SFU CMPT 379
Compilers
Fall 2022
Milestone 2

Assignment due Friday, November 4, by 11:59pm.

For this assignment, you are to expand your Bilby-1 compiler from milestone 1 to handle Bilby-2. Project setup and acceptable submissions are the same as for assignment 1 (for example, submit your java src directory).

Bilby-2 is mainly backwards compatible with Bilby-1. All restrictions and specifications from Bilby-1 are still in force, unless specifically noted. The only part that is not backwards compatible with Bilby-1 is typechecking; some operator signatures that were not accepted in Bilby-1 are accepted in Bilby-2 because of promotion.

Language Bilby-2

Tokens:

integerLiteral → [**0..9**]$^+$ // has type "integer"
floatingLiteral → ([**0..9**]$^+$ **.** [**0..9**]$^+$) (**E** (**+**|**-**)$^?$ [**0..9**]$^+$)$^?$ // has type "floating"
booleanLiteral → **true** | **false** // has type "boolean"
stringLiteral → " [^"**\n**]* " // has type "string"
// In this *specification* only, not in bilby, **\n** denotes newline.
// **(**bilby-1 does not have character escapes in strings;
// **do not** interpret **$n**, **#n**, or **\n** in a string constant as a newline, for instance)
characterLiteral → **#α** |**##** [**0..9#**] |**#**[**0..7**]$^+$ // has type "character"
// **α** is any printable ascii character (encoding is decimal 32 to 126)

identifier → [**a..zA..Z_@**][**a..zA..Z_@0..9**]*

punctuator → *operator* | *punctuation*
operator → *unaryOperator* | *arithmeticOperator* | *comparisonOperator* | *booleanOperator*
unaryOperator → **+** | **–** | **!** | **length** | **low** | **high**
arithmeticOperator → **+** | **–** | ***** | **/**
comparisonOperator → **<** | **<=** | **==** | **!=** | **>** | **>=** | **in**
booleanOperator → **&&** | **||**

punctuation → **;** | **{** | **}** | **(** | **)** | **[** | **]** | **%**| **:=**

comment → **%** [^**%\n**]* (**%** |**\n**) // starts at %, ends at next % or newline.
// a comment cannot start inside a string.

Grammar:

S → **main** *blockStatement*
blockStatement → **{** *statement** **}**

statement → *declaration*
assignmentStatement
ifStatement
whileStatement
printStatement
blockStatement

declaration → **imm** *identifier* **:=** *expression* **;** // immutable "variable"
mut *identifier* **:=** *expression* **;** // mutable variable

assignmentStatement → *target* **:=** *expression* **;** // Reassignment. expr must be promotable to target type.
target → *expression* // must be a targetable expression.

printStatement → **print** *printExpressionList* **;** // print the expr values
printExpressionList → *printSeparator** (*expression printSeparator*+)* *expression*? // a separated list of expressions (can be zero of them)
printSeparator → **$| $n | $s | $t**

ifStatement → **if (***expression***)** *blockStatement* (**else** *blockStatement*)? // expression must be Boolean
whileStatement → **while (***expression***)** *blockStatement* // expression must be Boolean

expression → *unaryOperator expression*
expression operator expression // all binary operations left-associative
(*expression* **)** // targetable iff the expression is.
[*expression* **as** *type* **]** // casting
< *expression* **..** *expression* **>** // range creation
expression **[** *expression* **]** // array indexing.
// first *expression* must be array type; second one integer
arrayExpression
literal

type → *primitiveType* | *arrayType* | *rangeType*
primitiveType → **bool** | **char** | **string** | **int** | **float**
arrayType → **[** *type* **]** // an array of the given type
rangeType → **<** *type* **>** // a range of the given type. Type must be **char, int,** or **float**.

arrayExpression → **alloc** *arrayType* **(** *expression* **)**
[*expressionList* **]** // expressions must all be promotable to same type

literal → *integerLiteral* | *floatingLiteral* | *booleanLiteral* | *characterLiteral* | *stringLiteral*
identifier // identifier can be targetable.
expressionList → *expression* (**,** *expression*)*

1. Boolean expressions

The boolean-and operator **&&** has the signature (boolean, boolean) -> boolean.
The boolean-or operator **||** also has the signature (boolean, boolean) -> boolean.

