Scratch2Catrobat Converter

Ralph Samer

2017-05-21

Note

- This presentation provides a brief introduction to the Scratch 2 Catrobat Converter application
- Before you start programming you should understand and be fully aware of all the discussed concepts
- Please excuse any typos and orthographic mistakes!
- I’ve not enough time to find and fix them ;)

What is Catrobat?

- it is a visual **programming language** (open source, GPLv3)
- Catrobat has been inspired by Scratch (the very popular visual programming language from MIT that runs on Flash-capable web browsers)
- **Catrobat programs** can be written by using the Catroid programming system on Android phones and tablets, using Catroid, or Catty for iPhones
- The **Catrobat project** is a large (umbrella) project consisting of many subprojects:
  - Scratch2Catrobat (that's us!)
  - Catroid (responsible for "Pocket Code" app on Android)
  - Catty (responsible for "Pocket Code" app on iOS)
  - HTML5 (responsible for "Pocket Code" webapp)
  - Webteam (responsible for the "Online Catrobat Store", where users can upload, share and download Catrobat programs of other users)
  - … many more others

What is Pocket Code?

- It's the **official name of the mobile app** that is released on Google's Play Store (for Android) and Apple's App Store (for iOS)
- It interprets programs that are written in the Catrobat Programming Language
- You can download the Pocket Code app for Android via this link:
  - https://play.google.com/store/apps/details?id=org.catrobat.catroid&hl=en
- **Note:** the Pocket Code iOS app is expected to be published on the App Store later this year

What is Scratch2Catrobat Converter?

- A tool for converting Scratch projects into Catrobat programs
  - it converts (or actually compiles) Scratch code files (JSON) into Catrobat code files (XML)
  - it detects Scratch blocks that are not yet supported by the Catrobat Programming Language (i.e. no Catrobat bricks/scripts exist for such Scratch blocks/scripts) and replaces them with Note-Bricks
  - it also checks for audio and image files that are not playable/displayable on Android or iOS (due to app or OS incompatibility reasons) and converts them into other compatible media formats (e.g. SVG files are converted into PNG files)
  - it provides a very simplistic web interface (hosted on one of our servers) to interact with users (about to be published soon)

Major goals

- Fill the gap between well known Scratch system (over 13 million projects as of February ‘16) and our Pocket Code project
  - it empowers kids to run their self-made Scratch projects on their own phone
  - That means, users can create projects on their PC/notebook
    - ... and then they can open and run their (converted) projects on their smartphones/tablets
    - … and since converted projects are real Catrobat programs, kids can even edit and modify their projects on their smartphones/tablets (so there is no need to switch back to the PC for every single change and reconvert the whole Scratch program)
- Fast conversions: users don’t have much time to wait…
- Produce accurate results
  - … but Scratch has some PC-related blocks:
    - e.g. WhenKeyPressed Scratch block: the problem is => we don’t have enough space for showing the entire (virtual) keyboard on the smartphone while the (converted) Catrobat program is running. So, this requires us to implement a workaround by adding an additional Sprite Object to the Catrobat program that shows the keyboard key as a button and when pressed triggers a broadcast to the original WhenKeyPressed script (that has been replaced by a Broadcast script)
  - … but Scratch programs are designed to run on larger displays and Scratch uses other coordinate system than Catrobat
    - we have to take this into account when converting motion blocks, …
  - ... but some Scratch blocks do not behave the same way as their Catrobat equivalents do:
    - this forces us to use tricky workarounds

Overview of the whole Scratch2Catrobat project

- Consists of the following 3 (independent!) applications:
  - Scratch2Catrobat Converter
    - the converter itself
  - Source Code Filter
    - needed to extract Catroid Class hierarchy from Catroid source files
  - Web Application
    - simplistic web interface and a Websocket JavaScript client (client-side)
    - WebSocket application (server-side)
      - handles and maintains conversion requests
      - creates jobs for those requests and puts them into queues
      - provides a TCP server to communicate with workers
    - Workers (server-side)
      - are responsible for the conversion process on the server
      - fetch jobs from queue and process them
      - a TCP client is integrated in order to connect to the (central) TCP server

Note

- But this presentation is only about the converter itself
- We are not going to dive deeper into the other two subprojects (Source Code Filter and Web Application)

- So, let’s continue with the converter!

Architecture of the Converter

- It consists of two major parts that are implemented in two separate modules:
  - Converter Frontend (Scratch module):
    - parses the Scratch project
    - analyzes and validates the Scratch project
      (checks for data inconsistency issues, e.g. whether the JSON code file contains references to media files that are not present in the Scratch project)
    - **each** Scratch script is translated into a **Script Element Tree**
      (analogous to Abstract Syntax Trees (AST) as you may already know from real compilers)
  - Converter Backend (Converter module):
    - transforms Script Element trees into Catrobat scripts
    - uses Catroid's **Java** class hierarchy to generate and persist the XML code for these Catrobat scripts
    - is also responsible for media file conversions

