Abitha Mahabakkalage

3CAD Programming Language

The language manages shapes in a 3d environment, from the creation of said shapes to top translation, scaling and various modifications.

S.1 Arithmetic and expressions

3CAD can evaluate a variety of arithmetic expressions.

Integers and doubles alike can be stored to variables, and their value can be retrieved.

The expressions can include addition, subtractions, divisions and multiplications such as:

x * 3, 4 / 2.0, x + y

S.2 Create shapes

3CAD can create three four distinct shapes. Cuboid, sphere, cylinder and cube

Their shape can be customized when being created.

Cube(20), Sphere(12)

S.3 Shapes can be moved and scaled

3CAD can perform translations on instantiated shapes. It can also scale the dimensions of the shapes based on the user's input. **translateX(P1, P2) scale(P1, P2)**

S.4 Duplicate shapes

3CAD can duplicate an already existing shape and keep all of the modifications. The duplicate shape will be its own entity after being instantiated and can be modified independently from the original

Duplicate(P1)

S.5 Change the colour

3CAD gives access to the RGB values of shapes, therefore it is able to generate a wide range of colours that can then be assigned to shapes.

changeColour(1, 0.3, 0.7, 0.3, 1.0)

S.6 Predicates

3CAD allows the comparisons between 32-bit integer, and return a Boolean

based on the comparison

x > 0.3, x > y, 0.3 != 0.5 , x < 20, 12 <= 3289,

S.7 Display

3CAD displays the shapes in the 3d environment in a pop -up window. The tilted camera angle allows the clear display of the 3d features of the shapes

Display()

S.8 Variable Assigment

3CAD allows variables to be assigned to numerical values and Shapes alike and their values to be retrieved or further modified.

Assign(a, cube(30))

seq(C1, C2) execute command C 1 followed by command C 2

sub(E1: int32, E2: int32) integer subtraction: E 1 − E 2

gt(E1: int32, E2: int32) integer greater-than E 1 > E 2

lt(E1: int32, E2: int32) integer less-than E 1 < E 2

ge(E1: int32, E2: int32) integer greater-than or equal E 1 >= E 2

le(E1: int32, E2: int32) integer less-than or equal E 1 <= E 2

ne(E1: int32, E2: int32) integer not-equals E 1≠ E 2

assign(N: string, E: int32) bind E to name N in variables map

deref(N)) retrieve binding for name N

if(P: bool, C1, C2) select if P then execute C 2 else execute C 3

while(P: bool, C1, C2) iterate while P then execute C 2 else execute

sphere(R:int32) create a sphere with radius R

Cuboid(H: int32, D: int32, W: int32) create a cuboid with height H, depth D and width W

Cube(S: int32) create a cube with side S

Cylinder(H: int32, R: int32) create a cylinder with height H and radius R

Display() creates a window displaying the shapes

Translate(S: int32 ,X: int32, Y: int32) translates the shape S on the X and Y axis

changeColour(S:int32, R: int32, G: int32, B: int32, O: int32) changes the colour of shape S, using the RGB values R G and B and the opacity O.

Scale(S: int32, C: int32) multiplies the dimensions of the shape S by the constant C

Duplicate(S: int32) creates an indepented but exact copy of shape S

Informal Language Specification:

The informal language specification, as the name suggests, is a brief description of the language's features. As opposed to following universal standards, the descriptions are written using the highest level language available, English (although any other language is allowed).

As opposed to providing a description for each available feature, I decided to group them into intuitive groups, such as 'Arithmetic expressions', which includes all of the basic algebraic operations available, or 'Predicates', containing all of the comparisons available.

Furthermore, each group of features contains an example of how the statement would be written in the language. Although the exact form is different from the final product, in which the Syntax is a lot more intuitive and convenient. It still provides a general idea of the "look" of the language.

Internal Syntax Signatures:

This next section provided a more detailed look at the features of the programming language. Not only does it include all of the "functions" available, but also provides the arguments required. These argument are further described by their type. As "art" is based around the "Value" object, most functions take one or more subclasses of Value, such as int32.

Accompanying these signatures, which are far more formal than the ones in the informal language specification and somewhat resemble what is found in the eSOS rules, is a description of what the function does specifically.

