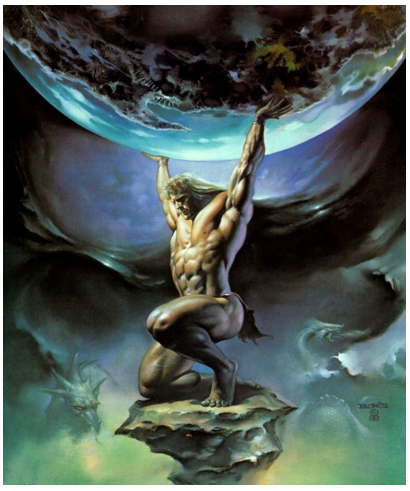
Bilkent University

Department of Computer Engineering

CS315 - Programming Languages Project

Atlas



Project Group(58) Members:

Eren Bilaloğlu 21401603 - Section 3

Gökcan Değirmenci - 21401658 - Section 3

Zeki Okur - 21400782 - Section 3

Supervisor & Course Instructor: Ertuğrul Kartal Tabak

Teaching Assistant of the course: Gizem Çaylak

Introduction:

Why we have wanted to create *Atlas* in the first place?

Atlas is a programming language that is designed for propositional calculus with assuring the maximum readability and type-safety. The most important feature of Atlas language presents is enabling use of compound logical formulas as if they are single operations. With this feature, Atlas allows users to perform **xor, nor, nand, if...then, if and only if** operations like the way the more basic operations (and, or) are performed in programming languages. Another feature that comes with Atlas is "**relational logic mapping**". This feature allows forming relations with two concepts so that if a relation exists, it returns true. Relational mapping is useful for creating logic relations with different concepts. With using Atlas, world becomes your playground: you may define interesting concepts of alternative truths.

Below sections mostly consist of description of language constructs with their conventions, why we use that constructs, our design decisions and detailed BNF description of the language.

Goals of this language are to:

- Provide a simpler programming language to its audience by giving them a handful of basic rules and keywords that are close to English language to increase readability, writability and adaptability.
- Be capable of constructing abstract code blocks via functions and predicates that are specialized for logic operations so that user can write shorter and better constructed programs to normalise the learning curve.
- Type-safety: better to catch the bugs in the early stage. No more "*Undefined is not an object*" errors.

1) Basic Structure of the Language

a) Variables

There are four types of variables in Atlas. Two of them are numeric types: namely **integer** and **float.** On the other hand, there are two non-numeric types, one of them is type **truth** and the other one is type **string**. The reason Atlas employs these two types is that they are the most suitable ones for propositional calculus. Truth type variable is essential for using propositional logic, for that holds the truth value of a variable (true, false or undefined). For printing statements on console and descriptive functionalities, string type is used.

Names of the truth type variables are denoted with single quote(‘). This allows users to name their variables as sentences; like ‘Man is mortal’ or ‘Earth is round’. This makes Atlas easy to read and adoptable. Strings are contained in double quotes. We wanted to stay loyal to common convention among other major languages. Sample string and truth type variable declarations and assignments are shown below:

```
string str = “This is the value of our variable: ”;

truth ‘Man is mortal’ = true;

string descriptiveResult  = str + ‘Man is mortal’;  ## Returns “This is the value of our variable: true”

int intValue = 12;

float floatVal = 5.2;
```

b) Assignment

Assignment operator is the “**=**” sign. Also, the words “**is**” and “**are**” can be used for assignment operation.

```
‘x’ = true;

‘x’ is true;

‘governments are corrupted’ is true;

‘expressions’ are false;
```

The reason for Atlas also allows the words **is** and **are** as assignment operators is making the programming language similar to natural languages, which increases readability and writability.

