The ALLCOT language

Marc Moreno Maza
University of Western Ontario

CS447

December 1, 2004

Introduction

The ALLCOT language is a simple programming language which is meant to support student assignments for the computer science course CS 447 at the `University of Western Ontario`. In particular, the students of this course should develop a front-end compiler for ALLCOT. The intermediate representation (i.e. the target language) will be a subset of the C language.

The name ALLCOT stands for *A Language for Learning COmpiler Theory.* and refers to the ALDOR programming language (*A Language for Describing Objects and Relationships*) developed at `ORCCA`. The ALLCOT language is largely inspired from ALDOR, but as a teaching language, the features of ALLCOT are very limited comparing to those of ALDOR.

This document is evolving and the remaining sections show the development of the specifications of the ALLCOT language. Sections 1, 2 and 3 were part of the original draft. They give an overview of the language. However, Section 3 evolved as Students implemented their first YACC parsers for ALLCOT. The changes related to the language (resolving inconsistencies, language extensions ...) are reported in Section 5 and Section 6 whereas those related to the grammar for statements are reported in Section 4.

In order to realize an ALLCOT compiler, one needs to clarify the specifications of the built-in types and the inheritance mechanism. This is done respectively in Section 7 and Section 8.

1 An Allcot program

An Allcot program consists of a sequence of definitions, the last one being the definition of a function called `main`, followed by a call to this function. Executing this program means executing this call. An example of such program is given below.

```
Monoid: Category == with {
  print: % -> ();
  times: (%, %) -> %;
  one: () -> %;
  equal?: (%, %) -> Boolean;
}
MonoidOfZ: Category ==  Monoid with {
  convert: Integer -> %;
}
OddNumbers: MonoidOfZ == add {
  attrib(n: Integer);
  convert(x: Integer): % ==
    if (x rem 2 = 1) then attrib(x) else error;
  print(x: %): () == print (x.n);
  times(x: %, y: %): % == attrib((x.n) * (y.n));
  one(): % == convert(1);
  equal?(x: %, y: %): Boolean == (x.n) = (y.n);
}
WordMonoid: Category == Monoid with {
  word: String -> %;
}
Word: WordMonoid == add {
  attrib(st: String, le: Integer);
  word(x: String): % == attrib(x, length(x));
  print(x: %): () == print (x.st);
  times(x: %, y: %): % == {
    z: String := concat(x.st,y.st);
    l: Integer := x.le + y.le;
    attrib(z,l);
  }
  one(): % == word(empty(),0);
```

```
  equal?(x: %, y: %): Boolean == {
    if (x.le ~= y.le) then return false;
    equal?(x.st,y.st);
  }
}
main(): () == {
  x: Word := word("Hello ");
  y: Word := word("Mars.");
  xy: Word := times(x,y);
  print(xy);
}
main();
```

Up to the *magic function* `attrib`, the program above could be a fragment of an ALDOR program. It declares three *categories* (or interfaces) and two domains (or classes). If you are an ALDOR user, you could be surprised by this `attrib` function: the goal here is to keep the grammar, the type checking and the translation simple!

Observe also that the above program relies on three *built-in* domains, namely `Boolean`, `String` and `Integer`.

2 The ALLCOT language: a first overview

Strictly speaking, the ALDOR language is not an object oriented language. Indeed, its notion of a *domain* is richer than a *class* (like a JAVA class). However, the ALLCOT language, because of its restrictions w.r.t. ALDOR, will essentially be a simple object oriented language.

In the above example `OddNumbers` and `Word` are classes in the sense that

- the values of these types are encoded by means of *attributes* that are not visible outside of the class,
- each function of these types is morally either a *method* (that is, a function whose first argument is of the type where the function is defined) or a *creator* (that is, a function which returns a value of the type where the function is defined).