Both of these operators perform short-circuit evaluation of their operands from left to right. This means that if the outcome of the operation is known after evaluating the left operand, then the right operand will not be evaluated.

The prefix unary boolean-not operator **!** has the signature (boolean) -> boolean.

2. Control flow

The control-flow statements (**if** and **while**) work as they do in most block-oriented languages (such as java and C++,) but note that they only take blockStatements as their clauses. In other words, one has to use the curly-braces **{** and **}** around the subordinate code, even if it is only one statement.

The **while (** *condition* **)** *body* loop checks the condition before entering the loop body, and may therefore execute the body zero times (if the condition is false upon statement entry).

3. Targetable expressions

An expression is called *targetable* if it may appear as the target of an assignment statement. Syntactically, an identifier is targetable and an array indexing expression (*expression*[*expression*]) is targetable. Also, a parenthesized expression is targetable if the expression inside the parentheses is. Any other target expression generates a syntax error.

Semantically, an identifier must be declared with **mut** to be targetable. Using any other identifier as a target results in a semantic error. All array elements are targetable.

In the code generator, targetable expressions will conveniently make *address code* rather than *value code*. In C++, the concept of *lvalue* corresponds with Bilby's *targetable.*

4. Type system

We now define a *type system* for Bilby: a way to write down types and some rules about them. Type systems are a broad concept: they are used to write about, reason about, and specify features of languages and programs. They often contain features that are not simply the types used within a language. For instance, in Bilby, the type system will include the *signatures* of the operators, which one cannot specify within the language Bilby itself.

Be very careful with types. The notation described below **is not used <u>in</u> Bilby programs.** It is used by people talking about Bilby programs. Only where it refers to explicit type specification or Bilby-syntax representation are we talking about the things a Bilby programmer can use in a program.

4.1. Primitive types

Primitive types are the basic built-in types in a type system, which cannot be further decomposed.

There are five primitive types in Bilby_2: the five types from Bilby_1: boolean, character, floating, integer, and string. In our type system, we will denote these types using their corresponding keywords **bool**, **char**, **float**, **int**, and **string,** or, when brevity is desired, **b**, **c**, **f**, **i**, and **s**, respectively. The bold type on these

notations in the previous sentence is to make them evident in that sentence; it is not necessary to embolden them in practice.

4.2. Compound types

A compound type is a type somehow composed from another type or types. Records, arrays, and objects are examples of compound types.

Bilby_2 introduces compound array types. Array is not itself a type, but **array [** *type* **]** is, for any valid type. (Here and henceforth we consider types and their denotation in the type system to be identical.) We may abbreviate array[*type*] as **a[***type***]** or simply **[***type***]**.

Thus, the following are all valid Bilby-2 types:

array[bool]
array[int]
array[array[f]]
a[a[a[a[c]]]]

Although Array is not a type, we use the phrase "array type" to stand in for "some type **array[**T**]**" where **T** can be any type".

We also introduce the compound range types. Again, range is not itself a type, but **range[** *type* **]** is, for *type* being **char, int,** or **float.** We may abbreviate range[*type*] as **r[***type***]** or **<***type***>.**
Thus, the following are all valid Bilby-2 types:

range[char]
<int>
r[float]
array[r[f]]

array[ ] and **range[ ]** are examples of a *type constructors*. They are operators we can apply to types in our type system to create new types. **range[ ]** is currently limited in the subtypes it can take, but that may change in future milestones. **array[ ]** can apply to any type.

4.3. Value types and reference types

Variables in the source program are associated with memory locations in a running program. If the memory location holds a value, the variable is a *value variable*, and if it holds a reference (pointer) to where the value is, the variable is called a *reference variable*. Typically the choice between keeping a variable as a value variable or reference variable is made based on its type. Thus, we will classify types as *value types* or *reference types* if their variables are implemented as value variables or reference variables, respectively.

