the control expression, but it appears at the end of the loop. Post Predicated loop statements must end in a semicolon.

```
integer x = 10;

/* Since the conditional is tested after the execution '10' is printed */
loop x -> std_output; while x == 0;
```

15.4.3 Iterator Loop

Loops can be used to iterate over the elements of an integer interval, or a vector of any type. This is done by using domain expressions (for instance i in v) in conjunction with a loop statement.

When the domain is given by a vector, each time the loop is executed the next element of the vector is assigned to the domain variable. The elements of the domain vector are assigned to the domain variable starting from index 1, and going up to the final element of the vector. When all of the elements of the domain vector have been used the loop automatically exits. For instance:

```
/* This will print 123 */
loop i in [1, 2, 3] {
  i -> std_output;
}
```

Integer intervals can also be used instead. In this case it is the same as iterating over a vector created from the interval using by 1. For instance, the above iterator loop is equivalent to the following:

```
/* This will print 123 */
loop i in 1..3 {
  i -> std_output;
}
```

The domain is evaluated once during the first iteration of the loop. For instance:

```
integer[*] v = [i in 1..3 | i];

/* Since the domain 'v' is only evaluated once this loop prints 1, 2,
   and then 3 even though after the first iteration 'v' is the zero
   vector. */
loop i in v {
  v = 0;
  i -> std_output; "\n" -> std_output;
}
```

Multiple domain expressions may be used by separating them with commas.

```
loop i in u, j in v {
  "Hello!\n" -> std_output;
}

/* The above loop is equivalent to the loop below */

loop i in u {
  loop j in v {
    "Hello!\n" -> std_output;
  }
}
```

This can be done with as many domain expressions as desired.

15.5 Break

A break statement may only appear within the body of a loop. When a break statement is executed the loop is exited, and *Gazprea* continues to execute after the loop. This only exits the innermost loop, which actually contains the break.

```
/* Prints a 3x3 square of *'s */
integer x = 0;
integer y = 0;

loop while y < 3 {
  y = y + 1;

  /* Normally this would loop forever, but the break exits this inner loop */
  loop {
    if x >= 3 break;

    x = x + 1;
    "*" -> std_output;
  }

  "\n" -> std_output;
}
```

If a break statement is not contained within a loop an error must be raised.

15.6 Continue

Similarly to `break`, `continue` may only appear within the body of a loop. When a `continue` statement is executed the innermost loop that contains the `continue` statements starts its next iteration. `continue` stops the execution of the loop's body statement, the loop then continues as though the body statement finished its execution normally.

```
/* Prints every number between 1 and 10, except for 7 */
integer x = 0;

loop while x < 10 {
  x = x + 1;

  if x == 7 continue;  /* Start at the beginning of the loop, skip 7 */

  x -> std_output; "\n" -> std_output;
}
```

15.7 Return

The `return` statement is used to stop the execution of a function or procedure. When a function/procedure returns then execution continues where the function/procedure was called.

If the function/procedure has a return type then the `return` statement must be given a value that is the same as or able to be promoted to (see *Type Promotion*) the return type; this will be the result of the function/procedure call. Here is an example:

```
function square(integer x) returns integer {
  return x * x;
}
```

If a procedure has no `returns` clause, then it has no return type and a `return` statement is not required but may still be present in order to return early. In this case return is used as follows:

```
procedure do_nothing() {
  return;
}
```

15.8 Stream Statements

Stream statements are the statements used to read and write values in *Gazprea*.

Output example:

```
2 * 3 -> std_output;  /* Prints 6 */
```

Input example:

```
integer x = null;
x <- std_input; /* Read an integer into x */
```

CHAPTER

SIXTEEN

FUNCTIONS

A function in *Gazprea* has several requirements:

- All of the arguments are implicitly `const`, and can not be mutable.
- Functions can not perform any I/O.
- Functions can not rely upon any mutable state outside of the function.
- Functions can not call any procedures.

The reason for this is to ensure that functions in *Gazprea* behave as pure functions. Every time you call a function with the same arguments it will perform the exact same operations. This has a lot of benefits. It makes code easier to understand if functions only depend upon their inputs and not some hidden state, and it also allows the compiler to make more assumptions and as a result perform more optimizations.

16.1 Syntax

A function is declared using the function keyword. Each function is given an identifier, and an arguments list enclosed in parenthesis. If no arguments are provided an empty set of parenthesis, `()`, must be used. The return type of the function is specified after the arguments using returns.