Writing this section allowed me to have a clearer view of the language before beginning the implementation and I was constantly going back to using it as reference during development.

eSOS Rules For An Interpreter:

Before describing my implementation of the eSOS rules for the 3CAD language I will firstly explain the "ValueUserPlugin" Java class. Due to the language involving 3 dimensional shapes and relatively high level features, it was inevitable that another language with a more extensive set of libraries and predefined functions was used in the backend; ART uses a Java class to do so.

Figure 1. Java code showcasing the case system

Several eSOS rules, mainly the ones dealing with the shapes specifically will call the Java class. As each rule will require different Java code, a "Case" system is implemented where the first argument of the Java class is dedicated to deciding which piece of the code to use. As previously mentioned, Art uses "Value" objects to store the various results, therefore it is necessary to convert them to, in this case Integers, in order to correctly use the Java predicates.

Another vital feature of the Java class is that, whilst an eSOS rule is being implemented, the class is constantly running in the background. This is extremely important as it allowed me to store values that would be later referenced without them being deleted. This came in especially handy as I was able to, using a HashMap, store various shapes, that I would then be able to retrieve and modify conveniently, which is an imperative feature for a CAD based language.

Due to the nature of the language, the javafx library was extensively used. It allowed me to use "shape3D" to create the various shapes and additionally use the extensive selections of methods associated with the objects to alter the shapes, including changing their position, colour, and size.

Finally, the Java class not only receives inputs, in the form of an array of "Value" objects, but returns values as well that are later utilized in the eSOS rules. Its important to note that all of the various cases have a return statement, whether that be an Object, integer, or even "__done", which is used in the eSOS rule to signal that the task is completed.

The first thing I implemented in the eSOS rules, are the various arithmetic functions. These included the basic mathematical operations (addition, subtraction, multiplication and division) and the various predicates.

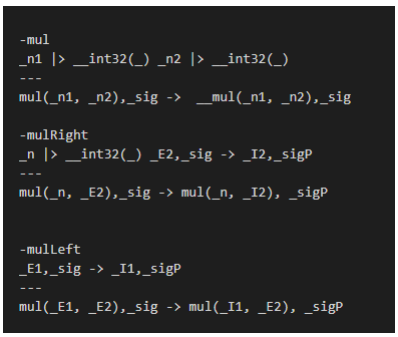

Figure 2. eSOS rule implementation of multiplication

These rules have very similar implementations, as an example I describe the one for the product of two integers. Once an eSOS “try” is run, it will try to match the arguments to a specific rule. As we are multiplying, the arguments must be numerical values, hence once they are integers, the predefined __mul() rule is called, which multiplies the two arguments. This, however raised an issue, as if one of the arguments wasn’t explicitly numerical, but let’s say was another mathematical expression, there wouldn’t be any matching rule. As such we add subrules that take an argument and evaluates it, and then calls mul() again with the updated arguments, until both are, in this case, integers. A similar method is used for all of the other expressions and predicates.

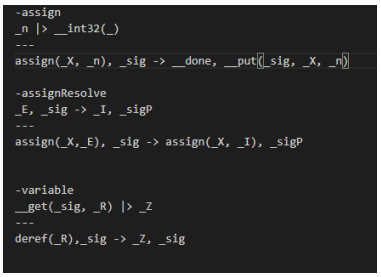

Figure 3. eSOS rule implementation of variable assignment

One important feature of the language that I knew was necessary is that it allows users to assign values to variables including Shapes, that can later be dereferenced to retrieve said value. As shown in *Figure 3* the assign rule requires the value to be an integer. As such Shape objects cannot be directly assigned to a variable. To circumnavigate this problem, I used a HashMap in the Java class, that would assign an integer, that keeps incrementing in order to prevent collisions, to a shape once created. Said integer is then returned to the eSOS rule and then assigned to a variable. This key is then used to retrieve the corresponding shape. In order to implement this feature, I had to make sure to evaluate the second argument of assign, allowing eSOS to evaluate a shape and retrieve the integer key. In order to get the integer value associated to a variable we used deref.