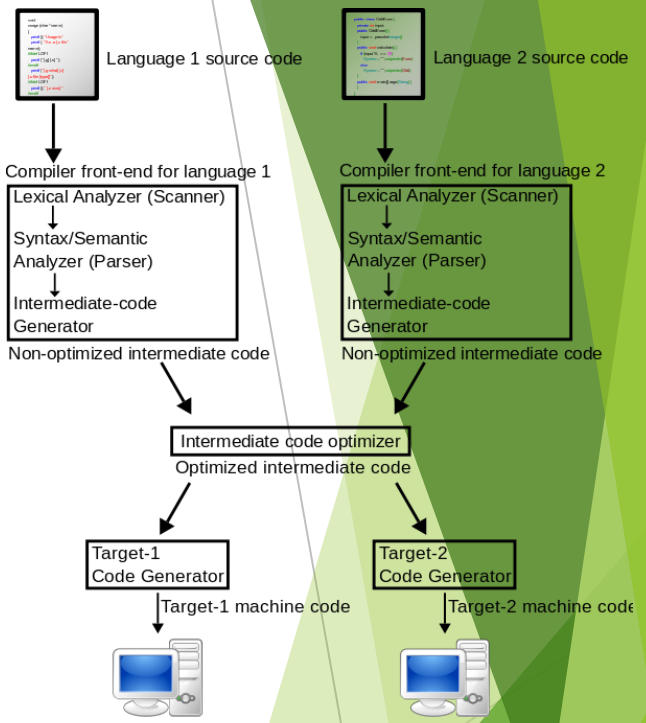
Code-conversion compared to a modern compiler

- **Input:** project.json file (instead of e.g. .c file)
- **Output:** code.xml file (instead of binary)

- **Compiler frontend:**
  - Lexical Analyzer
    - done by the JSON-parser
  - Syntax/Semantic Analyzer
    - for JSON-code this is also done by the JSON-Parser
    - for Scratch code within the JSON this is done automatically by our Traverser in converter.py module (see: later)
  - Create Abstract Syntax Tree (AST)
    - our scratch-module transforms the given JSON-code into object hierarchy/tree (see later)
    - so this can be understood as the creation of an AST tree for the Scratch code
  - Our scratch module (scratch.py) acts like a compiler frontend (therefore we name it **converter frontend**)

- **Intermediate code optimizer:**
  - for us this step is left for future work
  - in order to add such an optimizer one may create a new module and call it right after the frontend returns the Scratch project and immediately before the converter backend is called

- **Compiler backend:**
  Convert AST tree into binary
  - for us it's pretty much the same, despite the fact that in our case we generate a xml file instead of a binary
  - Our converter module acts like a compiler backend (therefore we name it **converter backend**; see later)



Source: Wikipedia.com

Technical requirements for the converter

- Jython 2.7 (strong dependency)
  - due to Catroid's **Java class hierarchy** (see: lib/catroid_class_hierarchy.jar)
  - but the converter also uses many other Java libs (e.g. jsoup, xstream, batik, …)
  - the converter code itself (except Java class hierarchy) is fully written in Python 2.7
- JVM and Java 8 (strong dependency)
  - the JVM is needed in order to run the Jython interpreter
  - Java 8 is required by catroid_class_hierarchy.jar
- Python 2.7 (e.g. CPython) (weak dependency)
  - for invoking the run script
  - but you can also call the script via Jython
  - then CPython will not be required any more

What is Jython?

- It's simply an interpreter for the Python Language that is implemented in Java
- Currently version 2.7 of Python is supported (Jython 2.7)
- Only supports a subset of the whole Python Language
  (but nearly all important features are supported)
- Since Jython is a normal Java application,
  it can only run on a Java Virtual Machine (JVM)
- Also the other JAR archives we are using in our Converter application contain Java class files that can only be executed by a JVM
- That's the reason why we need a JVM!

Modules of the Converter

- There are **two main modules** (as mentioned before; see Architecture slide)
  - **Scratch module** (Converter frontend)
  - **Converter module** (Converter backend)
- many other helper modules, e.g.
  - **main.py:** controls and orchestrates interaction between other modules, … (see later)
  - **scratchwebapi.py:** responsible for accessing the Scratch web API
  - **catrobat.py:** handles instantiation of some complex Catroid Java classes (e.g. Formulas, FormulaElements, …)
  - **common.py:** is intended to contain all Scratch-relevant (i.e. application-dependent) and Java-dependent helper functions (i.e. the common-module cannot be reused by other normal Python-applications => Jython is required)
  - **tools/helpers.py:** contains many common (i.e. application-independent) helper functions that do not use java-code (they can also be used by other normal Python-applications)
    - but at the moment the difference between the common- and the helper-module is not so clear (needs to be refactored!)
  - …
- and **two setup scripts** for the bootstrapping process
  - the **“run”** Python script (see next slide):
    - helper script for starting the converter
  - the **“run_tests”** Python script:
    - helper script to run tests

Run script (run[.py])

- Main entry point of the converter application
- Requires Python 2.7
- Tasks:
  - Bootstrapping
  - Basic setup
    - environment variables, paths (e.g. JYTHONPATH, …)
    - e.g. create “data” directories and subdirectories
    - Git commit hook
    - …
  - Call Jython interpreter and start main.py module (next slide)

Main module (main.py)

- From now on we are running in a Jython environment
- Tasks:
  - Parses and validates command line arguments
  - Downloads/opens and extracts scratch archive (.sb2)
  - Orchestrates interaction between other modules (scratch, converter, …)

Scratch module (scratch.py)

- Transforms given JSON-code into object hierarchy/tree:
  - Project-class is the root of this hierarchy (and inherits from RawProject) and contains:
    - *self.project_id:* scratch project ID (if available)
    - *self.name:* name of the project (if available)
    - *self.objects:* list that holds all Scratch Sprites as instances of Object class in it
      - getters/setters to access all json keys within a object child or stage object
      - *self.scripts:* contains all scripts as instances of Script class
        - *self.type:* type of this script: whenGreenFlag, whenIReceive, …
        - *self.script_element:* the root of the **Script Element Tree** (represents first block in the script) (instance of Script Element class)
          - *self.name:* name of the Scratch block
          - *self.children:* the root elements (i.e. other script elements) of the subtrees of this root script element (see next slide)
        - *self.arguments:* arguments that this script has e.g. receive-message if it is a whenIReceive script
    - *self.resource_names:* list with all MD5-hashes of costumes and Sounds
    - …

