gazprea

cmput415

Jun 11, 2025

CONTENTS

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

CHAPTER

ONE

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
- format
- function
- if
- in
- integer
- length
- loop
- not
- or
- procedure
- real
- return
- returns
- reverse

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

CHAPTER

TWO

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

THREE

NAMESPACE

Identifiers are used by variables, functions and procedures.

These share the same namespace in a scope and shadowing is possible between these types.

```
function x() returns integer; // "x" refers to this function in the global scope

procedure main() {
  integer x = 3; // "x" refers to this variable in the scope of main
}
...
```

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. It 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

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 setting the value to zero.

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[10] v = 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 */
}
```

5.1 Special cases

Special cases of declarations are covered in their respective sections.

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

CHAPTER

SIX

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.

6.1 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`.

6.2 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.

6.3 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

SEVEN

TYPES

7.1 Boolean

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

7.1.1 Declaration

A `boolean` value is declared with the keyword `boolean`. If the variable is not initialized explicitly, it is set to `false` (zero).

7.1.2 Literals

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

- `true`
- `false`

7.1.3 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`

7.1.4 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.

7.2 Character

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

7.2.1 Declaration

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

7.2.2 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`

7.2.3 Operations

The following operations are defined between `character` values.

Class	Operation	Symbol	Usage	Associativity
Grouping	parentheses	`()`	`(character)`	N/A
Comparison	equals	`==`	`character == character`	left
	not equals	`!=`	`character != character`	left

Scalar values with type `character` may be concatenated onto values with type `string` or arrays with type `character`.

7.2.4 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.

7.3 Integer

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

7.3.1 Declaration

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

7.3.2 Literals

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

```
1234
2
0
```

7.3.3 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	`==` `!=`

7.3.4 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.

7.4 Real

A `real` is an IEEE 754 32-bit floating point value. A `real` can be represented by a `f32` in *MLIR*.

7.4.1 Declaration

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

7.4.2 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
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 character `e` or `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
```

7.4.3 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), infinity(infs), and zeros.

7.4.4 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.

7.5 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.

7.5.1 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.

7.5.2 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. Spaces are not allowed inbetween elements in dot notation. Field indices *start at one*, not zero. For example:

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

Tuple access can both be used to retrieve the element value for an expression as well as to assign a new value to the element.

```
y = x + t1.1;     // Allowed
t1.1 = type-expr; // Allowed
```

7.5.3 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]);
```

7.5.4 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. Two tuples are equal when for every expression pair, the equality operator returns true. Two tuples are unequal when one or more expression pairs are unequal or the sizes mismatch. 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`

7.5.5 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.

7.6 Arrays

Arrays are fixed size collections, where each element of the array has the same type. Arrays can contain any of *Gazprea*'s base types (`boolean`, `integer`, `real`, and `character`) or compound type `tuple`.

7.6.1 Declaration

Aside from any type specifiers, the element type of the array is the first portion of the declaration. An array 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 array that is initialized with an integer value the integer will be promoted to a real value, and then used as a scalar initialization of the array. Be careful about type inference! If the type of the array is being inferred from the right had side, the previous example would create an `integer` array instead of a `real` array.

1. Explicit Size Declarations

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

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

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

   If the array is given a scalar value (`type-expr`) of the same element type then the scalar value is duplicated for every single element of the array.

   An array may also be initialized with another array. If the LHS array is initialized using a RHS array that is too small then the LHS array will be padded with zeros. However, if the LHS array is initialized with a RHS array that is too large then a `SizeError` should be thrown at compile-time or run-time. Check the *Compile-time vs Run-time Size Errors* section to know when you should throw the error.

2. Inferred Size Declarations

   If an array 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 array on the right-hand side is known.

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

3. Inferred Type and Size

   It is also possible to declare an array with an implied type and length using the var or const 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 array.

   ```
   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.

7.6.2 Construction

An array 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 array using this method of construction.

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

Gazprea **DOES** support empty arrays.

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

7.6.3 Operations

1. Array Operations and functions
    a. length

       The number of elements in an array 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 array `v` contains 3 elements.

    b. Concatenation

       Two arrays with the same element type may be concatenated into a single array 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 arrays 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 array `v` would be promoted to a real array 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 arrays.

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

An interesting corollary to array-scalar concatenation is that two scalars can be concatenated to produce an array:

```
integer[3] v = 1 || 2 || 3; // produces [1, 2, 3]
```

c. Dot Product

Two arrays 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. Range

The `..` operator creates an integer array holding the specified range of integer values. This operator must have an expression resulting in an integer on both sides of it. These integers mark the *inclusive* upper and lower bounds of the range.