In the above example, `Monoid`, `WordMonoid` and `MonoidOfZ` are interfaces in the sense that each of them is a list of function declarations. Note that both `WordMonoid` and `MonoidOfZ` inherit from `Monoid`. Observe also that

- on the contrary of JAVA, creators like `one`, `convert` or `word` are allowed in a category,
- the syntax of an ALLCOT function declaration is rather that of PASCAL than that of C or JAVA.

Observe also the role of the symbol %.

- In one of the categories `Monoid`, `WordMonoid` or `MonoidOfZ`, this symbol means an element of a domain implementing this category.
- In one of the domains `OddNumbers` and `Word`, this symbol means an element of this domain.

Finally, observe the different kinds of equality-like symbols.

== is used for a category definition, a domain definition or a function definition. In fact, in `aldor` it is used for a constant definition, that is for the definition of some variable whose value cannot change.

:= is used for a variable definition (whose value can change).

= is used for the equality test.

3 The ALLCOT syntax

In Section 3.1 and Section 3.2 we present domains and categories before giving their grammar in Section 3.5. A grammar for the ALLCOT statements and (arithmetic and boolean) expressions appear in Sections 3.3 and 3.4 respectively.

3.1 Defining a category

A category definition looks like

$$Identifier : \textbf{Category} == Identifier \ \textbf{with} \ \{ \ FunctionDeclarationSequence \ \}$$

or like

$$Identifier : \textbf{Category} == \textbf{with} \ \{ \ FunctionDeclarationSequence \ \}$$

where in each case

- the first $Identifier$ must be a new identifier,
- the second $Identifier$ must refer to an existing category,
- $FunctionDeclarationSequence$ is a sequence (possibly empty) of function declarations, each declaration ending with a **;**.

The specifications of the so-defined category consists of the function declarations on the right of the **with** key-word, plus the specifications of the existing category, if present.

A function declaration looks like

$$Identifier : \ (\ TypeSequence \ ) - > Type \tag{1}$$

where

- $Identifier$ must be a new identifier,
- $TypeSequence$ is a sequence (possibly empty) of % or type identifiers, separated by commas,
- $Type$ is % or a type identifier, or `()` (which means `void`).

such that

- either % is the first element in $TypeSequence$ (and the function is viewed as a method)
- or % is not the first element in $TypeSequence$ but $Type$ is % (and the function is viewed as a creator).

The above constraint is not required in ALDOR and is meant here to ease the type checking in ALLCOT. We shall return to this later.

Some remarks regarding the non-terminal $Type$. There are circumstances where $Type$ cannot be % or `()` . For instance, we shall never need to declare of variable of type `()` . Moreover, in the definition of the `main` function, we shall never need to declare of variable of type %. These special cases can be treated at the level of the type checking.

Observe that the empty list `()` can be generated by two different ways: as an empty list of parameter of a function or as the `void` type.

3.2 Defining a domain

A domain definition looks like

$$Identifier : Identifier == Identifier \ \textbf{add} \ \{ \ DomainBody \ \}$$

or like

$$Identifier : Identifier == \textbf{add} \ \{ \ DomainBody \ \}$$

where in each case

- the first $Identifier$ must be a new identifier,
- the second $Identifier$ must refer to an existing category,
- the third $Identifier$, if present, must refer to an existing domain.

The inheritance mechanism will be discussed later.

The *DomainBody* starts with an `attrib`-*statement*, that is a statement of the form

$$\texttt{attrib} \ (\ FormalParameterSequence \ ) \ ;$$

and continues with a sequence of function definitions. A function definition looks like

$$Identifier \ (\ FormalParameterSequence \ ) : \ Type == Statement;$$

or like

$$Identifier \ (\ FormalParameterSequence \ ) : \ Type == \{ \ StatementSequence \ \}$$

These function definitions must *satisfy* the category of the domain.

A sequence of formal parameters is a sequence of declarations of the form

$$Identifier : \ Type$$

separated by commas.

The goal of the `attrib`-statement is to declare the attributes of a value (or object) of the domain (or class) being defined. In the case of

- `OddNumbers`, each value has just one attribute which is an `Integer`,
- `Word`, each value has two attributes: a `String` called `st` and an `Integer` called `le` representing the length of this `String`.