Script Elements

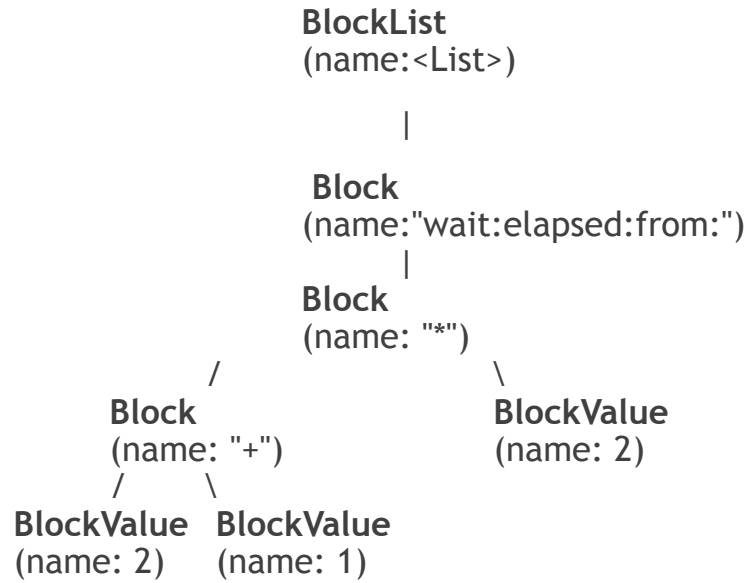
- There are 3 types of Script Elements
  - Block-List
    - represents an entire Scratch script or a nested block (i.e. Loop- or If-Else-sequences)
  - Block
    - represents a single non-nested block (e.g. `wait:elapsed:from:`)
  - Block-Value
    - represents a raw value (integer, float, string)
- all these 3 types are Python classes that inherit from the Script Element base class
- The Script Element base class uses a python-list (as property) to store all child Script Elements of this Script Element => that's how the Script Element tree is connected

Scratch module (scratch.py)

Script Element Tree Example

- JSON-code of a Scratch script:
  `["wait:elapsed:from:", ["*", ["+", 2, 1], 2]]`

- corresponding Script Element Tree:



Converter module (converter.py)

- The engine of the conversion process
- Tasks:
  - Preprocessing
    - Creates Catrobat project
    - registers/informs Data-Container (see Catroid-class-hierarchy) about user-lists and user-variables
  - Conversion
    - Sprite object conversion (implemented in `_ScratchObjectConverter` => see next slide)
      - transforms all Script Element Trees of this sprite object into Catrobat scripts
  - Postprocessing
    - Converts media files (if they are not compatible with Pocket Code or Android/iOS)
    - Adds default behavior in order to initialize Sprite objects (e.g. PlaceAtBricks, RotationBricks to update/correct the placement of the Costume/Look of the Sprite Object in the stage immediately after the Catrobat program is started on the smartphone/tablet)
    - …
  - Updates XML Header
    - information about the converter, description text, license URLs, … is added

Converter module (converter.py)

_ScratchObjectConverter

- called for each Scratch sprite object (see: `def __call__(self, scratch_object)`)
- responsible for Sprite object conversion
- i.e. actually converts complete Scratch sprite objects into Catrobat sprite objects
- Tasks:
  - Creates a new instance of the `org.catrobat.catroid.content.Sprite` Java-class (see Catroid class hierarchy)
  - Performs script conversion (for each script in the Scratch sprite object)
    - unsupported scripts are replaced by a Start script following a Note and a Wait brick
    - supported scripts are converted as you might expect
    - uses Visitor Pattern in order to convert all blocks within the script (next slide)

Converter module (converter.py)

_BlocksConversionTraverser

- implements a slightly modified version of the original Visitor Pattern
- traverses the `ScriptElement`-tree in depth-first order and uses a helper stack and stack pointer for this
- the helper stack contains the already visited script-elements + intermediate conversion results (e.g. values or already converted Catrobat bricks or formulas) that are arguments used as inputs for the currently visited/converted block
- the intermediate conversion results are always on top of the already visited script-elements (they are arguments for the next ScriptElement => see next line)
- the stack pointer (called arguments_start_index) is used to keep track of the start position of the arguments on the stack that are needed as inputs for the currently converted block
  - thus, when the stack pointer does not refer to the last element on the stack, we have arguments that get passed to the block-conversion-handler or lambda expression (see later in the slides) of the currently visited block
  - otherwise (i.e. the currently visited Script Element is on top of the stack) the block-conversion-handler, lambda expression or regular-block-converter does not expect any arguments for converting the currently visited block
- the stack pointer can help us finding conversion issues caused by invalid structures within the Scratch JSON file

Visiting a Script Element

- Visiting a Script Element means converting this Script Element
- The conversion of a script element depends on its type
- As mentioned before there are 3 types of Script Elements
  - Block-Lists
  - Blocks
  - Block-Values
- The Traverser uses depth-first-search in order to visit the Script elements
  - (0) at the beginning the stack is empty
  - (1) the traverser starts with the root-element of the Script Element Tree as the current script element
  - (2) the **name** of the script element is put on top of the stack
  - (3) check if the current script element has children, if there are children:
    - save the current element in a temporary variable
    - for each child set it as the current element and recursively call (2)
    - restore the current element from the temporary variable
  - (4) visit current element