Primitive types are often value types, and we will mainly follow this convention in Bilby. The range types will be value types. Strings are an exception, being a reference type; we have already noted this in Bilby-1. The compound *array* types will be reference types. Reference variables (variables whose type is a reference type) in Bilby will each consume 4 bytes, which is the size of a pointer on the ASM.

If *T* is a reference type, we will make a distinction between a *variable of type T*, which evaluates to a pointer, and a *record of type T*, which is a section of memory that (1) the pointer of a variable of type *T* points at, and (2) holds the values associated with the referenced structure. For instance, a variable of type array[float] holds a pointer, and that pointer should point at a record of type array[float], which is a hunk of memory holding a sequence of floating-point numbers (along with some extra array-control information; see below).

In java, primitive types are value types, and all objects are reference types. In C++, all types are value types, but there are compound types for pointers and references.

4.4. A signature of an operator

An n-ary operator is said to have a *signature* of

$(type_1, type_2, \ldots type_n) \rightarrow type$

if it accepts operands of types $type_1, type_2, \ldots type_n$ in order, and from them produces a result of type *type*. Sometimes the commas between the operand types are replaced with the Cartesian product symbol ×:

$(type_1 \times type_2 \times \ldots type_n) \rightarrow type$

in which case the parentheses are optional.

For instance, the binary operator ***** has a signature of (int, int) → int, as it accepts two integer operands and from them produces an integer result. It also has a signature of (float, float) → float.

As a further example, the unary operator **!** has a signature of (bool) → bool.

4.5. The type of operators

The *type* of an operator is the set of all of signatures that it accepts. For instance, ***** in Bilby-1 has a type of

{ (int, int)→int, (float, float)→float }

We sometimes call the type of an operator the *signatures* of the operator, for obvious reasons. (Note the plural usage here is distinct from the singular usage of section 4.4).

If an operator has only one signature, we sometimes shorten the notation by omitting the set braces. For instance, the operator **!** has a type of

(bool) → bool.

Or, equivalently and more formally,

{ (bool) → bool }.

4.6. Operators with *type* operands

Bilby_2 has two operators that take *type* (rather than expression) as one of their operands. These are the operators **[as]** (casting) and **alloc** (array creation). There are several ways we can notate and think about these operators.

4.6.1. Type operands as ordinary operands

The first way is to treat the operand as we do other operands, giving it an effective type of whatever type it represents. For instance, we could consider the cast **[68 as bool]** to have an integer first operand and a boolean second operand. (Note that for it to really have a boolean second operand, it would have to be something like **[68 as true]**). However, if we set the types of type expressions this way in the semantic analyzer, we can use signatures where (int, bool)->bool is a signature of casting. This is effective but not really clear.

4.6.2. Type operands as type literals

Another way is to treat the type operand as a *type literal* of its given type. This just means that the program text is literally a type. We'll use the notation **type T** for a type literal for the type **T**. Using this convention, we can speak of casting as having the signature

(int, type bool) → bool

With the example

[42 as bool]

matching that signature…this expression thus evalutates to a boolean value.

Note that we never have a type literal as the result of an operation; they can only be operands.

4.6.3. Type operands as constant parts of the operator

The third way is to treat the type operand as built into the operator. For instance, rather than considering **[as]** an operator, we consider **[as float]** as an operator. (And **[as int]**, **[as bool]**, **[as [int]]**, etc.) In this way, we can say that **[as float]** has a type of

{ (int) → float }

and that **[as int]** has a type of

{ (float) → int, (char) → int }

The operator **[as bool]** has a type of

{(int) -> bool, (char) -> bool}.

This approach is not really viable for large type systems, as the number of cases multiplies quickly.