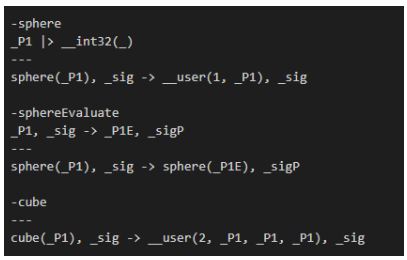

Figure 4. eSOS rule implementation of creation of Shapes

The language being used predominantly for CAD, it was inevitable that the majority of the rules would be in relation the various three-dimensional shapes. The rules displayed in *Figure 4* show the implementation of the sphere and cube rule. To instantiate these shapes, we have to use the Java class and the javafx library. In order to access the class we use __user, which calls the “ValueUserPlugin” with the corresponding arguments. As the case associated with the creation of a sphere is the first one, we simply pass “1” and another integer representing the radius.

The implementation for the cube is more interesting as there isn’t a javafx method to instantiate a cube specifically. As such, in order to create a cube I have to refer to the second case in the class associated with a

cuboid and provide the dimension of the three sides. As a cube has the same length for each side, I can simply include the length of the only input three times.

```
-scale
_n1 |> __int32(_) _n2 |> __int32(_)
---
scale(_n1, _n2), _sig -> __user(7, _n1, _n2), _sig

-scaleLeft
_n2 |> __int32(_) _P1, _sig -> _P1E, _sig
---
scale(_P1, _n2), _sig -> scale(_P1E, _n2), _sig


-scaleLeftRight
_n2 |> __int32(_) _P1, _sig -> _P1E, _sig
---
scale(_n2, _P1), _sig -> scale( _n2, _P1E), _sig


-scaleRight
_P1, _sig -> _P1E,  _sig
---
scale(_P1, _P2), _sig -> scale(_P1E, _P2), _sig
```

Figure 5. eSOS rule implementation of Scale function

The remaining set of provide functions that manipulate the shapes. These include, scaling the dimensions, translating, changing the colour and duplicating the various shapes. All of these have similar structures; it's important that the first argument is always evaluated as it will always be a variable associated to a shape. And as previously stated a variable must be dereferenced, which involves being evaluated. After the shape is retrieved, or more specifically the inter key associated with it, __user is again used to access the Java class, with the first value for the argument changing depending on the rule and the remaimng being used in the Java code.

External Syntax Parser Generating Internal Syntax Trees

!try seq(seq(seq(seq(assign(a, sphere(100)), translate(deref(a), 500, 200)), duplicate(deref(a))), translate(deref(a), 0, 200)), display), __map

Above is an example of a relatively simple eSOS "try". Due to its nested nature it becomes quickly impractical even for small sized programs. As such it is important to implement an external parser, that takes code written by the user in a form that is more ergonomic and then returns the corresponding eSOS rule that can then be evaluated by the machine.

To do so, the parser generates an internal syntax tree using a set of rules that I defined.

```
statement ::= seq^^ | assign^^ | if^^ | while^^ | display^^ | translate^^ | scale^^ | duplicate^^

seq ::= statement statement
```

Figure 6. First lines of the parser

A syntax tree is made of nodes and leaves, and in order to correctly translate a piece of code we need to correctly assign each element of the program the correct parents and children. As shown in figure one, we begin by defining a statement as a series of functions that the language possess. And right after we define a "seq" as a collection of two statements. This means that whenever two statements are found next to each other, they then become two children of "seq". When the Syntax tree is then converted to an eSOS rule set, the children of a node become its arguments, resulting in "seq(statement, statement)", where statement will

in practice be further translated into a valid expression. This works recursively due to a sequence of two statements being a statement itself allowing as to create large nests of “seq” that would normally be impractical to manually write.

```
translate ::= deref '->'^ 'translate'^ '('^ expression ','^ expression ')'^ ';'^
scale ::= deref '->'^ 'scale'^ '('^ expression ')'^ ';'^
duplicate ::= deref '->'^ 'duplicate'^ '('^ ')'^ ';'^
```

Figure 7. Parser rules for functions relating to shapes

The rest of the rules used by the parser are what determine the structure of the language. Similarly to eSOS rules, the parser will try to match a rule to a line of code in the program. As an example, the duplicate function will look for a variable (deref), followed by ‘-> duplicate ();’. The singular high hats are used to remove certain strings from the syntax tree. In this case all of the values are removed except for the variables referring to the shape, which is correct as duplicate only requires one argument.