Stack

- So on the stack we only have names of the **traversed** Script Elements
- The names for the Script elements depend on the type:
  - in case of a Block-List:
    “<list>”
    (constant string that acts as a marker/hint for the traverser)
  - in case of a Block:
    the block name as string, e.g.
    “wait:elapsed:from:”, …
  - in case of a Block-Value:
    the value itself, e.g.
    “Hello”, 2, 3.1415, …
- The **stack pointer (argument start index)** always refers to the index followed right behind the name of the currently visited Script Element on the stack

Visiting a Block Value

- Since a block value is a raw type (string, integer, float), there is nothing to do for us except leaving the name (value) on the stack and marking the element as visited in the tree!
- But it might be an argument of a math-operation Scratch block (e.g. +, -, *, random, …)
  - in this case it will get converted into a formula-element later when the corresponding operator is visited (as we should see later)

Visiting a Block (I)

- an instance of our Block-class represents a non-nested Scratch block
- Types of non-nested Scratch blocks
  - Action-blocks, e.g. wait:elapsed:from:
  - math-operation-blocks, e.g. “+”, “-”, “*”, random
  - Broadcast-blocks
  - User-defined-blocks
  - …

Visiting a Block (II)

- **Processing steps:**
  1. If there are arguments on the stack => take and remove them from the stack
  2. Take and remove the name (block-name) from the stack as well
  3. Fetch the corresponding object from the **complete_mapping dictionary** for the name provided as key
     - The corresponding object can be:
       - Java Class of the corresponding Catrobat Brick (if there is a 1:1 mapping or a conversion-handler exists)
       - Java Enum of the corresponding math operator/function (if there is a 1:1 mapping and the currently visited block is a math-operation-block)
       - lambda-Expression (otherwise)
  4. Fetch the corresponding block-handler (if exists!) from the **_block_name_to_handler_map dictionary** for the name provided as key
  5. Perform the conversion
     - If a block-handler exists: the handler is called (see next slide)
     - If no block-handler exists: perform regular block conversion
       - If the corresponding object is a lambda expression it gets called
       - If the corresponding object is a Java Enum a FormulaElement is created and all stack-arguments (either one or two) of this currently visited element on the stack are converted into FormulaElements (if they are raw values and not already FormulaElements; see Block-Value slide) as well and then passed to the FormulaElement as a left and/or right child of the currently visited element
       - If the corresponding object is a Java Class it gets instantiated and all stack-arguments are passed to the constructor of this class
       - But this is only a sneak peek of what the _regular_block_conversion method actually does.
         For more details on this, please take a look at the code of the _regular_block_conversion method
  6. Put the converted result on the stack

Visiting a Block (III)

Block-Handlers

- They are used for converting “complex” blocks where
  - a simple instantiation is not possible
    i.e. brick that takes arguments and has no standard constructor!
  - a lambda expression would be impractical or too complex
- How to implement a handler?
  1. Add a new entry in the complete_mapping dictionary for the block with the block-name as key and Catrobat brick’s class name as value,
     e.g. *"wait:elapsed:from:": catbricks.WaitBrick*
  2. Create a new method in the _BlocksConversionTraverser class and insert the register-handler-decorator-line right before you declare the method-signature
  3. Finally implement this method
     - you can access your Catrobat brick’s class of this block (via self.CatrobatClass) that you just added to the dictionary before and also access the stack-arguments via self.arguments
  - A complete example may look like this:

```
@_register_handler(_block_name_to_handler_map, "wait:elapsed:from:")
def _convert_wait_block(self):
    [duration_formula_element] = self.arguments
    assert isinstance(duration_formula_element, catformula.FormulaElement)
    return catbricks.WaitBrick(catformula.Formula(duration_formula_element))
```

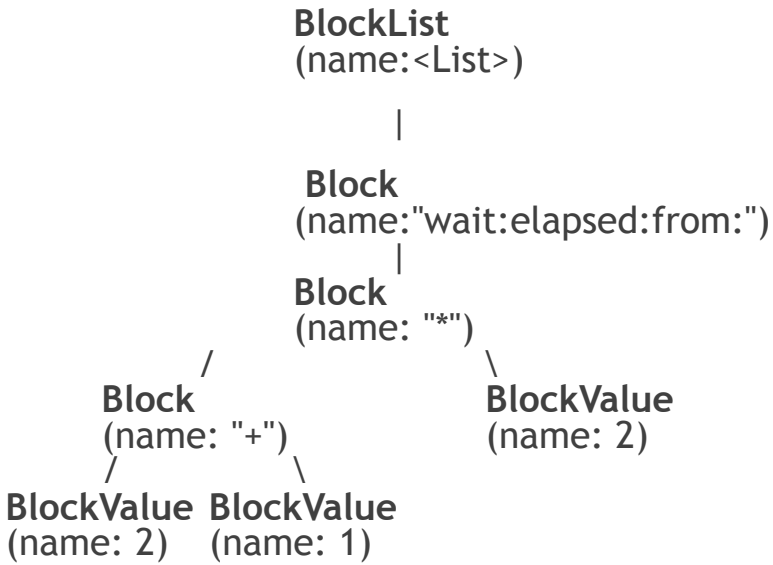
Visiting a Block List

- When the Traverser visits/hits a block list, all containing blocks and block values of that block list have already been visited before
- Thus, a block-list is only a hint for the traverser that the end of a fully converted block sequence is reached
- All arguments (i.e. all elements) that are on top of this block-list on the stack are Catrobat-bricks (in the correct order!)
- **Processing steps:**
  1. Take all arguments/bricks and remove them from the stack
  2. Take and remove the block-list from the stack (its only a marker/placeholder/hint)
  3. Put all arguments/bricks into a single python-list
  4. Append the list as a **single (!)** element to the stack
- Now if this block-list is the root element of the ScriptElement tree, all elements of the ScriptElement tree have been visited and there is only one single element left on the stack (our python-list with our Catrobat-bricks)
  - in this case a Script is created and the python-list element from the stack is passed to the script (all bricks are added to the brick-list of this script)
- But, if this block-list is not the root element of the ScriptElement tree, than this single python-list will be an argument for the nested-block (any Loop or If-Else block) that is going to be converted/visited in the next step