Note that `attrib` is also used as a private creator. Public creators have to use `attrib` directly or indirectly in order to initialize a new value.

3.3 Statements

We give now a grammar for generating statements of the Allcot language.

$$
\begin{array}{lll}
AttribStatement & \longmapsto & \mathbf{attrib}\ (\ FormalParameterSequence\ ) \\
Statement & \longmapsto & Assignment \\
Statement & \longmapsto & Alternative \\
Statement & \longmapsto & WhileLoop \\
Statement & \longmapsto & RightValue \\
Statement & \longmapsto & \mathbf{error} \\
Statement & \longmapsto & \mathbf{return}\ RightValue \\
Statement & \longmapsto & ElementCreation \\
Assignment & \longmapsto & LeftValue\ \mathbf{:=}\ RightValue \\
LeftValue & \longmapsto & \mathbf{id} \\
LeftValue & \longmapsto & \mathbf{id}\ \mathbf{:}\ \ Type \\
LeftValue & \longmapsto & AttributeCall \\
RightValue & \longmapsto & FunctionCall \\
RightValue & \longmapsto & ArithmeticExpression \\
RightValue & \longmapsto & BooleanExpression \\
AttributeCall & \longmapsto & \mathbf{id}\cdot\mathbf{id} \\
FunctionCall & \longmapsto & \mathbf{id}\ (\ ActualParameterSequence\ ) \\
ElementCreation & \longmapsto & \mathbf{attrib}\ (\ ActualParameterSequence\ ) \\
ActualParameterSequence & \longmapsto & ActualParameter \\
ActualParameterSequence & \longmapsto & \varepsilon \\
ActualParameterSequence & \longmapsto & ActualParameter\ \mathbf{,}\ MoreParamters \\
MoreParamters & \longmapsto & ActualParameter \\
MoreParamters & \longmapsto & ActualParameter\ \mathbf{,}\ MoreParamters \\
ActualParameter & \longmapsto & ArithmeticExpression \\
Alternative & \longmapsto & \mathbf{if}\ Test\ \mathbf{then}\ IfTrue\ \mathbf{else}\ IfFalse \\
Alternative & \longmapsto & \mathbf{if}\ Test\ \mathbf{then}\ IfTrue \\
Test & \longmapsto & BooleanExpression \\
IfTrue & \longmapsto & \{\ \ StatementSequence\ \} \\
IfFalse & \longmapsto & \{\ \ StatementSequence\ \} \\
WhileLoop & \longmapsto & \mathbf{while}\ Test\ \mathbf{repeat}\ IfTrue
\end{array}
$$

3.4 Arithmetic and boolean expressions

Here is a grammar for the arithmetic expressions and boolean expressions of Allcot.

$$
\begin{array}{lcl}
ArithmeticExpression & \longmapsto & E \\
E & \longmapsto & E + T \\
E & \longmapsto & E \text{ - } T \\
E & \longmapsto & T \ \textbf{rem} \ T \\
E & \longmapsto & T \\
T & \longmapsto & T \ \texttt{*} \ F \\
T & \longmapsto & F \\
F & \longmapsto & (\ E\ ) \\
F & \longmapsto & \textbf{num} \\
F & \longmapsto & \textbf{id} \\
F & \longmapsto & AttributeCall \\
BooleanExpression & \longmapsto & B \\
B & \longmapsto & E = E \\
B & \longmapsto & E \sim = E \\
B & \longmapsto & E < E \\
B & \longmapsto & E <= E \\
B & \longmapsto & B \ \textbf{and} \ B \\
B & \longmapsto & B \ \textbf{or} \ B \\
B & \longmapsto & (\ B\ ) \\
B & \longmapsto & \textbf{not} \ B \\
B & \longmapsto & AttributeCall
\end{array}
$$

The meaning of operators $+$, -, **rem**, *, $=$, $<$, $<=$, **and**, **or** and **not** are as usual. The operator $\sim =$ stands for the *different* predicate. The terminal **num** stands for integer values.