c) Truth Values

Type truth variables holds truth values, which is essential to propositional calculus, and there are three types of truth values in Atlas: "**true**", "**false**" and "**un**" (which stands for *undefined*). If true or false value is not assigned to a variable, variable's truth value becomes undefined. Also, programmers can assign undefined value to a variable by using "un". Declaration of type truth variables and assignment of the truth values are shown below:

```
truth 'Man is immortal' = false;

truth 'Democracy is a lie' = true;

truth 'There is a other side' = un;
```

d) Equality Check

Equality check operator is "==" and also it can be used with verbal keyword "remains"**.**

```
if 'truthValue' == un {## Do something}

if 'truthValue' remains un {## Do something}
```

e) Constants

Constants can be defined by using "**pure**" keyword. If "pure" keyword is added before the "truth", it means that variable's truth value is defined once and cannot be changed in below following lines of the program. Declaration of a constant is shown below:

```
pure truth 'Man is immortal' = false;

'Man is immortal' = true  ##Throws error
```

That means if x is assigned true, false or undefined, no other assignment can be made to 'Man is immortal' in the rest of the program.

f) Connectives

There are handful of connective logical expressions in the Atlas programming language. Infinite number of logic expressions can be derived from below connectives. No signs can be used for connective logical expressions “Nand”, ”Nor” and “Xor” so that there will be no complication at using above logical expressions and logical expression of “Not”.

Not

if truthValue remains not true {##}

if !(truthValue remains true) {##}

And

if param1 and param2 {##}

if param1 && param2 {##}

Or

if param1 or param2 {##}

if param1 || param2 {##}

If Then

if param1 -> param2 {##}

If and Only If

if param1 <-> param2 {##}

Not And

if param1 nand param2 {##}

Not Or

if param1 nor param2 {##}

Exclusive Or

if param1 xor param2 {##}

g) Predicates and Predicate Instantiations

Predicates correspond the functions which return true, false or undefined values.

They can be defined with two different formats:

```
pred  firstPred(param1:truth, param2:truth) { return param1 xor param2;}

firstPred =  (param1:truth, param2:truth) => { return param1 xor param2;}
```

The above statements demonstrate two different ways of defining a predicate. For the first style, “pred” keyword is used before the predicate name, then the predicate name and signature follows. Predicate body is contained in curly brackets. For the second style, there is no need for using pred keyword. Predicate name is assigned to a signature and then the double arrow indicates the predicate body. Statement below shows a predicate call.

```
firstPred(true, false);
```

h) Functions

Functions that return other values than truth variables. Functions can take truth values, integers and string variables as parameter. And they can return string, integer, boolean type values. So that they can be used to create Strings that correspond to explanation of a constructed logic operation or count the different possibilities that may come out from a logic operation. Functions can be defined in only one format:

```
function  firstFunc (param1:truth, param2:string) {

string out is param2 + “param1” + “is” + param1;

return out;

}
```

Statements above demonstrate the way to define a function.“function” keyword is used before the function name, then parameters in brackets and function body in curly brackets. And a function is called in the way below:

```
firstFunc(true, “Or operation produces”);
```

i) If , If - Else Statements

If statement is denoted with “if” keyword that is directly followed with a truth value or a type truth variable which holds a truth value. If the truth value is true, statement in between the curly brackets is executed. It is same for the “else if”. The structure is shown below,

where x and y are type truth variables:

if ‘x’ {##}

else if ‘y’ {##}

else {##}

Another practical way of making an if statement is using question mark (?) and colon (:). Truth value is followed by a question mark which indicates the statement that will be executed if the truth value is true, and if not, colon expresses the else condition that comes after it. The structure is shown below:

‘man is mortal’ ? say “we are mortal” : say “we are immortal”;

j) Loops

Loops in Atlas are based solely on propositional truth values. Therefore, Atlas provides a loop statement similar to while loops in other major languages. In Atlas, keyword for the loop statement is “as long as”. The structure for loop statement is shown below:

as long as ‘x’ remains true *{##}*

Assume that x is a type truth variable (that holds a truth value) and as long as its truth value remains true, the loop will continue. It could be the other way around (‘x’ remains false or un as the continuing condition for the loop).

k) Input / Output Statements

To output a statement, “say” command is used in Atlas. To exempilify, the following command prints “Hello World” on console:

say “Hello World”;

To take an input from the user, “get” command is used. The following command takes an input from the user (assuming we have a variable named str which is of type string):

get str;

l) Relational Mapping (Dictionary-a-like)

Relational mapping is a structure that is similar to dictionary for indicating two concepts are related (or not) to each other. A relational map can be defined in two ways. The first style is the following:

rel earthTruth = [“Earth” : “Our planet”];

As shown above, rel keyword is used for relational map. After the keyword, relational map’s name follows. On the right side of the assignment operator, relational map’s structure can be shown. It takes two string parameters which are intented to be related to each other, and a colon sign is put between these parameters. The whole expression is contained in square brackets.