Converter module (converter.py)

BlocksConversionTraverser Example

- Now, let's take a look at our previous example again and apply our Depth-First-Order conversion algorithm on the Script Element Tree.
  Just to remind you, the Script Element Tree has already been generated by our scratch-module (for the given JSON-code `["wait:elapsed:from:", ["*", ["+", 2, 1], 2]]`).

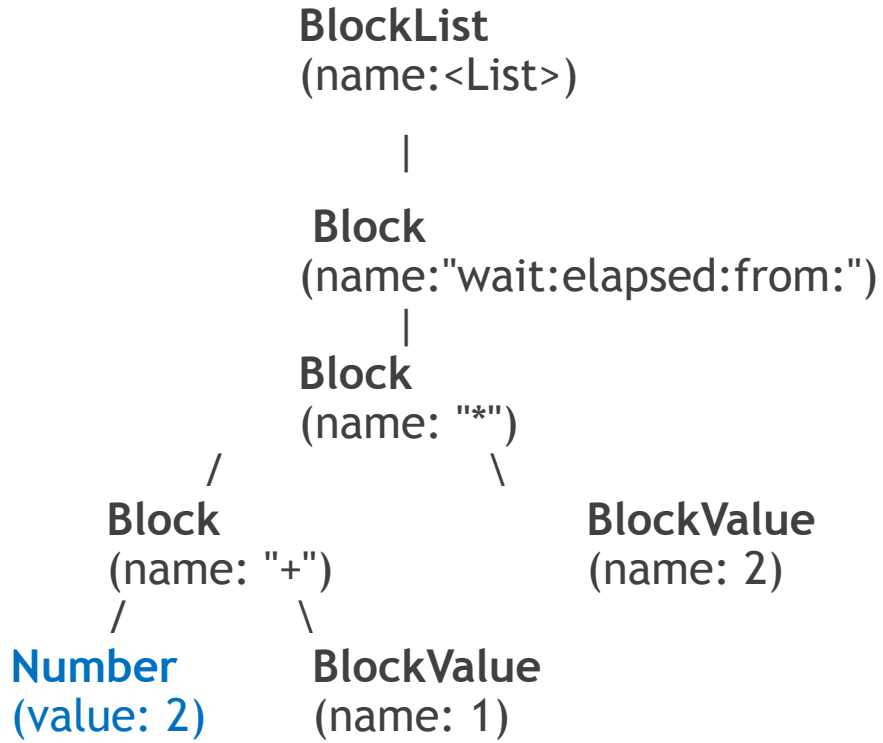


The currently visited element is highlighted blue

Already visited elements are highlighted green

Converter module (converter.py)

BlocksConversionTraverser Example

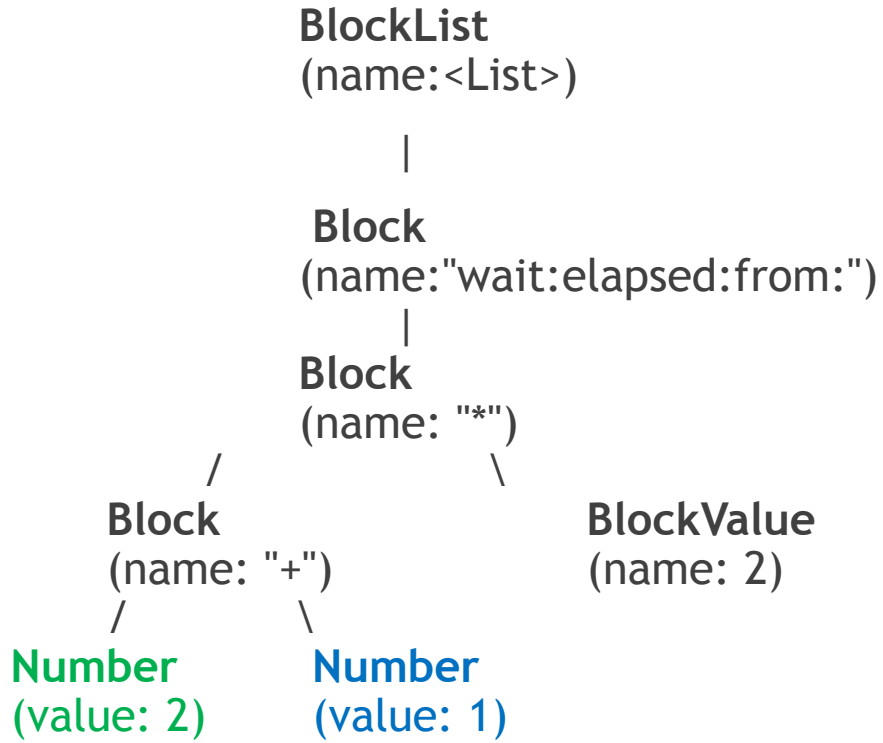


The currently visited element is highlighted blue

Already visited elements are highlighted green

Converter module (converter.py)