A function can be given by a single expression. For instance:

```
function times_two(integer x) returns integer = 2 * x;
```

This defines a function called times_two which can be used as follows:

```
/* Prints 8. value gets assigned the result of calling times_two with an
   argument of 4
 */
integer value = times_two(4);

value -> std_output; "\n" -> std_output;
```

Functions can have an arbitrary number of arguments. Here are some examples of functions with different numbers of arguments:

```
/* A function with no arguments */
function f() returns integer = 1;

/* A function with two arguments */
function pythag(real a, real b) returns real = (a^2 + b^2)^(1./2);
```

(continues on next page)

(continued from previous page)

```
/* A function with different types of arguments */
function get(real[*] a, integer i) returns real = a[i];
```

These can be called as follows:

```
integer x = f(); /* x == 1 */
real c = pythag(3, 4); /* Type promotion to real arguments. c == 5.0 */
real value = get([i in 1..10 | i], 3); /* value == 3 */
```

A function's body can also be given by a block statement instead of a single expression. In this case the return value of the function is given with the return statement. A return statement must be reached by all possible control flows in the function before the end of the function is encountered.

```
/* Invalid -- should cause a compiler error */
function f (boolean b) returns integer {
  if (b) {
    return 3;
  }
}

/* Valid, all possible branches hit a return statement with a valid type */
function g (boolean b) returns integer {
  if (b) {
    return 3;
  }
  else {
    return 8;
  }
}
```

`f` is invalid since if `b == false`, then we reach the end of the function without a return statement, so we don't know what value `f(false)` should take on.

```
/* This is invalid because if the loop ever finished executing the
   function would end before a return statement is encountered. In
   general the compiler can not tell when a loop would execute
   forever, so we make the assumption that all branches in the control
   flow could be followed. */
function f() returns integer {
  integer x = 0;
  loop {
    x = x + 1;
  }
}

/* This is valid. Even though the loop goes on forever so that a
   return is never reached, execution never hits the end of the
   function without a return. */
function g() return integer {
  integer x = 0;
  loop {
    x = x + 1;
```

(continues on next page)

(continued from previous page)

```
  }

  return x;
}
```

Each function has its own scope, but globals can be accessed within the function if they were declared before the function was defined.

16.2 Forward Declaration

Functions can be declared before they are defined in a *Gazprea* file. This allows function definitions to be moved to more convenient locations in the file.

```
/* Forward declaration, no body */
function f(integer x) returns integer;

procedure main() returns integer {
  integer y = f(13);
  /* Can use this in main, even though the definition is below */
  return 0;
}

function f(integer x) returns integer = x^2;
```

If the type signatures of the forward declaration of the function and the definition of the function differ then an error must be raised. A function may only be declared once.

Note that only the type signatures of the forward declaration of the function and the definition must be identical. Therefore, the forward declaration of the function may have different argument names from its definition.

A function that has a forward declaration must have a definition somewhere within the file. If the function does not have a definition then an error should be raised.

16.3 Vector and Matrix Parameters and Returns

The arguments and return value of functions can have both explicit and inferred sizes. For example:

```
function to_real_vec(integer[*] x) returns real[*] {
    /* Some code here */
}

function transpose3x3(real[3,3] x) returns real[3,3] {
    /* Some code here */
}
```

CHAPTER

SEVENTEEN

PROCEDURES

A procedure in *Gazprea* is like a function, except that it does not have to be pure and as a result it may:

- Have arguments marked with `var` which can be mutated. By default arguments are `const` just like functions.
- A procedure may only accept a literal or expression as an argument if and only if the procedure declares that argument as `const`.
- Procedures may perform I/O.
- A procedure can call other procedures.
- Procedures can only be called in assignment statements/procedure call statements.
- When used in an assignment statement the procedure may only be used with unary operations.

Aside from this (and the different syntax necessary to declare/define them), procedures are very similar to functions. The extra capabilities that procedures have makes them harder to reason about, test, and optimize.

17.1 Syntax

Procedures are almost exactly the same as functions. However, because procedures can cause side effects, the returns clause is optional. Due to this, the `= <stmt>;` declaration format is not available for procedures. For example, the following code is illegal:

```
procedure f() returns int = 1;
```

If a returns clause is present, then a return statement must be reached by all possible control flows in the procedure before the end of the procedure is encountered. For instance:

```
procedure change_first(var integer[*] v) {
  v[1] = 7;
}

procedure increment(var integer x) {
  x = x + 1;
}

procedure fibonacci(var integer a, var integer b) returns integer {
  integer c = a + b;
  a = b;
  b = c;
  return c;
}
```

These procedures can be called as follows:

```
integer x = 12;
integer y = 21;
integer[5] v = 13;

call change_first(v); /* v == [7, 13, 13, 13, 13] */
call increment(x); /* x == 13 */
call fibonacci(x,y); /* x == 21 and y == 34 */
```

It is only possible to call procedures in this way. Functions must appear in expressions because they can not cause side effects, so using a function as a statement would not do anything, and thus *Gazprea* should raise an error. If the procedure has a return value and is called in this fashion the return value is discarded.