Another way of defining a relational map is shown below:

$ =>[“Earth” : “Our planet”];

$ =>[“Earth” ! : “Flat”]; *## !: means two concepts are not related to each other, and it will return false*

$ => [“Cumhuriyet Bayrami: 29 Ekim”];

if earthTruth.“Earth” $ “Our Planet” {##} **##**Statement in curly brackets is executed since earthTruth is true (relational map is defined above).

m) Arithmetic Operations

Atlas also includes basic arithmetic operations: addition, substraction, multiplication and division.

Addition

int numericValue = 1 + 2;

int numericValur is 1 plus 2;

Substraction

int numericValue = 26 - 23;

int numericValue is 26 minus 23;

Multiplication

int numericValue = 6 * 6;

int numericValue is 6 times 6;

Division

int numericValue = 2 / 1;

int numericValue is 2 over 1;

n) Comments and Escape Character

Comments are identified with double hashtag (**##**) in Atlas and as the escape character (that makes special characters/keywords not special) backslash **(**) is used.

At the end of each statement, semi-colon (;) is used in Atlas as an indicator for end of the statement.

o) Beginning and End of a Program

“init” keyword is used for indicating the beginning of a program that is written in Atlas language. The double at sign , “@@”, is used for indicating the end of the program. An Atlas program is contained like the following (dots represent program statements):

init

.

.

.

.

@@

2) BNF Description

Variables and Types:

<var> → *<type>* *<var_name>*

<type> → *<numeric type>* | truth | string

<numeric type> → int | float

<constant> → pure *<var>* *<var_name>*

Assignment:

<assign> → *<var>* = *<var_value>* | *<var>* is *<var_value>* | *<var>* are *<var_value>*

<var_value> → *<int_value>* | *<string_value>* | *float_value* | *<proposition>*

Truth Values and Logical Operations:

<truth value> → *<truth value>* *<logic op>* *<truth value>* | true | false | undefined

<logic op> → and | && | or | || | xor | nor | nand | -> | <->

<negation> → not | !

<proposition> → *<proposition>* *<logic op>* *<proposition>* | <negation> <proposition> | *<truth value>*

Arithmetic Operations:

<expr> → *<expr>* + *<term>* | *<expr>* - *<term>* | *<term>*

<term> → *<term>* * *<factor>* | *<term>* / *<factor>* | *<factor>*
<factor> → *<exp>* ** *<factor>*

<exp>

<exp> → *(<expr>)* | *<var>*

<var> → *<int_value>* | *<float_value>*

If, If - Else Statements:

<if then statement> → if *<proposition>* *<statement>*

<if then else statement> → if *<proposition>* *<statement>* else *<statement>*

Equality:

<equality> → *<var>* == *<var_value>* | *<var>* remains *<var_value>*

Loop Statement:

<loop statement> → as long as *<proposition>* remains *<truth value>* *<statement>*

Predicates and Functions:

<predicate> → pred *<pred_name>* (*<parameter_list>*) *<pred_body>* | *<pred_name>* = (*<parameter_list>*) => *<pred_body>*

<pred_body> → *<statement>*

<function> → func *<func_name>* (*<parameter_list>*) *<func_body>*

<func_body> → *<statement>*

<parameter_list> → *<parameter>* | *<parameter_list>*

<parameter> → *<var>*

Relational Mapping:

<relational_map> → rel *<relational_map_name>* = [*<string_value>* : *<string_value>*] | $ => [*<string_value>* : *<string_value>*]

Comment and Escape Character:

<*comment*> → ##

<*escape_char*> → \