BlocksConversionTraverser Example

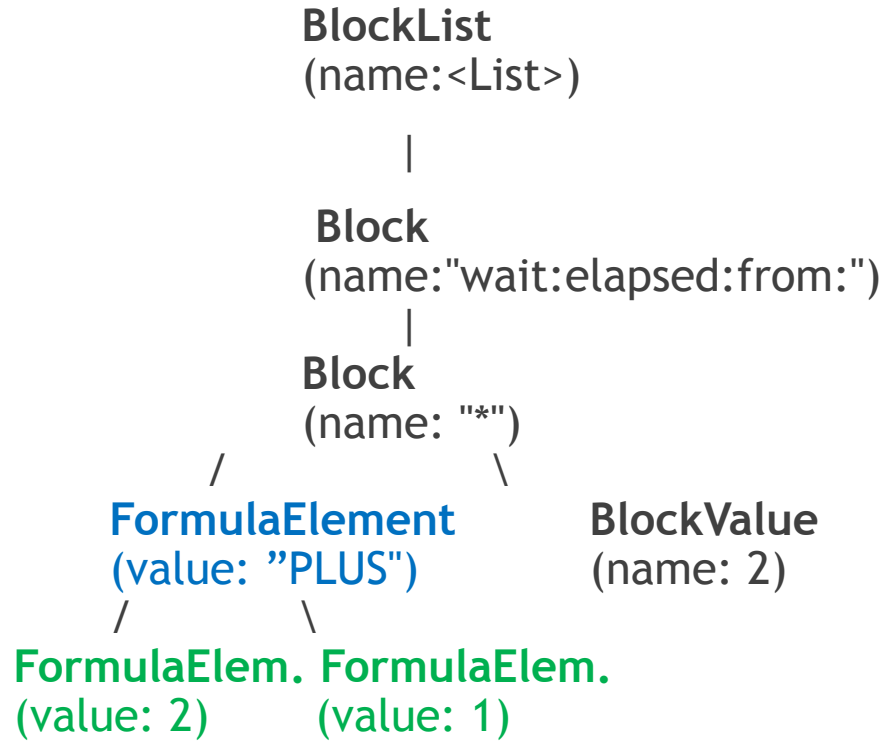


The currently visited element is highlighted blue

Already visited elements are highlighted green

Converter module (converter.py)

BlocksConversionTraverser Example

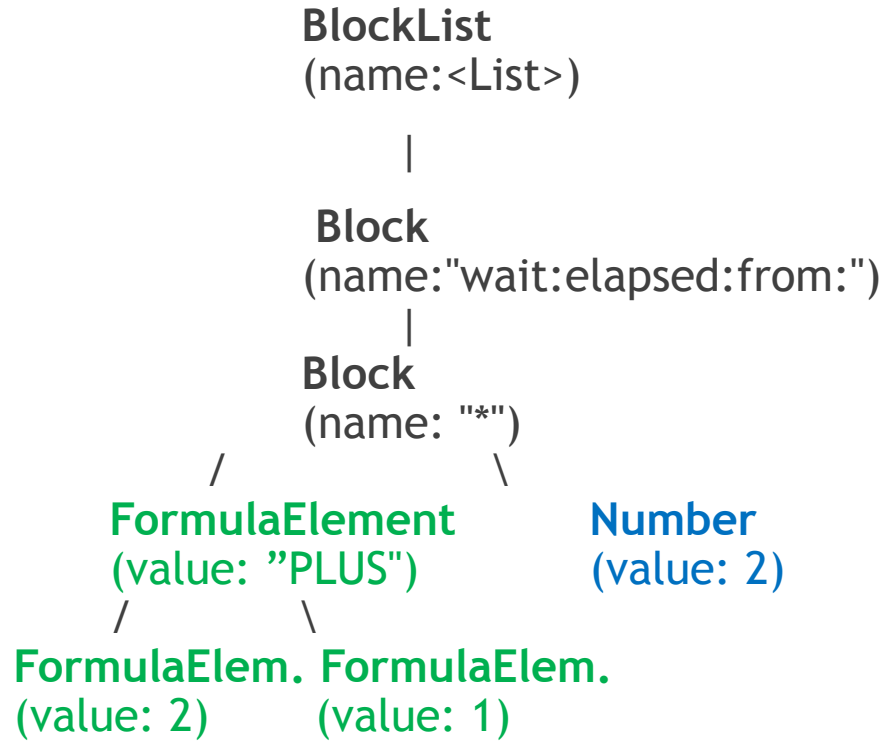


The currently visited element is highlighted blue

Already visited elements are highlighted green

Converter module (converter.py)