3.5 Programs

We give now a grammar for Allcot programs.

$$
\begin{array}{lcl}
Program & \longmapsto & DefinitionSequence\ MainSequence \\
DefinitionSequence & \longmapsto & CategoryDefinition\ DefinitionSequence \\
DefinitionSequence & \longmapsto & DomainDefinition\ DefinitionSequence \\
DefinitionSequence & \longmapsto & \varepsilon
\end{array}
$$

CategoryDefinition	$\longmapsto$	**id: Category == ** *ParentCategory WithBody*
ParentCategory	$\longmapsto$	**id**
ParentCategory	$\longmapsto$	ε
WithBody	$\longmapsto$	**with {** *FunctionDeclarationSequence* **}**
FunctionDeclarationSequence	$\longmapsto$	*FunctionDeclaration* **;** *FunctionDeclarationSequence*
FunctionDeclarationSequence	$\longmapsto$	ε
FunctionDeclaration	$\longmapsto$	**id : (** *TypeSequence* **)** $\longrightarrow$ *TypeOrVoid*
TypeSequence	$\longmapsto$	ε
TypeSequence	$\longmapsto$	*Type*
TypeSequence	$\longmapsto$	*Type* **,** *MoreTypes*
MoreTypes	$\longmapsto$	*Type*
MoreTypes	$\longmapsto$	*Type* **,** *MoreTypes*
Type	$\longmapsto$	**%**
Type	$\longmapsto$	**id**
TypeOrVoid	$\longmapsto$	*Type*
TypeOrVoid	$\longmapsto$	**( )**
DomainDefinition	$\longmapsto$	**id: id ==** *ParentDomain AddBody*
ParentDomain	$\longmapsto$	**id**
ParentDomain	$\longmapsto$	ε
AddBody	$\longmapsto$	**add {** *AttribStatement* **;** *FunctionDefinitionSequence* **}**
FunctionDefinitionSequence	$\longmapsto$	*FunctionDefinition FunctionDefinitionSequence*
FunctionDefinitionSequence	$\longmapsto$	ε
FunctionDefinition	$\longmapsto$	**id (** *FormalParameterSequence* **) :** *TypeOrVoid* **==** *FunctionBody*
FormalParameterSequence	$\longmapsto$	ε
FormalParameterSequence	$\longmapsto$	*FormalParameter*
FormalParameterSequence	$\longmapsto$	*FormalParameter* **,** *MoreFormalParameters*
MoreFormalParameters	$\longmapsto$	*FormalParameter*
MoreFormalParameters	$\longmapsto$	*FormalParameter* **,** *MoreFormalParameters*
FormalParameter	$\longmapsto$	**id:** *Type*
FunctionBody	$\longmapsto$	*Statement* **;**
FunctionBody	$\longmapsto$	**{** *StatementSequence* **}**
StatementSequence	$\longmapsto$	*Statement* **;**
StatementSequence	$\longmapsto$	*Statement* **;** *StatementSequence*

$$
\begin{array}{lll}
MainSequence & \longmapsto & MainDefinition\ MainCall \\
MainDefinition & \longmapsto & \textbf{main ():\ () == \{ } StatementSequence \textbf{ \}} \\
MainCall & \longmapsto & \textbf{main () ;}
\end{array}
$$

4 Avoiding conflicts in the parser

A YACC parser can work even if it has shift/reduce or reduce/reduce conflicts. Please refer to the YACC documentation available from our web site (Section 5 page 11). Some of these conflicts may lead to an unexpected behavior of your parser whereas others are safe.

The original grammar has been revised in order to limit the number of shift/reduce and reduce/reduce conflicts with YACC. However, some of them remain but are considered are safe. Here's a description of these changes w.r.t. the original grammar.