Procedures may also be called in expressions just like functions, but with a few more limitations. A procedure may never be called within a function, doing so would allow for impure functions. Procedures may only be called within assignment statements (procedures may not be used as the control expression in control flow expressions, for instance). The return value from a procedure call can only be manipulated with unary operators. It is illegal to use the results from a procedure call with binary expressions, for instance:

```
/* p is some procedure with no arguments */
var x = p(); /* Legal */
var y = -p(); /* Legal, depending on the return type of p */
var z = not p(); /* Legal, depending on the return type of p */
var u = p() + p(); /* Illegal */
```

This restriction is made by `Gazprea` in order to allow for more optimizations.

As long as they have an appropriate return type. The difference is that functions can be called within other functions, but procedures can not be used within functions since procedures may be impure. Procedures may only be called within procedures.

Procedures without a return clause may not be used in an expression. *Gazprea* should raise an error in such a case.

```
/* p is some procedure with no return clause */
var x = 1 + p(); /* Illegal */
```

17.2 Forward Declaration

Procedures can use *forward declaration just like functions*.

17.3 Main

Execution of a *Gazprea* program starts with a procedure called `main`. This procedure takes no arguments, and has an integer return type. If a program is missing a main procedure an error should be raised.

```
/* must be writen like this */
procedure main() returns integer {
  integer x = 1;
  x = x + x;
  x -> std_output;

```

(continues on next page)

(continued from previous page)

```
  /* must have a return */
  return 0;
}
```

17.4 Aliasing

Since procedures can have mutable arguments, it would be possible to cause aliasing. In *Gazprea* aliasing of mutable variables is illegal (the only case where any aliasing is allowed is that tuple members can be accessed by name, or by number, but this is easily spotted). This helps *Gazprea* compilers perform more optimizations. However, the compiler must be able to catch cases where mutable memory locations are aliased, and an error should be raised when this is detected. For instance:

```
procedure p(var integer a, var integer b, const integer c, const integer d) {
   /* Some code here */
}

procedure main() returns integer {
  integer x = 0;
  integer y = 0;
  integer z = 0;

  /* Illegal */
  call p(x, x, x, x); /* Aliasing, this is an error. */
  call p(x, x, y, y); /* Still aliasing, error. */
  call p(x, y, x, x); /* Argument a is mutable and aliased with c and d. */

  /* Legal */
    call p(x, y, z, z);
    /* Even though 'z' is aliased with 'c' and 'd' they are
    both const. */

  return 0;
}
```

Whenever a procedure has a mutable argument x it must be checked that none of the other arguments given to the procedure are x. This is simple for scalar values, but more complicated when variable vectors and matrices are passed to procedures. For instance:

```
call p(v[1..5], v[6..10]);
/* p is some procedure with two variable vector arguments */
```

In this case the arguments technically wouldn't be aliased, since the vector slices represent different locations in memory, but since the vector slices may depend upon variables:

```
call p(v[x], v[y]);
/* p is some procedure with two variable vector arguments */
```

It is impossible to tell whether or not these overlap at compile time due to the halting problem. Thus for simplicity, whenever a vector or a matrix is passed to a procedure *Gazprea* detects aliasing whenever the same vector / matrix is used, regardless of whether or not the sections used would overlap. Thus, this should cause an error to be raised:

```
call p(v[1..5], v[6..10]);
/* p is some procedure with two variable vector arguments */
```

17.5 Vector and Matrix Parameters and Returns

As with functions, the arguments and return value of procedures can have both explicit and inferred sizes.

CHAPTER

EIGHTEEN

GLOBALS

In *Gazprea* values can be assigned to a global identifier. All globals must be declared `const`. If a global identifier is not declared with the `const` specifier, then an error should be raised. This restriction is in place since mutable global variables would ruin functional purity. If functions have access to mutable global state then we can not guarantee their purity.

Globals must be initialized, but the initialization expressions must not contain any function calls, or procedures. If a global is initialized with an expression containing a function call, or a procedure call, then an error should be raised. Initializations of globals may refer to previously defined globals.