For example:

```
1..10 -> std_output;
(10-8)..(9+2) -> std_output;
```

prints the following:

```
[1 2 3 4 5 6 7 8 9 10]
[2 3 4 5 6 7 8 9 10 11]
```

The number of integers in a range may not be known at compile time when the integer expressions use variables. In another example, assuming at runtime that `i` is computed as -4:

```
i..5 -> std_output;
```

prints the following:

```
[-4 -3 -2 -1 0 1 2 3 4 5]
```

Therefore, it is *valid* to have bounds that will produce an empty array because the difference between them is negative.

d. Indexing

An array may be indexed in order to retrieve the values stored in the array. An array may be indexed using integers. *Gazprea* is 1-indexed, so the first element of an array 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 = [4,5,6][3] /* y == 6 */
```

Like Python, *Gazprea* allows negative indices, which are interpreted as starting from the _back_ of the array instead of the front:

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

Out of bounds indexing should cause an error.

e. Stride

The by operator is used to specify a step-size greater than 1 when indexing across an array. It produces a new array with the values indexed by the given stride. 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] */
integer[*] s = v by 4; /* [1] */
```

d. Slices

An array may be indexed by a range to create a new array that is a *slice* of the original.

```
integer[*] a = 0..10 by 2; /* a = [0, 2, 4, 6, 8, 10] */
integer x = a[2..4]; /* x == [2, 4, 6] */
```

Note that for slices only a stride of 1 is allowed. For indexing purposes three additions are made to range syntax:

	Interpretation
	all elements
i..	ith to nth elements
..-i |	first to n-i-1th elements

Examples:

```
integer[*] a = 0..10 by 2; /* a = [0, 2, 4, 6, 8, 10] */
integer x = a[..4]; /* x == [0, 2, 4, 6] */
integer y = a[4..]; /* x == [6, 8, 10] */
integer z = a[..-1]; /* x == [0, 2, 4, 6, 8] */
```

2. Operations of the Element Type

Unary operations that are valid for the Element type of an array may be applied to the array in order to produce an array whose result is the equivalent to applying that unary operation to each element of the array. 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 array may be also applied to two arrays. When applied to two arrays of the same size, the result of the binary operation is a array formed by the element-wise application of the binary operation to the array operands.

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

Attempting to perform a binary operation between two arrays of different sizes should result in a `SizeError`.

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

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

Additionally the element types of arrays may be promoted, for instance in this case the integer array must be promoted to a real array 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 array, an equality operation checks whether or not all of the elements of two arrays 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 array. The result is the logical negation of the result of the == operator.

7.6.4 Type Casting and Type Promotion

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

7.7 Vectors

Vectors are language supported objects that allow for dynamically sized arrays. Once created, vectors in *Gazprea* behave exactly like arrays: they can be intermixed with arrays in expressions; they can be use on the RHS of array declarations and initializations; and they can be passed as array arguments to subroutines and functions.

7.7.1 Declaration

Vectors are declared and (optionally) initialized as follows. (Note that we have replaced <> with | in the notation below since the literals < and > are used in the declaration)

```
Vector<|type|> |identifier|;
Vector<|type|> |identifier| = |type-expr|;
Vector<|type|> |identifier| = |type-array|;
```

Unlike the array type, *Gazprea* vectors do not have an explicit size specifier, often called *capacity* in other languages.

```
Vector<character> vec = ['a', 'b', 'c'];
var Vector<integer> ivec;
Vector<real[*]> ragged_right = [[1.0], [2.0, 2.0]];
const Vector<character> const_vec = vec;
```

As a language supported object, *Gazprea* provides several methods for `Vector`:

- push() - pushes a new element to the back of the vector
- len() - number of elements in the vector
- append - append another array slice to the vector

```
Vector<tuple(bool, integer)> tvec;
tvec.push((false, 0));
tvec.append((true, 1));
tvec[tvec.len()] -> std_output; // prints (true, 1)
```

7.7.2 Operations

Operations on `Vectors` are identical syntactically and semantically to operations on arrays. In particular, operand lengths must match for binary expressions and dot product.

A `Vector` or vector slice can be passed as a call argument that has been declared as an array slice of the same size and type.

When indexing a vector of arrays, the first index selects the array element within the vector, and the second index selects the element within the array:

```
Vector<real[*]> ragged_right = [[1.0], [2.1, 1.2]];
length(ragged_right[1]) -> std_output; // prints 1
ragged_right[2][2] -> std_output; // prints 1.2
```

7.8 String

A `String` is another object within *Gazprea*. Fundamentally, a `String` is a `Vector` of `character`. This means that, like a vector, a string behaves like a dynamically sized array, but because it is an object *Gazprea* can provide type specific features.

`String` vectors behave a lot like character arrays, but there are several differences between the two types: an *extra literal style*, the *result of a concatenation* and *behaviour when sent to an output stream*.