4.1 Rules of the form $A \longmapsto \textbf{id}$

A simple statement like

```
x := a
```

could be generated before by several different syntax trees. Indeed the sentential form `a` could be produced by

$$RightValue \quad \longmapsto \quad \textbf{id}$$

or by

$$
\begin{array}{lll}
RightValue & \longmapsto & ArithmeticExpression \\
ArithmeticExpression & \longmapsto & E \\
E & \longmapsto & T \\
T & \longmapsto & F \\
F & \longmapsto & \textbf{id}
\end{array}
$$

This ambiguity has been suppressed by forbidding the first case. See how $RightValue$ is defined now. Similarly, the following rule has been removed

$$ActualParameter \quad \longmapsto \quad \textbf{id}$$

since we have also

$$ActualParameter \quad \longmapsto \quad ArithmeticExpression$$

which covers the case where an actual parameter is a variable.

4.2 The attrib statement

The word **attrib** is a keyword. Indeed, we want to make sure that the programmer starts the implementation of a domain by defining the representation of the elements of this domain. This is why we have a rule:

$$AttribStatement \quad \longmapsto \quad \textbf{attrib} \; (\; FormalParameterSequence \;)$$

We also want **attrib** to be a reserved word, with its own token. Therefore, it is convenient to have a rule for an **attrib** call, that is when we create an element of the actual class.

$$ElementCreation \quad \longmapsto \quad \textbf{attrib} \; (\; ActualParameterSequence \;)$$

In addition, we need to say that an *ElementCreation* is a statement:

$$Statement \quad \longmapsto \quad ElementCreation$$

4.3 $MethodCall$ was superfluous

From a syntax point of view the former $MethodCall$ was a $FunctionCall$. Although ALLCOT (and ALDOR) have object-oriented features, they are also largely inspired by functional languages. So, the non-terminal $MethodCall$ was in fact superfluous.

4.4 Allowing $AttributeCall$ to be an $ArithmeticExpression$

Attribute calls are frequently involved in arithmetic or boolean expressions. It is therefore useful to add the rules

$$\begin{array}{rcl} F & \longmapsto & AttributeCall \\ B & \longmapsto & AttributeCall \end{array}$$

4.5 The return statement

It makes sense to have **return** as a reserved word and then to add the following rule

$$Statement \quad \longmapsto \quad \textbf{return} \; RightValue$$

4.6 Allowing *ArithmeticExpression* to be a *Statement*

For function returning an arithmetic or a boolean expression like

```
equal?(x: %, y: %): Boolean == (x.n) = (y.n);
```

it may be useful to add the rule

$$Statement \longmapsto RightValue$$

where $RightValue$ is now defined by

$$\begin{array}{lcl} RightValue & \longmapsto & FunctionCall \\ RightValue & \longmapsto & ArithmeticExpression \\ RightValue & \longmapsto & BooleanExpression \end{array}$$

You may also refuse the above function definition and prefer

```
equal?(x: %, y: %): Boolean == return ((x.n) = (y.n));
```

as in JAVA.

4.7 Actual parameters

Observe how $ActualParameterSequence$ is defined by

$$\begin{array}{lcl} ActualParameterSequence & \longmapsto & ActualParameter \\ ActualParameterSequence & \longmapsto & \varepsilon \\ ActualParameterSequence & \longmapsto & ActualParameter \text{ , } MoreParamters \\ MoreParamters & \longmapsto & ActualParameter \\ MoreParamters & \longmapsto & ActualParameter \text{ , } MoreParamters \\ ActualParameter & \longmapsto & ArithmeticExpression \end{array}$$

In particular, an actual parameter cannot be a function call. This is meant to simplify type checking. Of course, this restriction can be worked out by the programmer by using temporary variables.

5 Language restrictions

Syntactic sugars are easy to implement with a pre-processor. For a simple course project compiler, we may skip this and prefer to avoid implementing syntactic sugars in the YACC program. See Sections 5.1, 5.2 and 5.3. Another important language restriction, which is described at the end of the previous section, is the fact that an actual parameter cannot be a function call. This is made on purpose in order to limit the complexity of the type checking.