CHAPTER

NINETEEN

BUILT IN FUNCTIONS

Gazprea has some built in functions. These built in functions may have some special behaviour that normal functions can not have, for instance many of them will work on vectors of any element type. Normally a function must specify the element type of a vector specified.

The name of built in functions are reserved and a user program cannot define a function or a procedure with the same name as a built in function. If a declaration or a definition with the same name as a built-in function is encountered in a *Gazprea* program, then the compiler should issue an error message.

19.1 Length

`length` takes a vector of any element type, and returns an integer representing the length of the vector.

```
integer[*] v = 1..5;

length(v) -> std_output; /* Prints 5 */
```

19.2 Rows and Columns

The built-ins `rows` and `columns` operate on matrices of any dimension and type. `rows` returns the number of rows in a matrix, and `columns` returns the number of columns in the matrix.

```
integer[*, *] M = [[1, 2, 3], [4, 5, 6]];

rows(M) -> std_output; /* Prints 2 */
columns(M) -> std_output; /* Prints 3 */
```

19.3 Reverse

The reverse built-in takes any vector, and returns a reversed version of the vector.

```
integer[*] v = 1..5;
integer[*] w = reverse(v);

v -> std_output; /* Prints 12345 */
w -> std_output; /* Prints 54321 */
```

19.4 Stream State

When reading values of certain types from `std_input` it is possible that an error is encountered, or that the end of the stream has been encountered. In order to handle these situations *Gazprea* provides a built in procedure that is implicitly defined in every file:

```
procedure stream_state(var input_stream) returns integer;
```

This function can only be called with the `std_input` as a parameter, but it's general enough that it could be used if the language were expanded to include multiple input streams.

When called, `stream_state` will return an integer value. The return value is an error code defined as follows:

- `0`: Last read from the stream was successful
- `1`: Last read from the stream encountered an error.
- `2`: Last read from the stream encountered the end of the stream.

CHAPTER

TWENTY

BACKEND

You don't need to implement an interpreter for Gazprea. You only need to implement a llvm code generator.

20.1 Memory Management

It is important that you are able to automatically free and allocate memory for vectors and matrices when they enter and exit scope. You may use `malloc` and `free` for these purposes. This may be done in either your runtime or directly within LLVM, though you may find it easier to do in LLVM.

Below is an example of how to use `malloc` and `free` within LLVM:

```
define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i8* @malloc(i64 128)
  call void @free(i8* %1)
  ret i32 0
}

declare i8* @malloc(i64)
declare void @free(i8*)
```

It is important that the code generated by your compiler has no memory leaks, and that all memory is freed as it leaves scope.

20.2 Runtime Libraries

If you make a runtime library, the runtime library must be implemented in a runtime directory (lib). Beware that in C++ there is additional name mangling that occurs to allow class functions. Thus, we recommend that all runtime functions should be written in C and not in C++. There is a Makefile in the (lib) folder designed to turn all `*.c` and `*.h` pairs into part of the unified runtime library `libruntime.a`. An example of how to make a runtime function is provided bellow.

`functions.c`

```
#include "functions.h"

uint64_t factorial(uint64_t n) {
    uint64_t fact = 1;

    while (n > 0) {
```

(continues on next page)

(continued from previous page)

```
        fact *= n;
        n--;
    }

    return fact;
}
```

functions.h

```
#pragma once

#include <stdint.h>

uint64_t factorial(uint64_t n);
```

If ou compiler was compiling the following input `Input file`

```
3! + (2 + 7)!
```

Here is how to call the function in LLVM code.

LLVM src

```
target triple = "x86_64-pc-linux-gnu"
define i32 @main() {

    ; Calls factorial on 3 for the first part of expression
    %0 = call i64 @factorial(i64 3)

    ; Adds 2 and 7 together
    %1 = add i64 2, 7

    ; Calls factorial of (2 + 7)
    %2 = call i64 @factorial(i64 %1)

    ; Adds the results of 3! and (2 + 7)!
    %3 = add i64 %0, %2

    ; Done, return 0
    ret i32 0
}

; This makes the function available for calling
declare i64 @factorial(i64)
```

CHAPTER

TWENTYONE

COMPILER IMPLEMENTATION — PART 1

This section lists the portions of the *Gazprea* specification that must be implemented to complete the part 1 of the compiler implementation. All developers are advised to read the full specification for the language prior to start the implementation of Part 1 because decisions made while implementing Part 1 can make the implementation of Part 2 significantly more challenging. Thus, planning ahead for Part 2 is the recommended strategy.