7.8.1 Declaration

A string may be declared with the keyword `String`. The same rules of *vector declarations* also apply to strings, which means that all lenghts are inferred:

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

7.8.2 Literals

Strings can be constructed in the same way as arrays 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"
```

Although strings and character arrays look similar, they are still treated differently by the compiler:

```
character[*] carray = ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\n'];
carry -> std_output;
String vec = carray;
vec -> std_output;
```

prints:

```
[h e l l o   w o r l d
]
hello world
```

7.8.3 Operations

As character arrays, strings have all of the same operations defined on them as the other array data types, Remember that because a `String` and array 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 an array of `character`. Note that because a `String` is a type of `Vector`, concatenation may also be accomplished with `concat` and `push` methods:

```
String letters = ['a', 'b'] || "cd";
letters.concat("ef");
letters.push('g');
letters  -> std_output;
```

prints the following:

```
abcdefg
```

7.8.4 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.

7.9 Matrices

Gazprea supports two dimensional matrices as arrays of arrays. Although the syntax and concepts are easily generalizable to many dimensions, we are restricting the language to two dimensions for now.

7.9.1 Declaration

Matrix declarations are similar to array 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]];
```

7.9.2 Construction

A 2D matrix can be viewed as an array of arrays. The elements in each array form a single row of the matrix. All rows with fewer elements than the row of maximum row length are padded with zeros 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 zero. 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. Initializing with a scalar value makes every element of the matrix equal to the scalar.

Gazprea supports empty matrices.

```
integer[\*][\*] m = []; /* Should create an empty matrix */
```

7.9.3 Operations

Multi-dimensional arrays have binary and unary operations of the element type defined in the same manner as unidimensional arrays. Unary operations are applied to every element of the matrix, and binary operations are applied between elements with the same position in the arrays.

The operators ==, and != also have the same behavior independent of the dimensionality of the array. These operations compare whether or not **all** elements of are equal.

Two dimensional arrays 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 matrix multiplication. Specifically, the number of columns of the first operand must equal the number of rows of the second operand, e.g. an $m \times n$ matrix multiplied by an $n \times p$ matrix will produce an $m \times p$ matrix. If the dimensions are not correct a `SizeError` should be raised.

Arrays of any dimension support the built in functions `rows` and `columns`, which when passed a 2D array 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 array indexing, however, two indices must be used. Because matrices are arrays of arrays the indexing is coposite:

```
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 must be integers.

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

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

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

7.9.4 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

EIGHT

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.

CHAPTER

NINE

TYPE CASTING

Gazprea provides explicit type casting. Type casting is an expression. 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)
```

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

9.1 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	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

9.2 Scalar to Array

A scalar may be promoted to an array of any dimension with an element type that the original scalar can be cast to according to the rules in *Scalar to Scalar*. A scalar to array cast *must* include a size with the type to cast to as this cannot be inferred from the scalar value. For example:

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

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

9.3 Array to Array

Conversions between array types are also possible. First, the values of the original are cast 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 zero or truncated to match the destination type size. Note that the size is not required for array to array 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 array to an integer array.
integer[3] u = as<integer[*]>(v);

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

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

Casting non-variable empty arrays `[]` is not allowed, because a literal empty array does not have a type.

9.4 Multi-dimensional Arrays

Conversions between arrays of any dimension are possible. The process is exactly like *Array to Array* except padding and truncation can occur in all 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);
```

9.5 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);
```

CHAPTER

TEN

TYPE PROMOTION

Type promotion is a sub-problem of casting and refers to casts that happen implicitly.

Any conversion that can be done implicitly via promotion can also be done explicitly via typecast expression. The notable exception is array promotion to a higher dimension, which occurs as a consequence of scalar to array promotion.

10.1 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

10.2 Scalar to Array

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

The scalar will be implicitly converted to an array 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]
```

Other examples:

```
1 == [1, 1]  // True
1..2 || 3 // [1, 2, 3]
```

Note that an array can never be downcast to a scalar, even if type casting is used. Also note that matrix multiply imposes strict requirements on the dimensionality of the the operands. The consequence is that scalars can only be promoted to a matrix if the matrix multiply operand is a square matrix ($m \times m$).

10.3 Multi-dimensional Arrays

Array promotion to a higher dimension occurs because, for example, a row in a 2D array is equivalent to a 1D array. When a 2D array is initialized or operated on with a 1D array, each scalar element in the 1D array is interpreted as a row. By applying the scalar to array promotion rule, each scalar element in the array will be promoted to a row. The example below demonstrates scalar to row promotion, row padding and column padding all together.

```
integer[3][4] m1 = [1,[1,2,3]];
// m1 = [[1, 1, 1, 1], [1, 2, 3, 0], [0, 0, 0, 0]]
```

The number of columns in each row are inferred, first by the expression indicating the column size in the type declaration, and second, if the columnn size is inferred as *, by the size of the array literal or expression. Therefore, when the column size is infered, an array to matrix promotion always produces a square matrix.

```
integer[2][*] m2 = [3, 4];
// m2 = [[3, 3], [4, 4]]
```

Array promotions to a higher dimensionality apply in all contexts where operations on arrays are defined.

10.4 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);
```

10.5 Character Array to/from String

A `string` can be implicitly converted to an array 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

ELEVEN

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 can conflict with symbol names. The following is therefore legal:

```
typedef character main;
typedef integer i;

const main A = 'A';

procedure main() returns i {
  i i = 0;
  return i;
}
```

In addition to base types, `typedef` can be used with arrays, strings and tuples. Using `typedef` on tuples, or on arrays with sizes helps reusability and consistency:

```
typedef tuple(string[64], integer, real) student_id_grade;
student_id_grade chucky_cheese = ("C. Cheese", 123456, 77.0);

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 arrays with inferred sizes are allowed, but declarations of variables using the typedef must be initialized appropriately.

Because `typedef` is an aliased name for a type, you can use `typedef` on typedef'ed types:

```
typedef integer int;
typedef int also_int;
```

Duplicate `typedef` should raise a *SymbolError*

```
typedef integer ty;
typedef character ty;
```

Some typedefs may be parameterized with an expression, such as with arrays, such expressions are restricted to be composed exclusively from arithmetic operations on scalar literals. Practically speaking, this requires constant folding but *not* constant propogation.