The scale function require two arguments, as such both the variable and “expression” which would then be later further translated to an integer are retained in the tree as children of scale, discarding the rest of the strings.

```
expression ::= operand^^ | sphere^^ | cube^^ | cylinder^^ |cuboid^^ | add^^ | sub^^ | mul^^
sphere ::= 'sphere'^ '('^ operand ')'^
cube ::= 'cube'^ '('^ operand ')'^
cylinder ::= 'cylinder'^ '('^ operand ','^ operand ')'^
cuboid ::= 'cuboid'^ '('^ operand ','^ operand ','^ operand ')'^
add ::= expression '+'^ operand
sub ::= expression '-'^ operand
mul ::= expression '*'^ operand
```

Figure 8. Parser rules for expressions

As shown previously, expressions usually become children of functions. But expressions by themselves have no meaning in eSOS rules term, therefore they need to broken down further. Above is my implementation of expression, and as shown, they are all of the functions that can then be assigned to a variable. Just like the previous example, once a pattern is matched, only the relevant values are taken and then added to the tree. In this case, most of these values are operands, which again need to be broken down further as it holds no meaning in eSOS rules. Operand is either a variable, which will then be dereferenced, or an integer.

Attribute Evaluation Based Interpreter

In this section I will describe my implementation of an Attribute Grammar for the 3CAD language. An Attribute Grammar is unrelated to eSOS rules and provides a different way of translating and running code. Similar to eSOS, it uses a set of rules and patterns to translate the code, but unlike it, inhouse java code can be run to perform higher level tasks.

Although unrelated to eSOS specifically, my Attribute Grammar extensively uses the “UserValuePlugin” Java class, which is extremely convenient as most of the code written for eSOS is reused.

```
support {
  ITerms iTerms = new ITermsLowLevelAPI();
  Value variables = new __map();
}

statements ::= statement
| statement statements
```

Figure 9. Attribute Grammar firt lines

Figure 9 perfectly encapsulated the differences and similarities of eSOS rules and the Attribute Grammar. The main difference being shown at the start in which Java code is written directly, as opposed to using __user to refer to the class. And the similarity being the way both look for patterns in the code that match the rules and use recursion to concatenate statements. The variable map is used to store the various variables in the code, whilst iTerms is used to access the plugin Java class.

```
| 'for' condition< '|' statement< '|' statements< 'end'
  { artEvaluate(statement.condition1, condition1);
  while (((__bool) condition1.v).value()) {
    artEvaluate(statement.statement1, statement1);
    artEvaluate(statement.statements1, statements1);


    artEvaluate(statement.condition1, condition1);
    }
  }
```

Figure 10. Implementation of a for loop

Above is my implementation of a for loop. As a syntax tree isn’t built, there isn’t a necessity to remove the unnecessary strings such as brackets and slashes. The Attribute grammar will store the values not in quotation marks, and allow me to then reference them inside the curly bracket, where I can write Java code using said variables. In the case of the for loop, the three vales are “condition”, which is a predicate that, when met will end the loop; a statement that will constantly be evaluated and usually will eventually lead to the “condition” to be met; and finally “statements”, which as the name suggests, is the series of next statements until the string ‘end’ is encountered.

```
| 'display' '('  ')' ';' {iTerms.valueUserPlugin.user(new __int32(4,0));}

| expression '->''translate' '(' expression ',' expression ')' ';' {iTerms.valueUserPlugin.user(new __int32(5,0), expression1.v, expression2.v, expression3.v);}
```