1. *Comments*
2. *Types*
    - *Boolean*
    - *Character*
    - *Integer*
    - *Real*
    - *Tuple*
3. Type Support
    - *Type Qualifiers*
        - *Var*
        - *Const*
    - *Vector/Matrix Type Checking*
    - *Type Promotion*
    - *Type Casting*
    - *Type Inference*
    - *Typedef*
4. *Statements*
    - *Assignment Statements*
    - *Declarations*
    - *Globals*
    - *Block Statements*
    - *Loop*
        - *Break*
        - *Continue*

- *If/Else Statements*
- *Streams*
- *Functions*
- *Procedures*

5. *Expressions*
    - unary+, unary-, not
    - ^
    - *,/,%
    - +,-
    - <,>,<=,>=
    - and
    - or, xor
    - Variable references
    - Literal Values
    - Tuple reference
    - Function calls

CHAPTER

TWENTYTWO

COMPILER IMPLEMENTATION — PART 2

This section list the elements of the *Gazprea* specification that must be completed for the Part 2 of the compiler implementation. All the elements of Part 1 must have been completed because Part 2 builds on Part 1.

1. *All Previous Features*
2. *Types*
    - *Interval*
    - *Vectors*
    - *Matrix*
    - *String*
3. *Statements*
    - *Iterator Loop*
4. *Expressions*
    - *Operators*
    - *Generators*
    - *Filters*
5. *Built In Functions*
    - *Reverse*
    - *Rows and Columns*
    - *Length*
    - *Stream State*
6. *Memory Management*

CHAPTER

TWENTYTHREE

ERRORS

Your implementation is required to report both compile-time and runtime errors.

23.1 Compile-time Errors

Compile-time errors must be handled by throwing C++ standard exceptions.

You should create different exception classes for each of the different kinds of compile-time errors you report, such as a Type Error shown in the example below.

You must create all your exception classes in a single header file `exceptions.h` and extend `std::exception`.

Example exception class:

```
/* exceptions.h */

#include <string>
#include <sstream>

class TypeError : public std::exception {
private:
    std::string msg;
public:
    TypeError(std::string lhs, std::string rhs, int line) {
        std::stringstream sstream;
        sstream << "Type error: Cannot convert between "
                << lhs << " and " << rhs << " on line " << line << "\n";
        msg = sstream.str();
    }

    virtual const char* what() const throw() {
        return msg.c_str();
    }
};
```

Whenever you encounter an error, you throw an appropriate exception. To throw an exception, use the `throw` keyword. As an example for the exception defined above, we throw it as follows:

```
throw TypeError("int", "char", 10);
```

23.1.1 Syntax Errors

Syntax errors are also compile-time errors. ANTLR handles syntax errors automatically, but you are required to override the behavior and throw your own exception from `exceptions.h`.

Example:

```
/* exceptions.h */

#include <string>
#include <sstream>

class SyntaxError : public std::exception {
private:
    std::string msg;
public:
    SyntaxError(std::string msg) : msg(msg) {}

    virtual const char* what() const throw() {
        return msg.c_str();
    }
};
```

```
/* main.cpp */

class MyErrorListener : public antlr4::BaseErrorListener {
    void syntaxError(antlr4::Recognizer *recognizer, antlr4::Token * offendingSymbol,
                     size_t line, size_t charPositionInLine, const std::string &msg,
                     std::exception_ptr e) override {
        std::vector<std::string> rule_stack = ((antlr4::Parser*) recognizer)->
↪getRuleInvocationStack();
        // The rule_stack may be used for determining what rule and context the error␣
↪has occurred in.
        // You may want to print the stack along with the error message, or use the␣
↪stack contents to
        // make a more detailed error message.

        throw SyntaxError(msg); // Throw our exception with ANTLR's error message. You␣
↪can customize this as appropriate.
    }
};

int main(int argc, char **argv) {

    ...

    gazprea::GazpreaParser parser(&tokens);

    parser.removeErrorListeners(); // Remove the default console error listener
    parser.addErrorListener(new MyErrorListener()); // Add our error listener

    ...
}
```

For more information regarding the handling of syntax errors in ANTLR, refer to chapter 9 of The Definitive ANTLR 4 Reference.

23.2 Runtime Errors

Since the runtime library is written in C, you do not have access to C++ standard exceptions.

Instead, you are required to have a single header file `errors.h` containing all your functions which print error messages to `stderr` and exit.

Simply call any of the functions when you need to report an error.

Example:

```
/* errors.h */

#include <stdlib.h>

void sizeMismatchError() {
    fprintf(stderr, "Size mismatch error: Can not operate between two vectors or␣
↪matrices of differing size");
    exit(1);
}
```