```
typedef integer[1 + 3 - 2] vec_of_two;
procedure main() returns integer {
  vec_of_two v = 1..3;
}
```

Should raise a `SizeError` on line 3 since the `vec_of_two` type has a size of 2 and an array of size 3 is being assigned.

CHAPTER

TWELVE

STREAMS

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

12.1 Output Stream

Output streams use the following syntax:

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

12.1.1 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*: Prints T for true, and F for false.

Arrays 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 an array of arrays. 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, procedures with no return type and tuples cannot be sent to streams. Also, empty arrays and matrices can be send to streams, but not empty literals (e.g. `[]`), because they have no type.

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;
```

12.2 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.

Type promotion is not performed for stream input over any type.

12.2.1 Input Semantics

`std_input` expects an input stream of values which do not need to be whitespace separated. A read will consume the stream until a character or EOF occurs that breaks the pattern match for the given types specifier. The longest successful match is returned.

In general input stream semantics are designed for parity with `scanf`. The only differences are the *Stream State*, a boolean specifier and a restriction on the maximum number of bytes that can be consumed in a single read to 512.

For each of the allowed types the semantics are given below.

Reading a `character` from stdin consumes the first byte that can be read from the stream. If the end of the stream is encountered, then a value of `-1` is set. There is no concept of skipping whitespace for characters, since space and escaped characters must be readable.

An `integer` from stdin can take any legal format described in the *integer literal* section. It may also be preceded by a single negative or positive sign. All preceeding whitespace before the number or sign character may be skipped up to the limit imposed by the 512 byte read restriction.

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. Preceeding whitespace may be skipped in the same way as integers.

A `boolean` input from stdin is either `T` or `F`. Preceeding whitespace may be skipped in the same way as integers and reals.

For the following program:

```
boolean b;
character c;
integer i;
real r;
b <- std_input;
i <- std_input;
c <- std_input;
r <- std_input;
format(b) || " " || format(r) -> std_output;
```

And this input (where '\t' is TAB, '*' is space, and each line ends with a newline ('\n'):

```
\tF\n
1\n
*1.\n
```

The output would be:

```
F 1.0
```

because the white space is consumed for characters and skipped for other types.

12.2.2 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.

When an error occurs the the null value is assigned and the input stream remains pointing to the same position as before the read occured.

The program below demonstrates 4 reads which set the error states 1,0,0,2 respectively.

```
integer ss;
integer i;
boolean b;
character c;

i <- std_input;
i -> std_output;
ss = stream_state(std_input);
ss -> std_output;

c <- std_input; //eat the .
```

(continues on next page)

(continued from previous page)

```
i <- std_input;
i -> std_output;

c <- std_input;
ss = stream_state(std_input);
ss -> std_output;
```

With the input stream:

```
.7
```

And the expected output:

```
0172
```

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`	2
Integer	error	`0`	1
	end of stream	`0`	2
Real	error	`0.0`	1
	end of stream	`0.0`	2

CHAPTER

THIRTEEN

EXPRESSIONS

Expressions can only exist within a statement or another expression.

13.1 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

13.2 Generators

A generator may be used to construct either a one or two dimensional array. A generator creates a value of a 1D array type when one domain variable is used, and a 2D array 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 expressions, and an additional expression on the right hand side of the bar (|). This additional expression is used 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. Let T be the type of the expression to the right of the bar (|). Then, if the domain of the generator is an array of size N, the result will be a array of size N with element type T. Otherwise, if the domain of the generator is a matrix of size N x M, the result will be a matrix of size N x M with element type T. 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. */
```

13.3 Domain Expressions

Domain expressions consist of an identifier denoting an iterator variable and an expression that evaluates to **any** array type. Domain expressions can only appear within iterator loops and generators. A domain expression is a way of declaring a variable that is local to the loop or generator, that takes on values from the domain expression array in order. Domain expressions must evaluate to a type, which means empty literal arrays yield a TypeError. The scope of the domain variables (the left hand side of the declaration) is within the body of the generator or loop. The domain expressions (the right hand side) are all evaluated before any of the domain variables are initialized, and therefore the domain expression scope is the one enclosing the iterator loop or generator.

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, such as in matrix generators.

```
integer i = 2;

/* The "i"s both domain expressions are at the same scope, which is
 * the one enclosing the loop. Therefore the matrix is: [[0 0 0] [0 1 2] [0 2 4]]
 */
integer[3,3] = [ i in 0..i, j in 0..i | i*j ];
```

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 as well.

Iterator variables can be assigned to and re-declared within the enclosed iterator loop. The variable is re-initialized according to the expression each iteration.