Figure 11. Attribute Grammar rules for display and translate

Above is the implementation of functions in the Attribute Grammar. Similarly to eSOS rules, the Java class is used to carry more complex functionalities. The “Case” system is still used to access the correct part of the code, with the first parameter being an the appropriate integer (it is worth noting that since the user method in the java class requires the parameters to be “Value” objects, __int32 must be passed as opposed to a normal integer), and the other parameters being the expressions not in quotation marks. Similarly to Java, a semicolon is used to distinguish the end of an expression.

```
condition<v:Value> ::=
  expression { condition.v = expression1.v; }
| condition '>' expression { condition.v = condition1.v.__gt(expression1.v); }
| condition '<' expression { condition.v = condition1.v.__lt(expression1.v); }
| condition '>=' expression { condition.v = condition1.v.__ge(expression1.v); }
| condition '<=' expression { condition.v = condition1.v.__le(expression1.v); }
| condition '!=' expression { condition.v = condition1.v.__ne(expression1.v); }

expression<v:Value> ::=
  operand { expression.v = operand1.v; }
| 'sphere' '(' expression ')' { expression.v = iTerms.valueUserPlugin.user(new __int32(1,0), expression1.v);}
| 'cube' '(' expression ')' { expression.v = iTerms.valueUserPlugin.user(new __int32(2,0), expression1.v, expression1.v, expression1.v);}
| 'cuboid' '(' expression ',' expression ',' expression ')' { expression.v = iTerms.valueUserPlugin.user(new __int32(2,0), expression1.v, expression2.v, expression3.v);}
| 'cylinder' '(' expression ',' expression ')' { expression.v = iTerms.valueUserPlugin.user(new __int32(3,0), expression1.v, expression2.v);}
|  expression '-' operand { expression.v = expression1.v.__sub(operand1.v); }
|  expression '+' operand { expression.v = expression1.v.__add(operand1.v); }
|  expression '*' operand { expression.v = expression1.v.__mul(operand1.v); }
```

Figure 12. Attribute Grammar rules for condition and expression

Similar to eSOS, expressions by themselves don’t hold meaning, but can be further broken down, and as shown above, an expression is any value that can be assigned to a variable. These include the various shapes, and the result of mathematical operations.

A condition on the other hand is broken down into predicates that return Boolean. These are used in for, if, and while statements.

Example domain specific programs

To test the language, both the one using eSOS interpreter and the Attribute Grammar, as there are slight differences between the two, I’ve created four programs that can be run.

```
y <- 1;
Sphere <- sphere(50);
Sphere -> translate(0, 100);
for y < 150 | y <- y + 1; |
    Sphere-> translate(20,0);
    Sphere-> changeColour(y + 20, 0, 0, 1);
    Sphere-> duplicate();
    end
```

Figure 13. Snippet of 3CAD program

As shown in the above figure, 3CAD allows the user to assign a variable to both a numerical value and a shape. Note that ‘<-‘ is used to assign values, whereas ‘->’ precedes functions. Duplicate will create an exact copy of a shape in the same location. The for loop, creates 149 instances of the sphere, with the colour slightly changing creating a gradient. To end a for loop, ‘end’ must be included, this is in order to separate future statements.

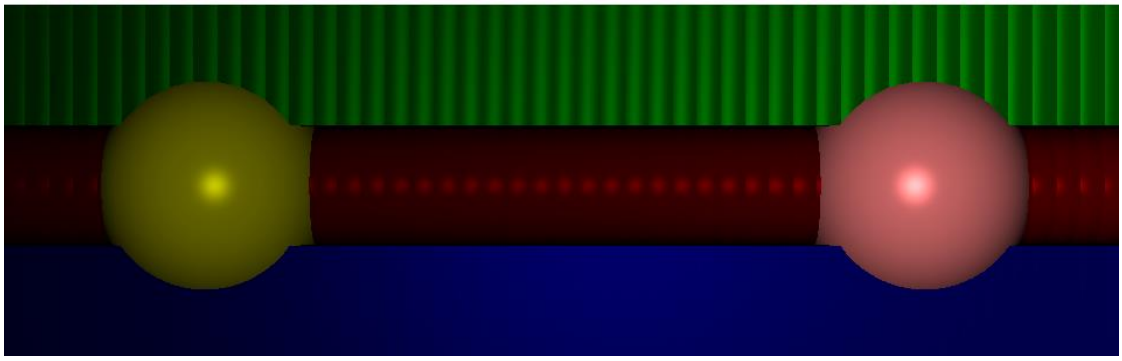

Figure 14. Output of the 3CAD script

Conclusion

In the years I have spent coding I always took Programming Languages for granted, a mere tool used to create interesting and complex work but embarking in this project instilled a newly found appreciation for the intricacy and hard work that went into modern programming languages. The attention to detail required in order to create a cohesive format and minimize issues is truly amazing.

The lack of extensive amount of similar work on the internet, forced me to truly use my problem-solving skills as opposed to simply scouring the web for solutions.

Overall, this has been one of the most rewarding experiences I have encountered in programming; a task that initially felt insurmountable, to create a new Programming Language, was then tackled piece by piece and completed.