4.7. Type variables

Oftentimes it is more effective to write the signatures of an operator using *type variables*. Type variables are denoted using capital letters near T. A type variable is a stand-in for "any type" unless its range is restricted. For instance, the array indexing operator **[]** has a signature of:

(array[T], int)→T

That is, it takes an array of any type as its first argument, and an integer as its second, and produces a result of the type that the array is made from. This is normal array indexing: if A is of type array[float], and you write something like "A[4]", you get back a float.

Type variables can be used inside type literals. For instance, we may consider the casting operator **[as]** to have a signature of:

(int, type T)→T

This is because it will take an integer and an explicit type T as operands, and give a result of type T. For example,

```
[68 as char]
```

Gives the result

```
D
```

However, this is not entirely true; if T is an array type, then casting does not have that signature. That is,

[68 as [bool]]

is a semantic error. So we would have to restrict the range of the variable, and say something like:

casting has a signature of (int, type T)→T for primitive types T.

Ideally, we would like to give the type of casting as

{ (S, type T) → T }

That is, casting should take an expression of any type S, and an explicitly specified type T, and convert the expression to the specified type. However, it does not do this for all pairs of types (S, T). One way to deal with this is to use the "such that" bar that is a standard part of set notation:

{ (S, type T) → T | (S, T) ∈{(int, float), (int, char), (float, int), (char, int)} }

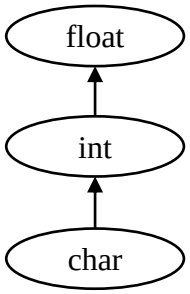
If the signatures are stated this way, then they are generally listed out (e.g. in FunctionSignatures.java). Type variables are more useful when dealing with an operator that takes any type rather than a specific subset of types.

5. Promotion: Implicit type conversions

Bilby-2 introduces implicit (unstated) type conversions, called *promotions*. Promotions are performed when the source has an operator whose operands do not match any signature of the operator.

A single operand may be promoted:

(1) from **char** to **int**,
(2) from **char** to **float**,
(3) from **int** to **float**,



This gives us a *promotion digraph* with three vertices (char, int, and float) and the two directed edges (char, int) and (int, float). We don't have a (char, float) edge because we can get from char to float with the two edges. A promotion must follow a path in this digraph. With more complex languages, and perhaps in the 3rd milestone, we get a more complicated promotion digraph. Most languages have a DAG (directed acyclic graph) for the promotion digraph. If it is not a DAG, then special rules must be used for types that can cyclically promote.

Let us call a promotion *matching* if applying it gives operand types that match a signature of the operator.

Promotion is different when applied to populated array creation, which can have many arguments, than other operators, which have one or two. See the section on populated array creation for how promotion is handled there.

We consider three different promotion levels when dealing with the other operators. For unary operators, only the first two levels are used.

Promotion	**Promotion(s)**
1	None
2	One operand
3	Two operands

We proceed to check each promotion level from 1 to 3 in turn. At any level we encounter,

1) If there are two or more matches to signatures, we issue an error.
2) If there is exactly one match, then we stop checking and use that promotion (or those promotions).
3) If there are no matches, we go on to the next level; if we're on level three, there's no match overall.

As an example of 1), consider a situation where we have actuals (operands) of types **c** and **i**, and an operator with signatures **{(i, i) → i, (f, i) → f}**. Promotion of argument 1 can yield matches to these two signatures.

For the purposes of promotion, assignment (in declarations and assignment statements) is considered an operator. Only expressions are promoted. Type literals are not promoted.

To implement promotions, I recommend having a promotion **enum** that can store which of the promotions is called for (and maybe the "null promotion" which doesn't do anything). Then have either FunctionSignature or a wrapper of FunctionSignature store one of these promotions for each argument. One must be careful with the former approach to make a copy of a FunctionSignature when there is a match to it, so that the promotions it contains aren't installed in every usage of that FunctionSignature throughout the program. This is why a wrapper object *might* be a better choice; you wouldn't have to copy FunctionSignatures, but you'd have to deal with handing out and using these wrappers. Then you'd need to write code that does matching with

promotions (as part of the matching code in FunctionSignatures) and generating the code for the promotions when you get an operatorNode in the code generator. I will elaborate on this in class.