```
loop i in 1..6 {
  integer i = 5;
}
```

CHAPTER

FOURTEEN

STATEMENTS

14.1 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 arrays and tuples. With arrays indices may be provided in order to change the value of an array element. In Gazprea, arrays cannot be indexed with array expressions. For instance, with single dimensional arrays:

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

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

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

This applies to arrays of any dimension.

```
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]] */
```

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. This assignment is performed left-to-right.

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;

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 arrays. You must ensure that values can not be aliased with an assignment between any types, including arrays 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] */
```

14.2 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`.

14.3 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* require the conditional expression to be enclosed in parentheses.

```
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
```

(continues on next page)

(continued from previous page)

```
   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. */
```

14.4 Loop

14.4.1 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;
}
```

14.4.2 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);
```

14.4.3 Iterator Loop

Loops can be used to iterate over the elements of an array 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 an array, each time the loop is executed the next element of the array is assigned to the domain variable. The elements of the domain array are assigned to the domain variable starting from index 1, and going up to the final element of the array. When all of the elements of the domain array have been used the loop automatically exits. For instance:

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

Array ranges can also be used instead:

```
// 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
   array. */
loop i in v {
  v = 0;
  i -> std_output; "\n" -> std_output;
}
```

Similarly, the domain variable is assigned from the domain array at the top of the loop for every iteration, even if it is reassigned in the body of the loop:

```
// This will print 123456
loop i in 1..6 {
  i -> std_output;
  i = 5;
}
```

Note that multiple domain expressions are *not* allowed:

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

// If you want multiple domains, use a nested loop
loop i in u {
  loop j in v {
    "Hello!\n" -> std_output;
```

(continues on next page)

(continued from previous page)

```
  }
}
```

14.5 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.

14.6 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;
}
```

14.7 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;
}
```

14.8 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;
x <- std_input; /* Read an integer into x */
```

1.. _sec:function:

CHAPTER

FIFTEEN

FUNCTIONS

A function in *Gazprea* has several requirements:

1. All of the arguments are implicitly `const`, and can not be mutable.
2. Function arguments cannot contain type qualifiers. Including a type qualifier with a function argument should result in a `SyntaxError`.
3. Argument types must be explicit. Inferred size arrays are allowed
4. Functions can not perform any I/O.
5. Functions can not rely upon any mutable state outside of the function.
6. Functions can not call any procedures.
7. Functions must be declared in the global scope.

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.

15.1 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);

/* 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;
  }
}
```

(continues on next page)

(continued from previous page)

```
/* 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() returns integer {
  integer x = 0;
  loop {
    x = x + 1;
  }

  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.

15.2 Function Prototypes

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, and allows for multiple compilation units if the function definitions are in different source files.

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

procedure main() returns integer {
  integer y = f(13, 2);
  /* Can use this in main, even though the definition is below */
  return 0;
}

function f(integer x, integer z) returns integer = x*z;
```

Note that only the type signatures of the forward declaration of the function and the definition must be identical. That means the argument names in the prototype are *optional*. If the prototype arguments are given names they do not have to match the argument names in the function definition.

15.3 Array 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 */
}
```

Like Rust, array *slices* may be passed as arguments:

```
function to_real_vec(integer[\*] x) returns real[\*] {
   real[\*] rvec = x;
   return rvec;
}

function slicer() returns real[\*] {
    integer a[10] = 1..10;
    Vector<real> two_halves = to_real_vec(a[1..5]);
    two_halves.append(to_real_vec(a[6..]));
    return two_halves;
}
```

Remember that all function parameters are `const` in *Gazprea*, so that all functions are pure. That means that while it is legal to pass arrays and slices *be reference*, the array contents cannot be modified inside the function, because the change would be visible outside the function. You must check that the `const` requirement is honored.

CHAPTER

SIXTEEN

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` that 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 the RHS of declaration statements, RHS of assignment statements or as the procedure being called in a call statement.
- When used within a valid statement, the only legal operators which can be applied to a procedure call are unary operators and casts. Additionally, the result of the call may not be used in the direct construction of a type that does not match the return type of the procedure.

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.

16.1 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;
```

(continues on next page)

(continued from previous page)

```
  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 in a `call` statement would not do anything. *Gazprea* should raise an error if a function is used in a `call` statement.

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 example:

```
/* 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 */
```

These restrictions are made by *Gazprea* in order to allow for more optimizations.

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 */
integer x = p(); /* Illegal */
```

16.2 Procedure Declarations

Procedures can use *forward declaration* just like functions.

16.3 Main

Execution of a *Gazprea* program starts with a procedure called `main`. This procedure takes no arguments, and has an integer return type. `main` is called exclusively by the operating system, and the return value is used by the operating system, so if you are using multiple compilation units one and only one compilation unit must define `main`.

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

(continues on next page)

(continued from previous page)

```
  x = x + x;
  x -> std_output;

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

16.4 Type Promotion of Arguments

Argument types can be promoted at call time, but only if the argument is call by value (`const`). The reason is that mutable arguments are effectively call by reference, and are therefore *l-values* (pointers).

```
procedure byvalue(string x) returns integer {
  return len(x);
}
procedure byreference(var string x) returns integer {
  return len(x);
}
procedure main() returns integer {
  character[3] y = ['y', 'e', 's'];

  integer size = byvalue(y); // legal
  call byreference(y);       // illegal

  return 0;
}
```

16.5 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. */
```

(continues on next page)

(continued from previous page)

```
  /* 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 arrays are passed toprocedures. For instance:

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

It is impossible to tell whether or not these overlap at compile time due to the halting problem. Thus for simplicity, whenever an array is passed to a procedure *Gazprea* detects aliasing whenever the same array is used, regardless of whether or not the access would overlap.