BlocksConversionTraverser Example

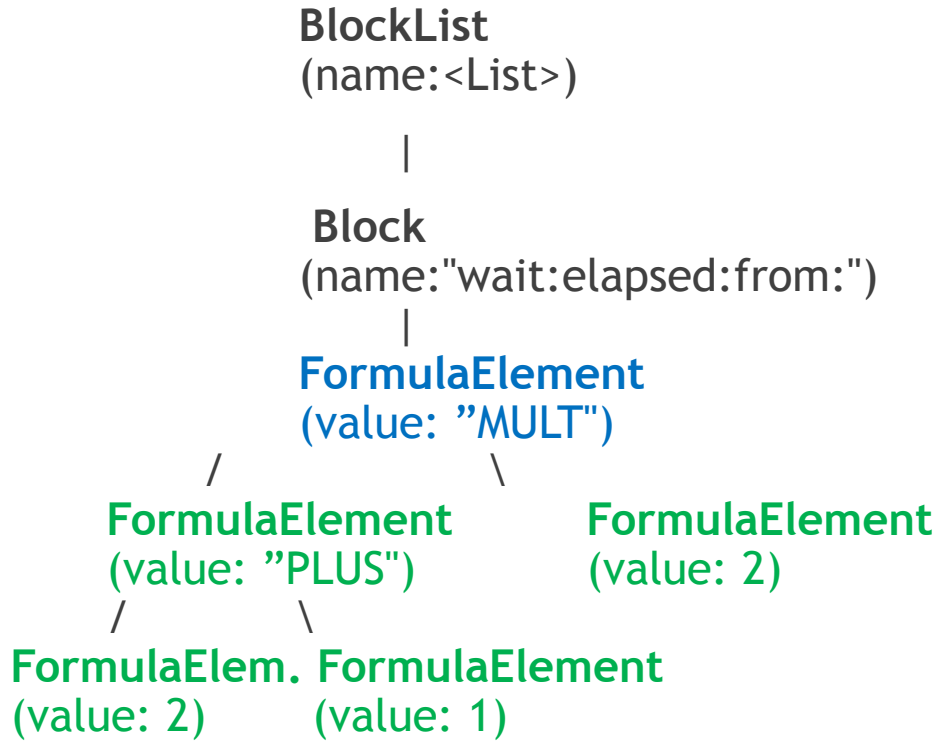


The currently visited element is highlighted blue

Already visited elements are highlighted green

Converter module (converter.py)

BlocksConversionTraverser Example

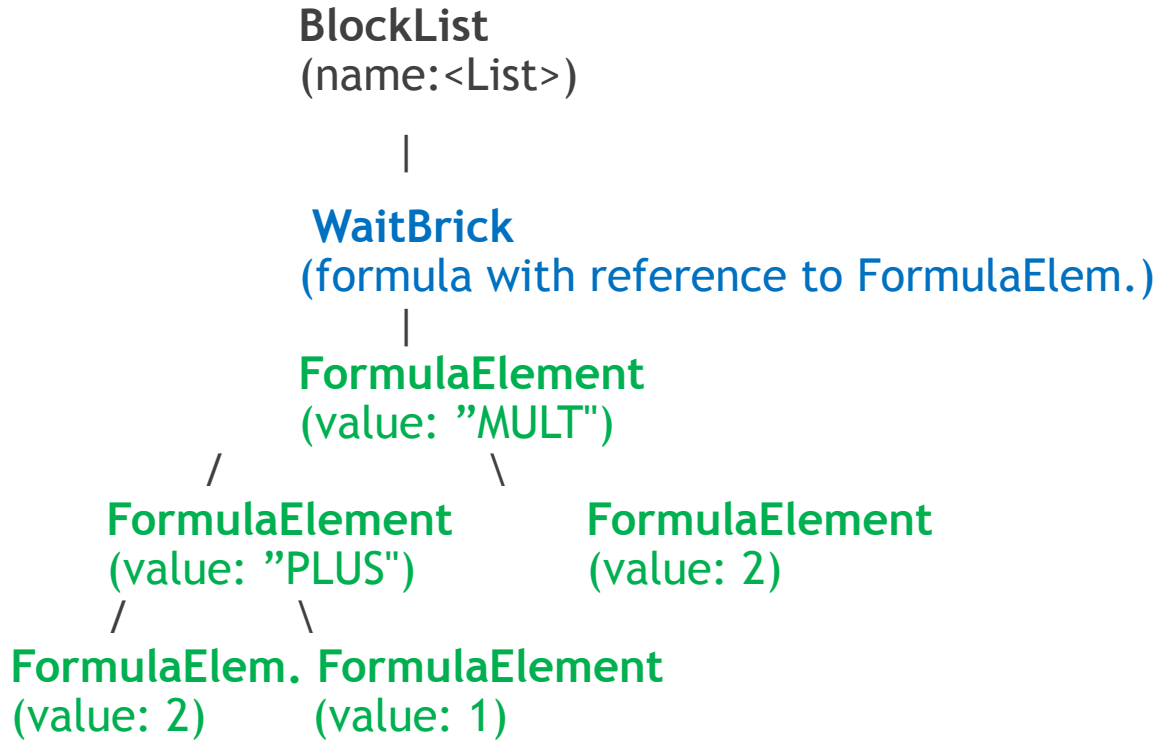


The currently visited element is highlighted blue

Already visited elements are highlighted green

Converter module (converter.py)