5.1 Function declarations

In order to make things simpler (and on the contrary of the example of Section 1), we can restrict function declarations to

$$FunctionDeclaration \quad \longmapsto \quad \mathbf{(}\ TypeSequence\ \mathbf{)} \longrightarrow TypeOrVoid$$

and forbid

$$FunctionDeclaration \quad \longmapsto \quad Type \longrightarrow TypeOrVoid$$

5.2 Function bodies

Similarly, one may **not** consider the following syntactic sugar

$$FunctionBody \quad \longmapsto \quad Statement\ \mathbf{;}$$

and restrict function bodies to be of the form

$$FunctionBody \quad \longmapsto \quad \mathbf{\{}\ StatementSequence\ \mathbf{\}}$$

5.3 Forbidding *ArithmeticExpression* to be a *Statement*

As mentioned in Section 4, , one may **not** consider the following syntactic sugar

$$Statement \quad \longmapsto \quad RightValue$$

6 Language extensions

6.1 Arrays

The ALDOR solution is very elegant. It is based on the following features

- In ALDOR, one can write a function which returns a domain (or a category). Moreover, such functions can take categories, domains or elements among their arguments. So, if `T` is a domain, one can create a function `Array` such that `Array(T)` is a domain whose elements are arrays of elements from `T`.
- In ALDOR, one can define a domain, a category or a function in any scope.

Of course, we do not plan to implement such features in ALLCOT. But we would like to create arrays of any type in any scope of the source program. However, in order to make such arrays easy to translate into our intermediate language, we can restrict to fixed size arrays. For declaring a variable of an array type, let us modify the *Type*-rules as follows.

$$\begin{array}{lcl} Type & \longmapsto & \mathbf{\%} \\ Type & \longmapsto & \mathbf{id} \\ Type & \longmapsto & \mathbf{Array}\ \mathbf{(}\ \mathbf{id}\ \mathbf{,}\ \ ArithmeticExpression\ \mathbf{)} \end{array}$$

where, in the third rule, **id** is meant to be a type and $ArithmeticExpression$ is meant to be a positive integer. For initializing an array, we need to add a special form of $RightValue$, such that the $RightValue$-rules become

$$\begin{array}{lcl} RightValue & \longmapsto & FunctionCall \\ RightValue & \longmapsto & ArithmeticExpression \\ RightValue & \longmapsto & BooleanExpression \\ RightValue & \longmapsto & [\ RightValueSequence\ ] \\ RightValueSequence & \longmapsto & RightValue \\ RightValueSequence & \longmapsto & RightValueSequence\ \mathbf{,}\ RightValue \end{array}$$

Then, we need to read and overwrite an element in an array, which means accessing an element in an array. This can be done exactly like in an $AttributeCall$. Therefore, the following program fragment becomes syntactically valid

```
x: Array(Integer, 3) := [2,3,6];
i: Integer := 2;
x.i := 5;
```

Observe that this solution for arrays is coherent for the philosophy of ALLCOT and ALDOR. In fact, the ALLCOT `Array` can be viewed as a built-in domain constructor.

7 Built-in domains

In addition to arrays, we need at least two built-in domains: `Integer` and `String`. `Boolean` can be skipped at a first stage. Indeed, we have booleans expressions and we can encode boolean values by integers.

`Integer`. The built-in domain `Integer` needs to support the arithmetic operations appearing in *ArithmeticExpression* plus the `print` function with signature

```
print: (Integer) -> ()
```

`String`. The built-in domain `String` needs to support *creation* (like in `s: String := "Allcot"`) *element access* (like in *AttributeCall*) plus a `concat` and a `print` function with signatures

```
concat: (String, String) -> String
print: (String) -> ()
```

Note also that the grammar needs to be updated in order to have literal strings

$$\begin{array}{lcl} RightValue & \longmapsto & LiteralString \\ LiteralString & \longmapsto & \textbf{" string "} \end{array}$$

8 The ALLCOT semantics

8.1 Categories

A category is a set of signatures. If a category `C2` inherits from a category `C1`, then the set of signatures of `C2` is the union of that of `C1` plus the set of signatures from the **with**-part of `C2`. So, duplicates are removed.