Another instance of aliasing relates to tuples, such as passing the same tuple twice in one procedure, or passing the entire tuple and separately passing a single tuple field. In both cases this can cause aliasing.

```
call p(t1, t1.1);
/* p is some procedure with a tuple argument and a real argument */
```

16.6 Array Parameters and Returns

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

Similarly, slices can be used whereever arrays are declared as parameters, and unlike functions, array parameters in procedures can be `var`.

CHAPTER

SEVENTEEN

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

EIGHTEEN

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 arrays of any element type. Normally a function must specify the element type of an array argument.

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.

18.1 Length

`length` takes an array of any element type, and returns an integer representing the number of elements in the array.

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

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

18.2 Shape

The built-in `shape` operates on arrays of any dimension, and returns an array listing the size of each dimension.

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

shape(M) -> std_output; /* Prints [2, 3] */
```

18.3 Reverse

The reverse built-in takes any array, and returns a reversed version of it.

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

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

18.4 Format

The `format` built-in takes any scalar as input and returns a string containing the formatted value of the scalar.

```
integer i = 24;
real r = 2.4;

"i = " || format(i) || ", r = " || format(r) || '\n' -> std_output;
// Prints: "i = 24, r = 2.4\n"
```

Note that `format` will have to allocate space to hold the return string. You will have to figure out how to manage the memory so it is reclaimed eventually.

18.5 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.

`stream_state` is initialized to `0`, which is the value return if no read has been issued.

```
boolean b;
integer i;

// Input stream: 9
b <- std_input;               // b = false (error reading boolean)
i = stream_state(std_input); // i = 1     (last read was error)
i <- std_input;               // i = 9     (successfully read integer)
i = stream_state(std_input); // i = 0     (last read was success)
b <- std_input;               // b = false (read end of stream)
i = stream_state(std_input); // i = 2     (last read was end of stream)
```

The input stream is described in more detail in the *input stream* section.

CHAPTER

NINETEEN

BACKEND

You don't need to implement an interpreter for Gazprea. You only need to implement a *MLIR* code generator that outputs *LLVM IR*.

19.1 Memory Management

It is important that you are able to automatically free and allocate memory for arrays when they enter and exit scope. You could allocate them on the stack, but this could be problematic if the arrays are very large. It is likely safer to use `malloc` and `free` for these purposes. This may be done in either your runtime or directly within MLIR.

Below is an example of how to use `malloc` and `free` within MLIR using the LLVM dialect:

```
module {
  llvm.func @malloc(i32) -> !llvm.ptr<i8>
  llvm.func @free(!llvm.ptr<i8>)
  llvm.func @main() -> i32 {
    %0 = llvm.mlir.constant(128 : i32) : i32
    %1 = llvm.call @malloc(%0) : (i32) -> !llvm.ptr<i8>
    llvm.call @free(%1) : (!llvm.ptr<i8>) -> ()
    %c0_i32 = llvm.mlir.constant(0 : i32) : i32
    llvm.return %c0_i32 : i32
  }
}
```

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

19.2 Runtime Libraries

If you make a runtime library, the runtime library must be implemented in a runtime directory (`runtime`). 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 `runtime` 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"
```

(continues on next page)

(continued from previous page)

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

    while (n > 0) {
        fact *= n;
        n--;
    }

    return fact;
}
```

functions.h

```
#pragma once

#include <stdint.h>

uint64_t factorial(uint64_t n);
```

If your compiler was compiling the following input

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

Here is how to call the function in the LLVM dialect of MLIR:

MLIR src

```
 module {
   // This makes the function available for calling
   llvm.func @factorial(i64) -> i64

   llvm.func @main() -> i32 {
     // Calls factorial with the constant 3 as an argument
     %0 = llvm.mlir.constant(3 : i64) : i64
     %1 = llvm.call @factorial(%0) : (i64) -> (i64)

     // Adds 2 and 7 together
     %2 = llvm.mlir.constant(2 : i64) : i64
     %3 = llvm.mlir.constant(7 : i64) : i64
     %4 = llvm.add %2, %3 : i64

     // Calls factorial with the result of 2+7
     %5 = llvm.call @factorial(%4) : (i64) -> (i64)