6. Ranges

6.1. Storage and manipulation

Ranges are a new type in Bilby. A range has a subtype of either **char**, **int**, or **float**. A range with subtype **char** or **int** is known as a *discrete* range. A range consists of a *low end* and a *high end,* both of which are instances of the subtype. A range of **char** takes up 2 bytes on the ASM, a range of **int** takes up 8 bytes, and a range of **float** takes up 16 bytes. When on the ASM stack, any range takes up two locations, with the low end being underneath the high end. So when you "put a range on the accumulator stack", you push two elements: first the low end, then the high end. Calculating with ranges may require temporary storage; this storage can be allocated statically, as it should not be needed during a recursive call. Another (slower) option is to use the *memory manager* to allocate temporary storage.

6.2. Range creation

There are no range literals. Instead, ranges are created from two instances of the subtype by the **< .. >** operator. Range creation has the signatures

(char, char) → range[char]
(int, int) → range[int]
(float, float) → range[float]

For instance,

< 3 .. 8 >

Denotes an integer range with low end 3 and high end 8. The expressions in the range creation operator do not need to be constants:

< start .. 3 * hi + 1 >

is a perfectly valid range.

In a range creation, you must check that the low end is less than or equal to the high end and give a runtime error if it is not.

With ".." as a token, you must give special consideration in the lexical analyzer to scanning integer ranges. If there are no spaces in the range creation expression, it still must work. So if you see "3." in the lexical analyzer, do not assume it is a floating-point number (like "3.14") unless it is **not** followed by another "." (i.e. it could be "3..8" as part of a range expression, which should be lexically analyzed as three tokens "integer, .. , integer".)

6.3. Range-scalar addition

The operator **+** has the new signatures

(range[T], T) → range[T] and
(T, range[T]) → range[T]

for T in {**int**, **float**}.

This applies addition twice: once to the low end and once to the high end. For instance,

<4 .. 7> + 2

gives

<6 .. 9>

as would

2 + <4 .. 7>

6.4. Range casting

range[S] can only be cast to its own type (**range[S]**). Ranges participate in no other casts.

6.5. New operators

Two new prefix unary operators have been introduced for working with ranges: **low** and **high.** Both have the same precedence as the other prefix unary operators. Both have the signature of **range[T] → T.** The operator **low** gives the low end of the range, and the operator **high** gives the high end of the range.

The new binary comparison operator **in** is also for working with ranges. It has a signature of **(T, range[T]) → bool** for any **T** that is valid for a range. The expression

a **in** r

should test if a is in the range, i.e. it is equivalent to

((a **>= low** r) && (a **<= high** r))

6.6. Printing a range

To print a range, print a left-angle-bracket (**<**), then the low end, then a two-dot ellipsis (**..**) then the high end, and end with a right-angle-bracket (**>**). This mimics their construction notation. A range with a low end of 14 and a high end of 101 would print as:

<14..101>

where there are no spaces anywhere.

A range with a low end of #c and a high end of #z would print as:

<c..z>

(remember, printing a character does not print the #, it prints the character only.)

6.7. Implementing range

You can implement ranges either as three primitive types or as a true compound type. The latter involves making a subclass of Type.java for ranges. This subclass would hold a type as its subtype, and that type could be only **char**, **int**, or **float**. You will have to do something similar for arrays.

Range types require instruction *sequences* to handle loads and stores. For example, currently, stores are handled by a single instruction—in the code generator visitor in AssignmentNodes, DeclarationNodes and the routine "opcodeForStore". This latter routine is a switch on the type and like many such switches it should be replaced with something polymorphic on the type. Here, this means we should have a routine such as "generateStore" on types. For primitive types (or array types), this will just generate a fragment containing a single opcode. For range types, it will be more involved. This will be discussed in class.

7. Arrays

Arrays are a new compound type in Bilby-2. The term *length,* when referring to an array, means the number of elements in the array.