8.2 Domains

Basic principles.

- Each domain `D1` has a category `C1` and must implement every function from `C1`.
- Every element from `D1` is encoded in the same way and this encoding is given by the *AttribStatement*.
- We call *attribute* each of the formal parameters in an *AttribStatement*.
- The *AttribStatement* is mandatory in every domain definition but its list of attributes may be empty.
- If a domain `C2` inherits from a domain `C1` then the attributes of `C2` are those of `C1` followed by those declared in the *AttribStatement* of `C2`.
- In a domain `C2` inheriting from a domain `C1`, if a function `f` is defined in both `C1` and `C2`, then the version of `C2` is used.
- In a domain `C2` inheriting from a domain `C1`, if a function `f` is declared for both `C1` and `C2` but is only implemented in `C1`, then two cases arise:
  - if `C1` and `C2` have the same attributes, then the function `f` defined in `C1` can be reused to implement the `C2`-function `f` just by making a call (with a possible cast). See the example below.
  - if `C2` has some additional attributes, then in order to be in the previous case, the target program needs to define a map from `C2` to `C1`.

When translated to C, a domain `D1` will become a `struct` definition followed by function definitions. For instance, consider the following ALLCOT program.

```
CoupleType: Category == with {
  construct: (Integer, Integer) -> %;
  equal: (%, %) -> Integer;
  print: (%) -> ();
}
```

```
Couple: CoupleType == add {
  attrib(left: Integer, right: Integer);
  construct(l: Integer, r: Integer): % == attrib(l, r);
  equal(x: %, y: %): Integer == {
    if (x.left = y.left) and (x.right = y.right)
       then {return 2; } else {return 0; }
  }
  print(x: %): () == {
     print(x.left);
     print(x.right);
  }
}

Pair: CoupleType == Couple add {
  attrib();
  construct(l: Integer, r: Integer): % ==
     if (l = r) then error else attrib(l, r);
  equal(x: %, y: %): Integer == {
     if ((x.left = y.left) and (x.right = y.right) or
        (x.left = y.right) and (x.right = y.left))
     then {return (2); }
     else {return (0);}
  }
}

main():() == {
  p: Couple := construct(1,2);
  print(p);
  q: Pair := construct(2,1);
  print(p);
}
main();
```

A possible translation to C would be

```
struct Couple
{
  int left;
  int right;
```

```
};

struct Couple *construct_Couple(int l, int r)
{
  struct Couple *tmp_Couple = (struct Couple *) malloc(sizeof(struct Couple));
  (tmp_Couple) -> left = l;
  (tmp_Couple) -> right = r;
  return (tmp_Couple);
};

int equal_Couple(struct Couple *tmp1_Couple, struct Couple *tmp2_Couple)
{
  if (((tmp1_Couple) -> left == (tmp1_Couple) -> left) &&
      ((tmp1_Couple) -> right == (tmp1_Couple) -> right)) {
    return (2);
  } else {
    return (1);
  }
}

int print_Couple(struct Couple *tmp_Couple)
{
  int tmp_int_0 = 0;
  printf("%d \n", (tmp_Couple) -> left );
  printf("%d \n", (tmp_Couple) -> right);
  return (tmp_int_0);
};

struct Pair
{
  int left;
  int right;
};

struct Pair *construct_Pair(int l, int r)
{
  struct Pair *tmp_Pair = (struct Pair *) malloc(sizeof(struct Pair));
  if (l == r) {
    exit(1);
  } else {
```

```
    (tmp_Pair) -> left = l;
    (tmp_Pair) -> right = r;
    return (tmp_Pair);
  }
};

int equal_Pair(struct Pair *tmp1_Pair, struct Pair *tmp2_Pair)
{
  if ((((tmp1_Pair) -> left == (tmp2_Pair) -> left) &&
      ((tmp1_Pair) -> right == (tmp2_Pair) -> right)) ||
    (((tmp1_Pair) -> left == (tmp2_Pair) -> right) &&
     ((tmp1_Pair) -> right == (tmp2_Pair) -> left))) {
    return (2);
  } else {
    return (0);
  }
}

int print_Pair(struct Pair *tmp_Pair)
{
  return (print_Couple((struct Couple *) tmp_Pair));
};

int main()
{
  int tmp_int_1 = 1;
  int tmp_int_2 = 2;
  int tmp_int_3 = 2;
  int tmp_int_4 = 1;
  struct Couple *p = construct_Couple(tmp_int_1, tmp_int_2);
  struct Pair *q = construct_Pair(tmp_int_3, tmp_int_4);
  print_Couple(p);
  print_Pair(q);

  return 0;
};
```