     // Adds the result of 3! with (2+7)!
     %6 = llvm.add %1, %5 : i64

     // Done, return 0
     %c0_i32 = llvm.mlir.constant(0 : i32) : i32
     llvm.return %c0_i32 : i32
   }
}
```

CHAPTER

TWENTY

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*
    - *Type Promotion*
    - *Type Casting*
    - *Type Inference*
    - *Typedef*
4. *Statements*
    - *Assignment Statements*
    - *Declarations*
    - *Globals*
    - *Block Statements*
    - *Loop*
        - *Break*
        - *Continue*
    - *If/Else Statements*

- *Streams*
- sec:function
- *Procedures*

5. *Expressions*
    - unary+, unary-, not
    - ^
    - *,/,%
    - +,-
    - <,>,<=,>=,==,!=
    - and
    - or, xor
    - Variable references
    - Literal Values
    - Tuple reference
    - Function calls
6. *Errors*
    - SyntaxError
    - SymbolError
    - TypeError
    - AliasingError
    - AssignError
    - MainError
    - ReturnError
    - GlobalError
    - StatementError
    - CallError
    - DefinitionError
    - MathError

CHAPTER

TWENTYONE

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*
    - *Arrays*
    - *Matrices*
    - *String*
3. *Statements*
    - *Iterator Loop*
4. *Expressions*
    - *Operators*
    - *Generators*
5. *Built In Functions*
    - *Reverse*
    - *Shape*
    - *Length*
    - *Format*
    - *Stream State*
6. *Memory Management*
7. *Errors*

CHAPTER

TWENTYTWO

ERRORS

Your implementation is required to report both compile-time and run-time errors. You must use the exceptions defined in `include/CompileTimeExceptions.h` and the functions defined in `runtime/include/run_time_errors.h`. Do not modify these files, you can pass a string to a constructor/function to provide more details about a particular error. You must pass the corresponding line number to the exceptions for compile-time errors but not run-time errors. Do not create new errors. Your compiler is only expected to report the first error it encounters.

22.1 Compile-time Errors

Compile-time errors must be handled by throwing the exceptions defined in `include/CompileTimeExceptions.h`. To throw an exception, use the `throw` keyword.

```
throw MainError(1, "program does not have a main procedure");
```

Here are the compile-time errors your compiler must throw:

- `SyntaxError`

  Raised during compilation if the parser encounters a syntactic error in the program.

- `SymbolError`

  Raised during compilation if an undefined symbol is referenced or a defined symbol is re-defined in the same scope.

- `TypeError`

  Raised during compilation if an operation or statement is applied to or betweeen expressions with invalid or incompatible types.

- `AliasingError`

  Raised during compilation if the compiler detects that mutable memory locations may be aliased.

- `AssignError`

  Raised during compilation if the compiler detects an assignment to a const value or a tuple unpacking assignment with the number of lvalues different than the number of fields in the tuple rvalue.

- `MainError`

  Raised during compilation if the program does not have a procedure named `main` or when the signature of `main` is invalid.

- `ReturnError`

Raised during compilation if the program detects a function or procedure with a return value that does not have a return statement reachable by all control flows. Control flow constructs may be assumed to always be undecideable, meaning they may branch in either direction.

If the subroutine has a `return` statement with a type that does not match the owning subroutine's type, the line number of the `return` statement should be reported, along with the name and (correct) type of the enclosing routine.

Note also that, strictly speaking, this is a type error, not a return error. If the procedure/function is missing a `return` statement, then the line number of the subroutine declaration should be printed instead.

- `GlobalError`

  Raised during compilation if the program detects a `var` global declaration, a global declaration without an initializing expression, a global declaration with an invalid initializing expression or any statement that does not belong in the global scope.

- `StatementError`

  Raised during compilation if the program is syntactically valid but the compiler detects an invalid statement in some context. For example, `continue` or `break` outside of a loop body.

- `CallError`

  Raised during compilation if the procedure call statement is used to call a function. Also raised if a procedure is called in an invalid context. For example, a procedure call in an output stream expression.

- `DefinitionError`

  Raised during compilation if a procedure or function is declared but not defined.

- `LiteralError`

  Raised during compilation if a literal value in the program does not fit into its corresponding data type.

- `MathError`

  Raised during compile time expression evaluation when division by zero occurs. Conditions for raising are eqivalent to a runtime `MathError`.

- `SizeError`

  Raised during compilation if the compiler detects an operation or statement is applied to or between arrays with invalid or incompatible sizes. Read more about when a `SizeError` should be raised at run-time instead of compile-time in the *Compile-time vs Run-time Size Errors* section.

Here is an example invalid program and a corresponding compile-time error:

```
1 procedure main() returns integer {
2     integer x;
3 }
```

```
ReturnError on line 1: procedure "main" does not have a return statement reachable by␣
↪all control flows
```

22.1.1 Syntax Errors

ANTLR handles syntax errors automatically, but you are required to override the behavior and throw the `SyntaxError` exception from `include/CompileTimeExceptions.h`.

For example:

```
/* 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(line, 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.

22.2 Run-time Errors

Run-time errors must be handled by calling the functions defined in `runtime/include/run_time_errors.h`.

```
MathError("cannot divide by zero")
```

Here are the run-time errors you need to report:

- `SizeError`

  Raised at runtime if an operation or statement is applied to or between arrays with invalid or incompatible sizes. Read more about when a `SizeError` should be raised at compile-time instead of run-time in the *Compile-time vs Run-time Size Errors* section.