7.1. Array expressions

There are two ways to create array records in Bilby; these are the options for *arrayExpression*.

7.1.1. Populated array creation

The first way to create an array record is to list the members of an array between square brackets:

[*expressionList* **]**

All expressions in the *expressionList* must be promotable to the same type. If there is more than one type that they are all promotable to, then promote the least amount possible to get to a common type. Currently, this means that if they are all promotable to int and float, then promote them all to int.

If there are *n* expressions in the list, each promoted to type *T,* then this syntax creates an array record with length *n* with elements of type *T*. The values of the expressions are assigned to the elements of the array, in order.

For example, the expression

```
[#s, #a, #i, #d]
```

creates a 0-based array record with four characters, with the first character (index 0) being **s**, the second character (index 1) being **a**, etc.

One may not create a zero-length array with populated array creation:

```
var word = [];
```

is **not** legal Bilby. (This is because we would not know the type of the array at this definition of it.)

Array elements are always mutable.

7.1.2. Empty array creation

The second way to create an array record is to give its type and length:

alloc *arrayType* **(** *expression* **)**

Here the expression must be of type int. This form creates an array record of the given type and length. The expression gives the length of the array, and indexing is 0-based.

For example,

```
alloc [char](14)
```

creates a character array record of length 14 with lower index 0 (cf. java "new char[14]").

If the length given to an empty array creation expression is negative, then a runtime error is issued. Zero-length arrays are permitted with empty array creation:

```
alloc [char](0)
```

Creates a zero-length array of characters with lower index 0.

Arrays created with empty array creation have their data initialized to all zeroes.

7.2. Array variables

Arrays are reference types, so a variable of type array[T] (remember, this is type-system notation, not Bilby syntax notation) is a pointer to an array record, and thus the variable occupies 4 bytes. It does ***not*** occupy 16 + length*size(T) bytes.

Array variables, and any future reference-type variables, obey semantics much like java objects (which are themselves reference types):

Array variables declared with **imm** do not change their pointer; however, the elements in the record they point at may change.

```
imm R := [7, 5];
R[0] := 4;
```

is valid Bilby-2 and results in R pointing at an array record that contains the elements 4 and 5.
Assignment or initialization of arrays with other arrays is simply a pointer copy.

```
imm R := [7, 5];
imm S := R;
```

is valid Bilby-2 and results in R and S being two pointers to the same record. If this is followed by

```
R[0] := 4;
```

then not only will R[0] be 4, but S[0] will, as well.

Array variables declared with **mut** may change their pointer as well as the elements in the record.

```
mut R := [7, 5];
R[0] := 2;
…
R := alloc [int](5);
```

is valid Bilby-2.

However, array variables may not be assigned an array of a different type.

```
imm intArray := [7, 5];
imm charArray := [#a, #z];

mut R := intArray;
R := charArray;
```

is ***not*** valid Bilby-2. The **mut** declaration sets R's type as array[int], and the assignment tries to update it with an array[char], so this should generate a typechecking error.

7.3. Array indexing

The *array indexing expression* is of the form:

expression$_1$ **[** *expression*$_2$ **]**

Here, *expression*$_1$ must have type array[T] for some T, and *expression*$_2$ must be of integer type. The result of this expression is the *n*-th element of the array *expression*$_1$, where *expression*$_2$ evaluates to *n*. The type of the result is T. Thus **[ ]** (array indexing) has the signature **(array(T), int) → T** for any type **T**.

Whenever an array is indexed, the index must be checked for validity (i.e. it must be between 0 and the highest index in the array, inclusive). If the index is not valid, a runtime error is issued.

Array indexing expressions are targetable. In the code generator, generate *address code* for them.

7.4. Array length

The *length expression* is of the form:

length *expression*

The *expression* must have array type, and the result is the integer length (number of elements) of the array. Length thus has the signature **array(T) → int** for any type T.