8.3 Every function is a method or a creator

Remember that, in order to make life simple, ALLCOT is an OO language with functional notations. More precisely, every ALLCOT function is either a method or a creator. Hence, an ALLCOT cannot have static methods. So, consider a function call

```
f(a,b,c)
```

We can assume that the types of `a`, `b` or `c` are known. Let `T` be the type of `a`.

(1) If `T` has a function whose signature matches that of `f`, then we use this function for the call `f(a,b,c)`.

(2) Otherwise, we assume that `f` is a creator. So, we need the return type of `f(a,b,c)`. Observe that the only rule generating a $FunctionCall$ is

$$RightValue \quad \longmapsto \quad FunctionCall$$

and that $RightValue$ is generated by the rules

$$\begin{array}{lll} Statement & \longmapsto & \textbf{return}\ RightValue \\ Assignment & \longmapsto & LeftValue\ \textbf{:=}\ RightValue \\ RightValue & \longmapsto & [\ RightValueSequence\ ] \\ RightValueSequence & \longmapsto & RightValue \\ RightValueSequence & \longmapsto & RightValueSequence\ \textbf{,}\ RightValue \end{array}$$

Therefore, the (expected) type of a $RightValue$ can always be deduced from its context without evaluating this $RightValue$:

- The type of $RightValue$ in **return** $RightValue$ is the return type of the function where this return-statement appears.
- The type of $RightValue$ in $LeftValue$ **:=** $RightValue$ must be that of $LeftValue$, which is known after visiting the node corresponding to $LeftValue$.
- The type of $RightValue$ in $RightValueSequence$ **,** $RightValue$ is that of the array being defined.

So, let `S` be the return-type of `f(a,b,c)`. If the domain `S` has a function whose signature matches that of `f`, then we use this function for the call `f(a,b,c)`. Otherwise, we report *error*.

8.4 Integer

Consider `Integer` to be equivalent to the C type `int`. The built-in type `Integer` must support

addition: for instance `x + y`,

multiplication: for instance `x * y`,

subtraction: for instance `x - y`,

division-remainder: for instance `x rem y`,

unary-minus: for instance `-x`,

equality-test: for instance `x = y`,

unequality-test: for instance `x -= y`,

strict-inequality-test: for instance `x < y`,

large-inequality-test: for instance `x <= y`,

printing: for instance `print(x)`.

8.5 String

Consider `String` to be equivalent to the following C strings

```
#define MAX_STRING_LENGTH 255
typedef char allcotString[MAX_STRING_LENGTH+1];
```

The built-in type `String` should support:

concatenation: for instance `x + y` or `concat(x,y)`,

length: for instance `length(x)`,

printing: for instance `print(x)`,

equality-test: for instance `x = y`,

unequality-test: for instance `x -= y`,

substring: for instance `substring(x,i,j)`.

However, for the purpose of the assignment 3, the last three operations are not required.