- `IndexError`

  Raised at runtime if an expression used to index an array is an `integer`, but is invalid for the array size.

- `MathError`

  Raised at runtime if either zero to the power of N, where N is <= 0, or a division by zero is evaluated.

- `StrideError`

  Raised at runtime if the `by` operation is used with a stride value <=`0`.

Here is an example invalid program and a corresponding run-time error:

```
1 procedure main() returns integer {
2     integer[3] x = [2, 4, 6];
3     return integer[4];
4 }
```

```
IndexError: invalid index "4" on array with size 3
```

22.3 Compile-time vs Run-time Size Errors

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

```
integer[2] vec = 1..10;
```

For simplicity, this section defines a subset of the size errors detectable at compile-time for which your compiler should report a `SizeError` at compile-time.

In particular, your compiler should raise a `SizeError` at compile-time if and only if it finds one of the following five cases:

1. An operation between arrays with compatible sizes such that
    1. each operand array expression is formed by operations on literal expressions, and
    2. the sizes of the operand arrays do not match.
2. An array declaration, found either in a regular declaration statement, function parameter binding or constant procedure parameter binding such that
    1. the expressions used to declare the size of the array are formed exclusively from arithmetic operations on scalar literals
    2. the declaration or parameter is initialized with an array expression with compatible type that is formed by arithmetic operations on scalar literals
    3. the size of the initialization expression is larger, in some dimension, than the declared size.
3. An array declaration statement such that
    1. the declaration has no declared size and
    2. there is no initialization expression.
4. An array declaration statement such that
    1. the declaration has no declared size,

2. the initialization expression has compatible type, and
3. the initialization expression is not an array.

5. A function call where
    1. The argument is an array literal
    2. The parameter type is the same type but with a different literal size.
6. A return statement where
    1. The value being returned is an array literal
    2. The return type of the function is the same type but with a different literal size.

Here are some example statements that should raise a compile-time `SizeError`:

```
[1, 2, 3] + [1.3] -> std_output;
```

```
[[1, 2], [3, 4]] % [[2, 2]] -> std_output;
```

```
integer[2] vec = [1, 2, 3] + 1;
```

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

```
integer[2] vec = 1..10;
```

```
character[*] vec;
```

```
boolean[*] vec = true;
```

```
real[*] vec = 3;
```

```
function f(integer[3] x) returns integer = 0;
integer y = f(1..2); // Case 5
```

```
function f() returns integer[3] = 1..2; // Case 6
```

Here are some example statements that should not raise a compile-time `SizeError` in your implementation, but may raise a run-time `SizeError`:

```
[1, 2, 3] + vec -> std_output;
```

```
integer[2] vec = [1, 2, 3] + scal;
```

```
integer[two] vec = [1, 2, 3];
```

22.4 More Examples

```
/* Indexes */
character[3] v = ['a', 'b', 'c']; // Indexing is harder than it looks!
integer i = 10;
v(3) = 'X'; // SyntaxError
v[i] = '?'; // Run-timeerror
v['a'] = '!'; // TypeError
i[1] = 1; // SymbolError

/* Tuples */
tuple (integerm integer) a = (9, 5);
integer b;
integer c;
integer d;
b, c, d = a; // AssignError
tuple(integer, integer, integer) z = a; // TypeError
```

22.5 How to Write an Error Test Case

Your compiler test suite can include error test cases. An error test case can include a compile-time or run-time error. In either case, the expected output should include exactly one line of text.

For compile time error tests, only one error should be present in the test case and exactly one line of expected output should catch it. The single line should include the error type and the line number on which it occurs. Below is an example:

```
var integer x = 0;

procedure main() returns integer {
  return 0;
}
```

```
GlobalError on line 1
```

Precisely defining the line number on which an error occurs can be difficult. Should the `AssignError` below occur on line 3, 6 or in between?

```
procedure main() returns integer {
    const integer i = 5;
    i
    =
    5
    ;
}
```

Test cases that deliberately make the line number ambiguous will be disqualified. If an obvious line number is not apparent, refer to the reference solution on the 415 compiler explorer.

For runtime errors, the line number is not required. Here is an example of a run-time error test case and the corresponding expected output file:

```
procedure main() returns integer {
  1..1 by 0 -> std_output;
  return 0;
}
```

```
StrideError
```

22.6 How to make the Tester Happy

For error test cases, the tester inspects the first line from `stderr`. Therefore, you must ensure that you do not pollute this stream with debug messages etc.

Additionally, the tester only knows to stop the toolchain prematurely if your program terminates with a non-zero exit code. Once you have caught an error make sure to return a non-zero exit code.

Finally, the tester is lenient towards the type given to a particular errror. Specifically the tester simply confirms that the substring "Error" is present and for compile time errors that the correct line is provided.

This leniency is motivated by the fact that sometimes determining which type to call an error is difficult. For example, it may be arguable that a `ReturnError` should be interpreted as a `TypeError` and vice versa as previously mentioned.