7.5. Array printing

If an expression of array type is in the expressionList of a printing statement, then the array is printed as the following sequence:

[
A print of the first element
,

a space
a print of the next element
,
a space
…

A print of the last element
]

This is a quite natural way to print an array. For example, the array [1.23, 2.79, 5.41] is printed as

```
[1.23, 2.79, 5.41]
```

Note that the only spaces printed are single spaces after after each comma.
The "print of the *n*th element" parts are done recursively. It is your problem to figure out when that recursion is done—at compile time or at run time.

7.6. Multidimensional arrays

Bilby does not have true multidimensional arrays. Instead (like java), one must use arrays of arrays. An array with array elements must have all of those array elements of the same type, but they need not be of the same size. For instance, the following is legal:

```
imm numSets := [ [1, 2, 3], [4, 5], [6, 7, 8], alloc [int](0)];
```

This makes numSets have type array[array[int]], with four elements. For empty array creation, the following is the proper idiom for a rectangular array:

```
imm width := 4;
imm height := 7;
imm matrix := alloc [[float]](width);

mut x := 0;
while(x < width) {
      matrix[x] := alloc [float](height);
      x := x + 1;
}
```

7.7. Array casting

Array types may not be cast to other types, including other array types. No other type may be cast to an array type, although array types may be cast to themselves.

8. Records

In a running bilby program, we will often have many pieces of heap memory that we need to process and keep track of. In order to do this, we will enforce a simple record format, as follows:

Type identifier (4 bytes)	Status (4 bytes)	Rest of record (?? bytes)

The type identifier is an integer describing what type is being stored in this record. Currently there are only two "types" with records: **string** and **array( )**. (Remember that **array( )** is not by itself a type.) String is given type identifier 3, and array is given type identifier 5.

The status field currently holds only four bits of data, in the lowest bits.

- The datum in bit 0 is the *immutability status* of the elements of the record (1 is immutable, 0 is mutable).
- The datum in bit 1 is the *subtype-is-reference status* of the array; this indicates whether the subtype T of the array is a reference type (i.e. if the subtype is itself an array or string type)
- The datum in bit 2 is the *is-deleted status* of the record, which indicates that this record has been given to the memory deallocator.
- The datum in bit 3 is the *permanent status* of the record. Records with this bit set won't be deallocated. If an attempt is made to deallocate a permanent record, it is silently ignored. (It doesn't issue a runtime error).

8.1. Array records

An *arrayExpression* results in the allocation of a new block of memory—a record—from the heap at runtime. The record for the type array[T] (informally, an *array record*) has the following format:

Type identifier (4 bytes)	Status (4 bytes)	Subtype size (4 bytes)	Length (4 bytes)	Elements ((subtypeSize * length) bytes)

The *type identifier* for an array is the integer 5. When creating an array, set

- The *immutability status* to 0. (Array elements are mutable.)
- The *subtype-is-reference status* according to the subtype of the array.
- The *is-deleted status* to 0.
- The *permanent status* to 0. Array records can be deleted.

The subtype size is the number of bytes consumed by a variable of type T; we will refer to this as size(T). The *length* is the number of elements in the array. (Note that the highest index in the array is *length*-1).

Over the lifetime of this record, only the elements and possibly the status flags may change. The other values will not.

To create an Array record, you will need to call the memory manager. This means you must enable the memory manager. This involves uncommenting one line (in makeASM( ) in the code generator), and adding another line somewhere. Look at the public methods in MemoryManager.java to figure out what the added line should be (and then you must decide exactly where it should go). You do not need to understand the MemoryManager in detail; you only need to figure out its API.

9. Operator precedence

The precedence of operators is

Highest precedence	parentheses, populated array creation empty array creation casting range creation	**( ) [ ] alloc[]() [as] <..>**

	array indexing	**[ ]**
(prefix unary operators are right-associative)	prefix unary operators	**+ - ! length low high**
	multiplicative operators	***** /
	additive operators	**+ –**
	comparisons	**< > <= >= == != in**
	and	**&&**
Lowest precedence	or	**||**

These are all left-associative operators, except as noted.