BlocksConversionTraverser Example

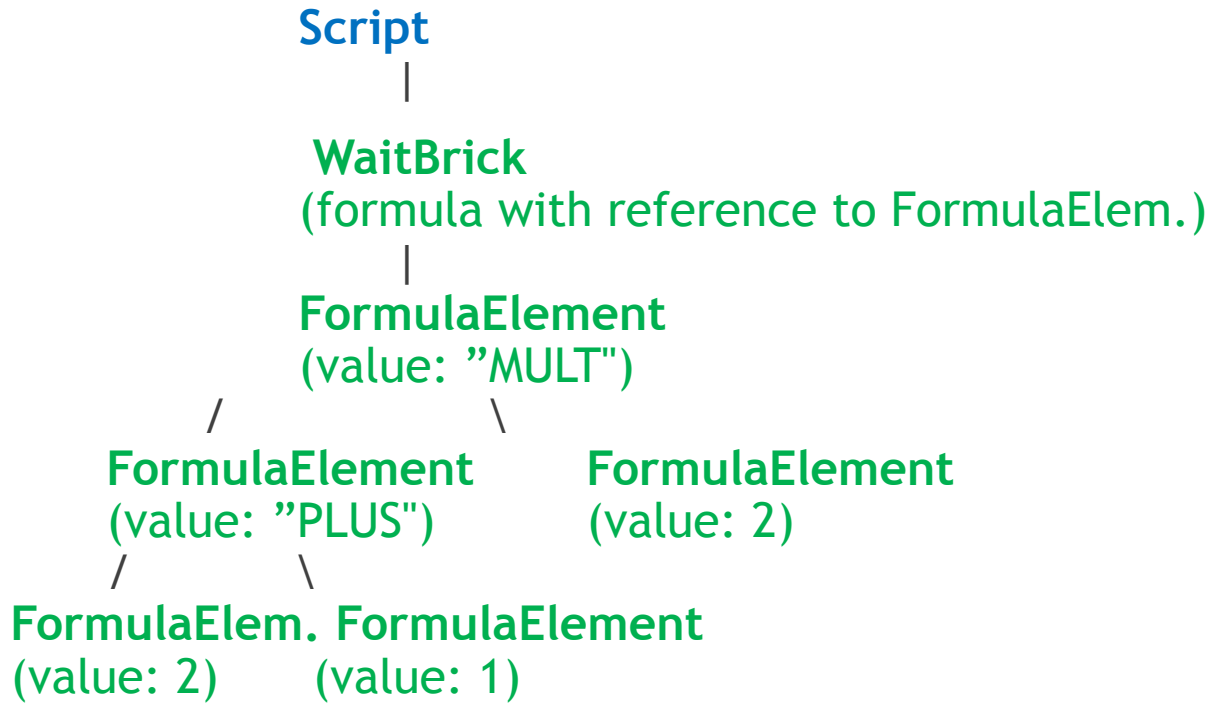


The currently visited element is highlighted blue

Already visited elements are highlighted green

Converter module (converter.py)

BlocksConversionTraverser Example



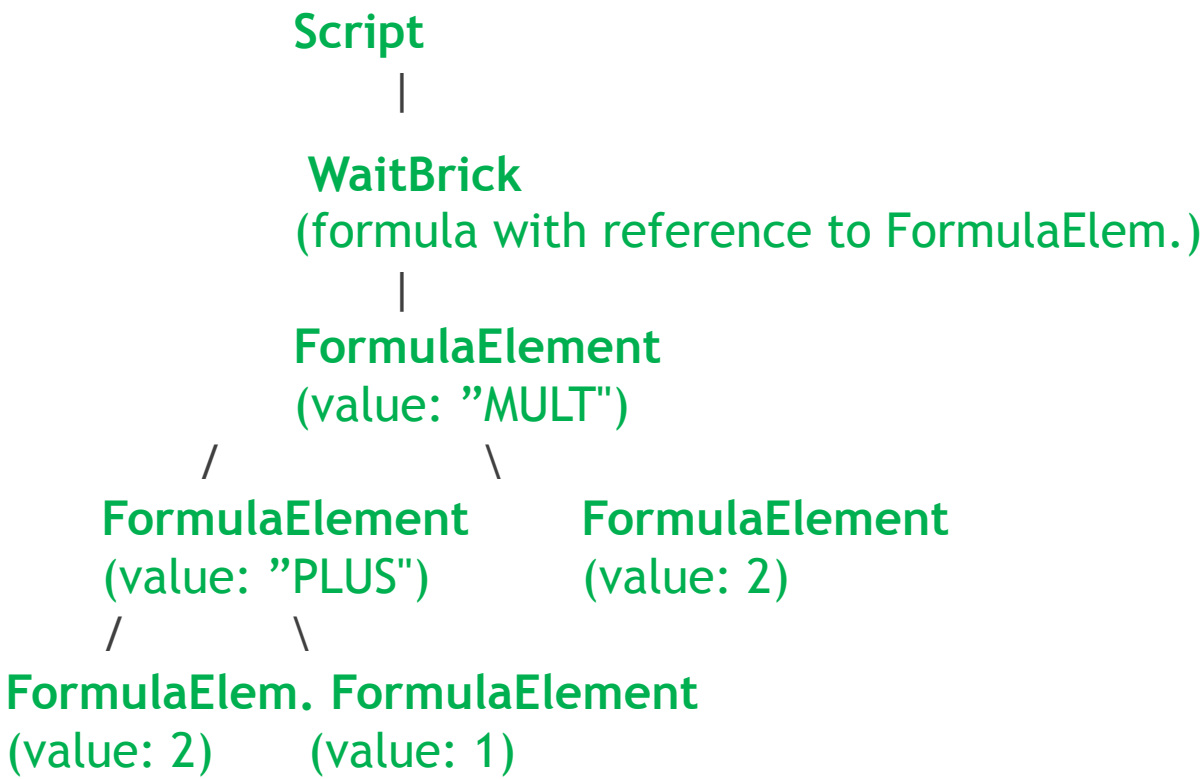
The currently visited element is highlighted blue

Already visited elements are highlighted green

Converter module (converter.py)

BlocksConversionTraverser Example

- ***Catrobat Script*** for the given JSON-code `["wait:elapsed:from:", ["*", ["+", 2, 1], 2]]`:



Converter module (converter.py)

BlocksConversionTraverser Example

- But what about the script?
  - Scripts are created outside of BlocksConversionTraverser
  - In Catrobat we have to distinguish between three script types (StartScript, BroadcastScript and WhenScript). We also already know the type of the script from the example tree, since the script element tree is part of the Scratch script coming from the scratch module and the correct script type has already been selected by the scratch module.
  - Hence, the complete JSON-code of a valid Scratch script would look like this:
    `[["whenGreenFlag"], ["wait:elapsed:from:", ["*", ["+", 2, 1], 2]]]`
  - … and the final Catrobat script for the Scratch script would be: