CODE-CHIPS: INTERACTIVE SYNTAX IN VISUAL PROGRAMMING

Emmanouil Agapakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Anthony Savidis*

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

CODE-CHIPS: INTERACTIVE SYNTAX IN VISUAL PROGRAMMING

Thesis submitted by
Emmanouil Agapakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: [illegible]
Emmanouil Agapakis

Committee approvals: [illegible]
Anthony Savidis
Professor, Thesis Supervisor

[illegible]
Polyvios Pratikakis
Associate Professor, Committee Member

[illegible]
Dimitrios Grammenos
Principal Researcher, Committee Member

Departmental Approval: [illegible]
Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, December 2021

CODE-CHIPS: INTERACTIVE SYNTAX IN VISUAL PROGRAMMING

Abstract

In this thesis, we present a novel general-purpose syntax-directed visual editor that accepts as input a programming language grammar, and offers direct-manipulation interactive visual programming features. Compared to typical syntax-directed text editors, it offers a complete block-based visual style for program elements, enabling users to form programs even in an exploratory fashion, without the need of remembering or recalling detailed program structures (learning by programming).

Particularly, the syntax-directed part of the editor allows end-users to expand non-terminal grammar symbols by selecting one of all the possible expansions in the symbol's context. At the same time, given any produced program element, the editor can display its production chain in an easily comprehensible block-based form.

Current visual programming editors offer typical jigsaw-style blocks that may be freely placed onto a canvas or connect directly to other blocks, forbidding any syntactic errors. Although such an approach enforces grammatical correctness, it fails to explicitly communicate syntactic information and therefore causes the underlying language grammar to be experientially assimilated.

With our approach, the programming language's grammar is explicit as well as an integral part of the program, enabling a learning process which is based on language exploration via editing and reviewing programs. To enhance the provided editing experience, the system supports features such as undo-redo and syntactic copy-paste, as well as aspects of modern visual programming, such as drag-and-drop insertion of pre-constructed program elements.

Finally, the system employs a row-based grid layout for spatial code organization with indentation, as well as offers the ability to view a visual program's textual form in its source language and JavaScript. In this way we increase familiarity with text-based programming and facilitate an eventual transition to typical programming environments.

CODE-CHIPS: ΔΙΑΔΡΑΣΤΙΚΟ ΣΥΝΤΑΚΤΙΚΟ ΣΤΟΝ ΟΠΤΙΚΟ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ

Περίληψη

Σε αυτήν την εργασία παρουσιάζουμε ένα νέο γενικού-σκοπού και γραμματικά-καθοδηγούμενο συντάκτη που δέχεται ως είσοδο τη γραμματική μιας γλώσσας προγραμματισμού και παρέχει σύγχρονες αλληλεπιδραστικές λειτουργίες οπτικού προγραμματισμού για τη σύνθεση του προγράμματος. Σε σύγκριση με τους παραδοσιακούς γραμματικά-καθοδηγούμενους συντάκτες κειμένου, προσφέρει μία ολοκληρωμένη οπτική μορφή βασισμένη σε γεωμετρικές δομές για τα στοιχεία του προγράμματος και δίνει τη δυνατότητα στους χρήστες να συνθέτουν προγράμματα ακόμη και μέσω διερεύνησης, χωρίς την απαίτηση να μνημονεύουν και να θυμούνται λεπτομερείς προγραμματιστικές δομές (μαθαίνοντας μέσω προγραμματισμού).

Ειδικότερα, το γραμματικά καθοδηγούμενο τμήμα του συντάκτη επιτρέπει στους χρήστες να επεκτείνουν μη-τερματικά γραμματικά σύμβολα επιλέγοντας μια από τις δυνατές επεκτάσεις (παραγωγές) σύμφωνα με το περιβάλλον του συμβόλου. Παράλληλα, δεδομένου ενός οποιουδήποτε στοιχείου του προγράμματος, ο συντάκτης μπορεί να παρουσιάσει όλη την αλυσίδα παραγωγής του, σε εύκολα κατανοητή οπτική μορφή.

Τα ήδη υπάρχοντα προγράμματα οπτικού προγραμματισμού προσφέρουν στοιχεία προγράμματος με συμβατική μορφή κομματιών παζλ που μπορούν να τοποθετηθούν ελεύθερα σε μια περιοχή-καμβά ή να συνδεθούν απευθείας με άλλα στοιχεία, απαγορεύοντας τα συντακτικά λάθη. Μολονότι μια τέτοια προσέγγιση επιβάλλει γραμματική ορθότητα, αποτυγχάνει να μεταδώσει πληροφορίες σχετικά με το συντακτικό και επομένως επιτρέπει μόνο την εμπειρική αφομοίωση της γραμματικής της γλώσσας.

Με την προσέγγιση μας, η γραμματική της γλώσσας προγραμματισμού είναι σαφής και αναπόσπαστο μέρος του προγράμματος, επιτρέποντας μια διαδικασία εκμάθησης βασισμένη στη γλωσσική εξερεύνηση μέσω της συντακτικά-οδηγούμενης επεξεργασίας και της ανασκόπησης προγραμμάτων. Για να βελτιωθεί ακόμη περισσότερο η παρεχόμενη εμπειρία σύνταξης, το σύστημα υποστηρίζει χαρακτηριστικά όπως η αναίρεση-επανάληψη και η συντακτική αντιγραφή-επικόλληση, αλλά και ευκολίες του σύγχρονου οπτικού

προγραμματισμού όπως η εισαγωγή προκατασκευασμένων στοιχείων προγράμματος μέσω εύκολης αλληλεπιδραστικής διαχείρισης.

Τέλος, το σύστημα χρησιμοποιεί μια διάταξη πλέγματος βασισμένη σε γραμμές για χωρική οργάνωση κώδικα με δυνατότητες εσοχής και προσφέρει τη δυνατότητα προβολής του οπτικού προγράμματος σε μορφή κειμένου στην γλώσσα πηγής και την JavaScript. Με αυτόν τον τρόπο αυξάνουμε την εξοικείωση με τον προγραμματισμό που βασίζεται σε κείμενο και διευκολύνουμε μια ενδεχόμενη μετάβαση σε κλασικά προγραμματιστικά περιβάλλοντα.

Acknowledgements

I would like to express my gratitude to my supervisor, Anthony Savidis, for his valuable guidance throughout this project as well as his helpful advice throughout my studies.

I also express my thankfulness to my colleagues at the PLATO laboratory for our excellent cooperation over the years.

I am very thankful for my friends, who supported me in this journey and provided a, sometimes much needed, breathing space.

Last but not least, I am sincerely grateful for my family's unconditional love and support, without which, my studies and therefore this project would not have been possible.

Στην οικογένειά μου

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we present Code-Chips, a tool for generating fully-functional powerful visual programming editors given programming language grammar specifications. Our work is inspired by the early work on the discipline of syntax-directed editing and the currently prevalent block-based visual programming editors, used in various modern teaching and learning programming applications. This chapter provides background knowledge on syntax-directed editing, visual programming, and learning programming, as well as analyses our motivations and contributions.

1.1 Background

This section provides useful background knowledge in subjects that are directly related to the work presented in this thesis and aims to increase the reader's familiarity with topics that are discussed throughout it.

1.1.1 Syntax-Directed Editing

Programming languages, due to their highly structured syntactic forms, should not be treated as mere text, composed of strings of characters. Editors that acknowledge and use this structure to maintain syntactically correct programs during editing are called syntax-directed editors [1].

Traditionally, syntax-directed editors provide usable formed language constructs, called templates. Templates may contain placeholders, which the user can replace by inserting other templates or code. Figure 1 depicts an "if-then-else" template, which contains a condition placeholder and two statement placeholders (one for the "if-part" and one for the "else-part"). In order to manipulate programs, the user moves the cursor and types commands that result to editing operations based on cursor placement. Other than insertion, syntax-directed editors can support a variety of program manipulation operations such as deletion and copy-paste, all based on the user's cursor position. Each editing operation is performed by the editor only when the resulting program maintains syntactical correctness.

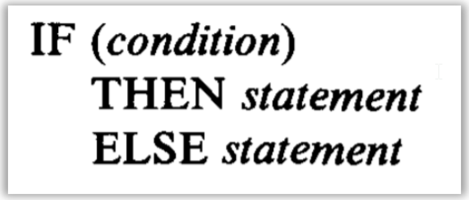

IF (*condition*)
THEN *statement*
ELSE *statement*

Figure 1*: An "if-then-else" template*

For a better understanding of program editing using a syntax-directed editor let's consider how the user would alter a complete "if-then" statement, including a condition and inner statements, to a "while" statement with the same condition and inner statements. Firstly, the user would insert a new "while" template using an appropriate command. The user would then copy the condition from the "if-then" statement and paste it into the condition placeholder of the "while" statement. Following, the user would do the same for the inner statements. Finally the user would delete the "if-then" statement to complete the task.

The above example is commonly used in related research work as it reveals a simple scenario in which syntax-directed editing complicates program manipulation compared to traditional text-editing. In this case, enforcing syntax correctness restricts the user from simply deleting "if", one character at a time, and inserting "while", one character at a time. Despite that, the main advantage of a syntax-directed editor remains: editing does not require knowledge of the language's syntax and it is impossible to make syntax errors. Due to this, syntax-directed editors have been successfully used as teaching and learning mediums to help novices learn programming along with learning a new language and its structure [2] [3] [4]. The "Related Work" section contains additional information on syntax-directed editing.

1.1.2 Visual Programming

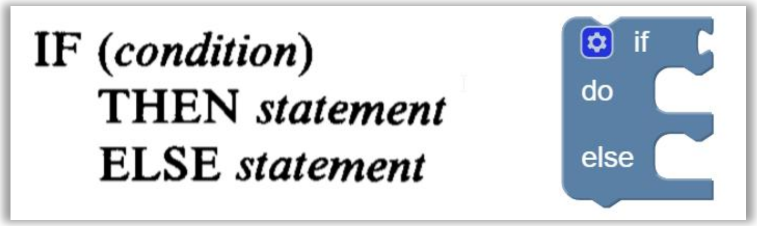


Figure 2*: An "if-else" statement as in The Cornell Program Synthesizer (left) and the corresponding Blockly block (right)*

Visual programming refers to any system that allows the user to specify a program in a multi-dimensional fashion [5]. Traditional text-based languages are considered to define programs in one-dimension since they are processed as one-dimensional streams of characters and thus do not qualify as visual programming languages. Following, we briefly describe two popular and widely used forms of visual programming: block-based visual programming and flow-based visual programming.

Block-based visual programming has become predominant in recent years, mainly in the field of teaching and learning, through visual programming environments, such as Scratch [6] and App Inventor [7], that empower children and teenagers to develop apps, games, animations, stories and more. Block-based visual program editors allow users to drag-and-drop shaped and colored structures, called blocks, and connect them with each other to form programs. Blocks connect to other blocks only to form syntactically correct programs, resembling syntax-directed editing. Figure 2 compares a traditional syntax-directed editor "if-then-else" template to a corresponding block. Due to the blocks' design which indicates and facilitates syntactically correct placement, as well as the simplicity of the drag-and-drop gesture, block-based editing appeals to the younger audience and allows for a pleasant introduction to programming. The "Related Work" section further discusses block-based visual editing and popular block-based visual programming environments.

Flow-based visual programming uses the concept of the data-flow computational model. The program can be represented by a directed graph: nodes represent functions and edges represent the flow of data (incoming edges represent input data and outgoing edges represent output data) [8]. Flow-based visual programming has various application domains such as general-purpose programming [9], game development [10] [11] and music [12]. Particularly, in the context of game development, flow-based visual programming has been incorporated by the most popular industry game engines in order to facilitate beginners and non-programmers.

Most flow-based visual programming editors use boxes for representing functions and arrow-lines to represent the flow of data. Boxes have input points, to which arrow-lines can connect, as well as output points, from which arrow-lines can originate. Input and output points can be accompanied by descriptive labels and icons for the user's convenience. The type of data flowing over arrow-lines may be indicated by the editor and used to perform type-checking, preventing connections between functions and non-matching data types. Figure 3 shows Unity's flow-based visual scripting editor.

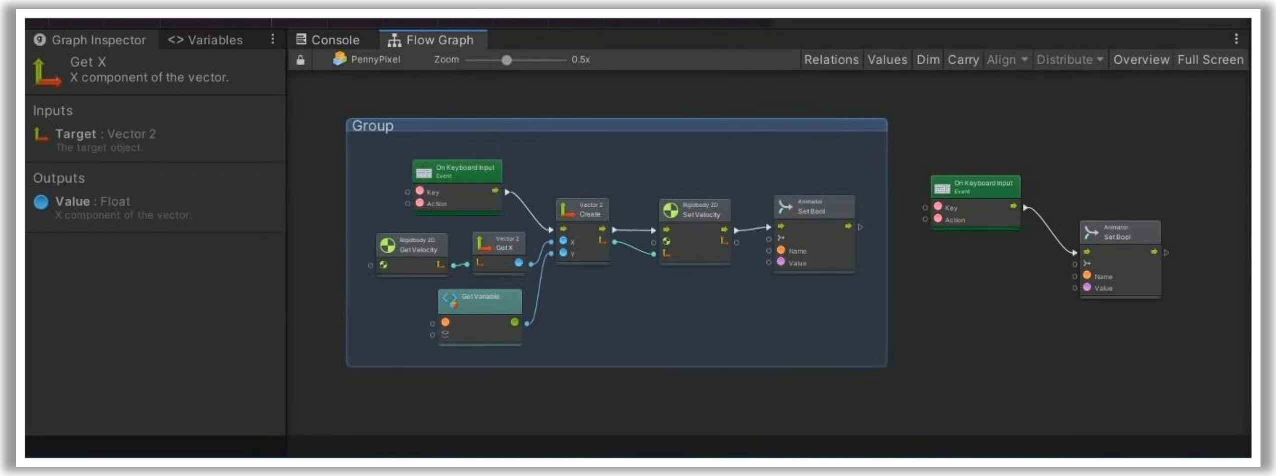


Figure 3: Flow-based visual scripting in Unity

1.1.3 Learning Programming

When considering programming for its educational benefits, one should not perceive it as a mere means of precisely defining computer instructions. Programming is a demanding process that requires algorithmic and computational thinking. During the learning process, students improve their creative thinking and reasoning skills, but also acquire and develop valuable problem-solving and cognitive skills including decomposing, abstracting, iterating, and generalizing. These skills can successfully transfer to everyday life and other application domains including physics, biology, arts and social sciences [13] [14] [15].

As a result, educational systems and schools worldwide have included programming as a core course with national curriculum, as a tool related to information and communications technology (ICT), or as a tool merely related to ICT [15]. Younger children in elementary and middle schools are introduced to computer science through applications in games, music, robotics and HTML, while high school students may be taught object orientation, data-structures, functional programming and core principles in algorithmic thinking.

Inevitably, a plethora of educational tools has been developed in order to satisfy the needs of teaching and learning programming and benefit from its aforementioned cognitive advantages. These tools are not necessarily designed for solely teaching and learning in schools or generally learning with the aid of a teacher; learning without the guidance and supervision of a teacher is often supported by using intuitive and beginner-friendly interfaces and by providing tutorials. Educational tools include visual programming games [16] [17], visual programming environments [6] [18] [19] for developing games and apps as well as games and game-based environments requiring textual programming [20] [21].

1.2 Motivation: Language Understanding

Thorough knowledge of a language's grammar and syntactic structure, whether it is a natural language or a programming language, leads to effective, expressive linguistic productions and overall a better language understanding. Particularly, according to N. Chomsky, the study of a language expands the individual's universal language comprehension, and hence facilitates subsequent learning processes for different languages [22].

Despite the benefits of learning the syntax of a programming language, novices face significant difficulties with it when forming programs in traditional text-based programming languages [23] [24]. As previously discussed, programming is an overall demanding process, requiring advanced cognitive skills. The overhead of learning the syntactic details of a programming language, on top of newly introduced concepts and the challenging nature of programming, discourages students and increases the likelihood of quitting. Modern educational tools and learning programming environments should recognize and address this issue.

Current visual programming editors only allow editing operations that result in syntactically correct programs, surpassing the difficulties of having to deal with syntax errors. Particularly, in block-based visual programming, a match in the shape of a block and the shape of a placeholder indicates syntactic compatibility. By interacting with blocks, the user might slowly form a better understanding of the language, but generally, the underlying language grammar is not communicated in a well structured manner. Additionally, although this approach breaks through the syntax barrier and succeeds in teaching fundamental computer science concepts, when used on its own, it fails on smoothly transitioning the student to textual programming languages.

Facilitating this transition is not a trivial task. Blockly Games [17] initially introduces the user to visual programming through playing a series of games. To increase familiarity with textual programming, blocks only use lowercase letters and, after the completion of a level, the system displays the JavaScript equivalent of the user's visual code. In later game stages, blocks display actual JavaScript instead of paraphrased text. Finally, to complete the transition to textual programming, the block editor is replaced by a text-based JavaScript editor.

In this thesis, we combine the methods of traditional syntax-directed editors and modern block-based visual programming editors, and enhance them with additional features in order to effectively communicate language specific syntactic information and maintain the desired intuitive human-computer interaction. With this approach, we aspire to contribute to learning programming and understanding a language's underlying structure, as well as to facilitate the transition from visual programming to textual programming.

1.3 Contribution: Learning By Editing

This thesis presents Code-Chips: a modern visual programming editor able to host any programming language given its grammar, with features inspired both by syntax-directed editing and block-based visual programming. With our approach, the programming language's grammar is explicit as well as an integral part of the program and program manipulation. Novice programmers are empowered with a block-based structured editing process that not only eliminates syntax errors, much like syntax-directed editors and visual programming editors, but also provides meaningful information on the programming language's underlying syntax.

Code-Chips, given a grammar specification for a programming language, transforms terminal and non-terminal grammar symbols into the appropriate blocks. For instance, for a traditional imperative programming language, a “statement” non-terminal symbol would be visualized as a dropdown menu block with options for expansion such as “if-statement” and “assign-statement”, while an “integer” terminal symbol would be rendered as an input box block. The editor maintains and displays grouping information. For instance an “assign-statement” block is displayed as a compound block that groups an “expression” dropdown menu block, a “=” simple terminal block and a second “expression” dropdown menu block.

To compose programs, the user iteratively expands grammar productions by interacting with the generated blocks and provides simple keyboard input, such as strings, integers and identifiers. In contrast to traditional syntax-directed editors, remembering commands is not necessary for expanding grammar symbols: the user can simply click on a dropdown menu block, and then select the desired option for expansion. For instance, to produce an ”if-statement” block, the user clicks on a “statement” block and selects “if-statement” from the displayed options. Furthermore, the user has the ability to examine any phrase within a Code-Chips program, obtaining a block view of its production path, which visualizes the expansion steps that were used to construct said phrase. With the aforementioned facilities, we aspire to provide the means of discovering a programming language's structure and semantics while editing and reviewing simple programs.

Although the process of expanding non-terminals into an available right-hand-side communicates information about the language's structure, we acknowledge that it is repetitive, especially for users that already possess this knowledge. For instance, going through the expansion of “statement” to “expression” to “call” to “console” to “print”, in order to invoke an internal library function “print”, is tedious for users that already know this method's existence and production path. To facilitate such expansions, Code-Chips supports copying and pasting blocks, as well as, dragging and dropping blocks. Users may copy or drag any phrase-block they have previously produced, or is available to them through a toolbox with blocks premade by the language author. Of course, pasting and dropping preserves the syntactically correct state of the program, by allowing only insertions in non-terminal blocks that, according to the language grammar, result in valid productions.

To ease the transition to textual programming, Code-Chips provides a textual code view mode. This mode transforms blocks into pure text and shows the program in its textual form. The language author is given the ability to include any additional tokens, such as parentheses and braces, which were not necessary within block view, but are essential in this view mode. Furthermore, when the language author defines a translation to JavaScript for a given abstract syntax tree, Code-Chips hosts a view mode which displays the user's code in JavaScript, useful for increasing familiarity with textual programming. Finally, in contrast to current block-based visual programming editors that incorporate a canvas-like editing layout for editing, we provide a line-based structured layout, which resembles the one of a text-based editor. The user may freely place new-lines and tabs inside group blocks, enabling user-defined indented code patterns.

Chapter 2

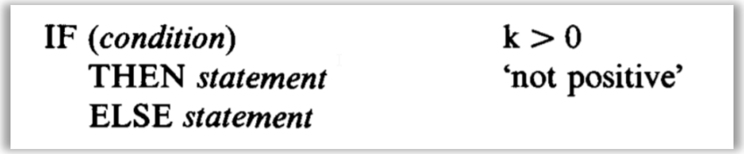
Related Work

This chapter discusses research work and systems that are relevant to the work presented in this thesis. For the purposes of this thesis, we emphasize on research with revolutionary ideas and educational systems with widespread usage, as including all available research work is not feasible. For each concept, we analyze the reasons it is thematically related to our work and specifically refer to the features each system introduces and incorporates. Particularly, for the programming editors, we focus on human-computer interaction, the visual structure and the editing features available to the end-users. Of course, for educational programming environments, we discuss their goals and achievements in teaching and learning.

2.1 Syntax Directed Editing

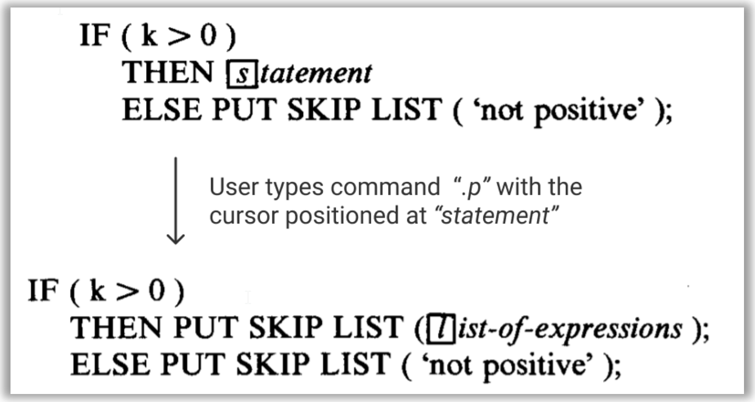
Code-Chips is a tool for generating syntax directed editors, thus it is vital to present existing work in this field. Syntax directed editing, also known as structured editing, was primarily researched in the 1980s. Although the work presented in this section is not particularly recent, there is major significance in the systems' ideas, features and intent. Programs are not text; they are hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint [3]. Syntax-directed editors did not prove to be ideal for professional development, but this core idea has influenced modern integrated development environments (IDEs) to incorporate features such as syntax highlighting and code completion. Additionally, in more recent years, syntax-directed editing has been significant in teaching and learning, inspiring a plethora of visual programming environments based on structured editing. Following, we present research work in the field of syntax directed editors.

The Cornell Program Synthesizer



```
IF (condition)                    k > 0
    THEN statement                'not positive'
    ELSE statement
```

Figure 4*: A template example (left) and a phrase example (right)*



```
  IF ( k > 0 )
      THEN [s]tatement
      ELSE PUT SKIP LIST ( 'not positive' );
```

User types command *".p"* with the cursor positioned at *"statement"*

```
IF ( k > 0 )
    THEN PUT SKIP LIST ([l]ist-of-expressions );
    ELSE PUT SKIP LIST ( 'not positive' );
```

Figure 5*: Insertion of template "PUT SKIP LIST (list-of-expressions)" at the cursor position with command ".p"*

The Cornell Program Synthesizer (1981) [3] is an interactive programming environment with integrated facilities to create, edit, execute, and debug programs. Editing is guided by the programming language's syntax and thus errors due to syntactic difficulties are avoided. When debugging the cursor's position changes, indicating the location of the instruction pointer.

The provided editor is a text and tree-view editor hybrid. It uses templates and phrases. Templates are predefined, formatted patterns of characters and punctuation marks whereas phrases are arbitrary sequences of typed symbols. Figure 4 shows examples for both templates and phrases.

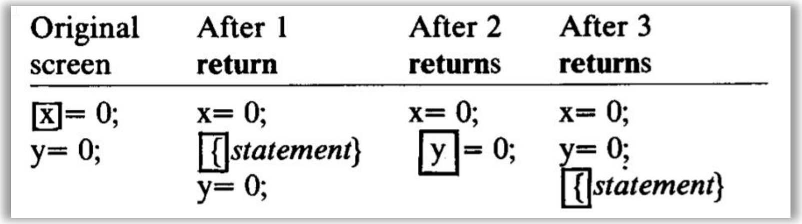
Phrases and templates can be inserted into templates replacing placeholders (such as the "condition" and "statement" placeholders of Figure 4). Insertions are ordered by quick commands that the user orders by typing. All commands are validated and executed at the current cursor position only if the program would maintain correct syntax after execution. Phrases are typed by the user one character at a time. To enforce syntactic correctness, the typed text is parsed when the cursor is directed away from the phrase and if needed an error message is displayed. Figure 5 provides an example for template insertion.

All modifications of program text occur relative to the current position of the editing cursor. The user moves the cursor using keys such as the *up, down, left, right* keys. The cursor can be positioned anywhere within a phrase and only at the leftmost symbol of a template or placeholder. Due to this, cursor movement differs from that of a traditional text editor.

The editor incorporates metasyntactic lists. Each list item is preceded and followed by "{placeholder}". The user can use these placeholders to insert templates or phrases, depending on the language's syntax. The placeholders are only displayed when the cursor is positioned there using the return key, as shown in Figure 6.

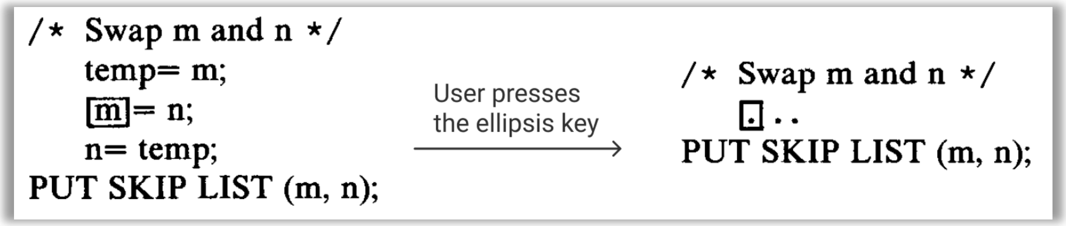
Furthermore, optional components of templates are supported and denoted as "[placeholder]". These placeholders are not normally displayed; they are displayed with a specific available command.

Finally, the editor provides comment templates as the mechanism to hide the details of a file and express computational abstractions. It provides a template with both the comment "/*comment*/" and the statements "{statement}", thus comments are not inserted in arbitrary positions of the program. With a key press, the user can hide the statements, viewing only their documentation: the comment (as depicted in Figure 7).



```
Original    After 1        After 2     After 3
screen      return         returns     returns
x= 0;       x= 0;          x= 0;       x= 0;
y= 0;       {statement}    y= 0;       y= 0;
            y= 0;                      {statement}
```

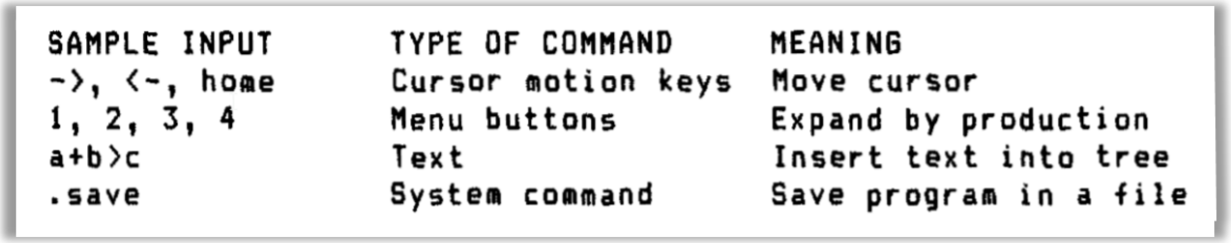
***Figure 6**: Snapshots of the editor when the user repeatedly presses "return"*



```
/* Swap m and n */
     temp= m;                        /* Swap m and n */
     m= n;                              . . .
     n= temp;                        PUT SKIP LIST (m, n);
PUT SKIP LIST (m, n);
```

***Figure 7**: Hiding the statements inside of a comment template*

SUPPORT Environment



SAMPLE INPUT	TYPE OF COMMAND	MEANING
->, <-, home	Cursor motion keys	Move cursor
1, 2, 3, 4	Menu buttons	Expand by production
a+b>c	Text	Insert text into tree
.save	System command	Save program in a file

Figure 8*: Command types for SUPPORT*

With *SUPPORT Environment [25]* (the *Still Unnamed Production Programming Oriented Research Tool Environment*), for which research began in 1981, a programmer can build a program tree within a window by expanding non-terminals into program text. The system provides an integrated program development and execution environment for Pascal, although new languages may be added since it is grammar based. Figure 8 depicts the system's available command types.

SUPPORT separates data and control, enabling the programmer to switch to a data declaration window to add declarations. This way declarations are not simply context free productions added at specific locations in the program tree. Additionally, the user can access the chain of references for any given variable and navigate into the source code.

The main contribution of this work expands on The Cornell Program Synthesizer's [3] idea of bottom-upparsing by making it available for any non-terminal grammar symbol, instead of just expressions. As a result, the user may type code and the parser will build a subtree, which will be appended to the program tree if no errors are detected in the parsing and the insertion. In the case of an error, the text is available to the user for modification and resubmission. Furthermore, the system provides the ability to clip existing subtrees and display them in a separate small editor window, through which the user can make textual modifications. After editing, the system produces a new non-terminal using the parser and appends it to the main program tree.

Finally, the system supports comments. The user may add comments by pressing a "comment button". Comments are viewed and edited through the aforementioned small editor window.

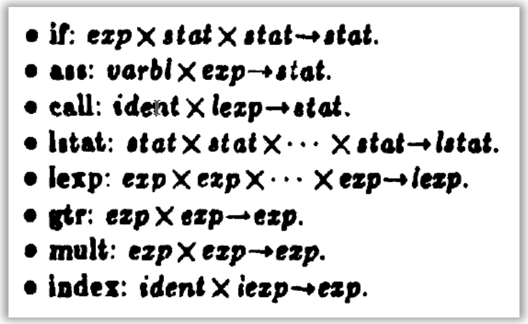
MENTOR

MENTOR [26], published in 1980, is a processor designed to manipulate structured data. This data is represented as operator-operand trees, generally called abstract syntax trees (*ASTs*). For a given language, the creator must declare a set of sorts, and a set of operators with sorted operands. Additionally, the creator must specify a parser which, given a sort, maps a concrete syntax string into the corresponding AST, and an unparser for the inverse operation. Figure 9 depicts the typical sorted operators for Pascal and Figure 10 shows a Pascal program and its parsed AST.

MENTOR is driven by the tree manipulation language *MENTOL*, through which the user issues commands to the system in order to perform tree operations. The user has access to MENTOL variables (markers) which may be assigned locations (locs) in the AST. Additionally, MENTOL supports pattern matching. A pattern is any AST containing special terminal nodes called metavariables. A pattern matches any tree which is an instance of the pattern, replacing metavariables by appropriate subtrees. When the given language is loaded, MENTOR constructs predefined patterns for each operator. Those patterns are accessible through markers named by the operators.

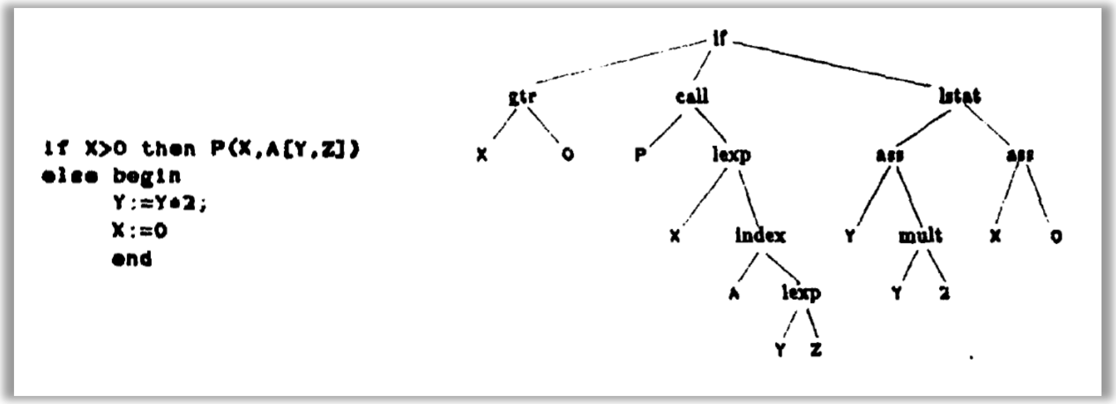
MENTOL is a full-fledged programming language, as it allows writing procedures. For instance, the predefined procedure “FORALL” takes a pattern and a command and starting from the current marker, with a preorder tree traversal, it executes the command for every instance of the pattern. Using procedures, the designer of a programming environment may provide powerful context-dependent program manipulations for the end-user, as opposed to the context-free manipulations of the MENTOL primitives.

Using the core of MENTOR, the authors developed *MENTOR-PASCAL*, a structured editor for Pascal programs. For this, the authors wrote higher-level MENTOL procedures that are the main user commands to construct and modify Pascal programs and their documentation. An example is “FPROC” which is used to move to the top of a user-given procedure. Additionally, MENTOR-PASCAL offers normalization and documentation tools for Pascal programs. Normalization includes arranging them in a more readable but semantically equivalent form and cleaning-up unnecessary structures such as empty statements. Documentation includes automatic generation of comments, scope-structures and cross-reference tables.



- **if**: *exp* × *stat* × *stat* → *stat*.
- **ass**: *varbl* × *exp* → *stat*.
- **call**: *ident* × *lexp* → *stat*.
- **lstat**: *stat* × *stat* × ··· × *stat* → *lstat*.
- **lexp**: *exp* × *exp* × ··· × *exp* → *lexp*.
- **gtr**: *exp* × *exp* → *exp*.
- **mult**: *exp* × *exp* → *exp*.
- **index**: *ident* × *iexp* → *exp*.

Figure 9*: Typical MENTOR-PASCAL operators and their sorts*



```
if X>0 then P(X,A[Y,Z])
else begin
     Y:=Y*2;
     X:=0
     end
```

Figure 10*: A Pascal program (left) and the AST it parses into (right) with MENTOR-PASCAL*

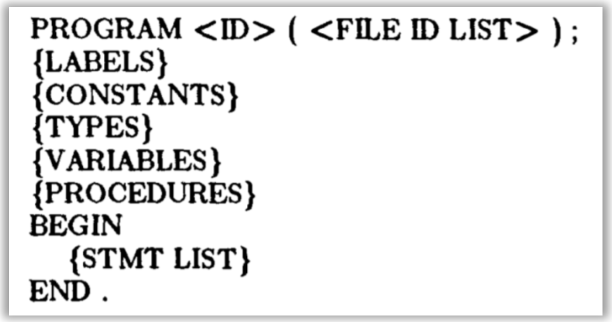
Poe

Poe [27] (*Pascal Oriented Editor*), published in 1984, is a language-based editor with knowledge of the syntactic and semantic rules of Pascal. Its user interface allows the user to move the cursor to a prompt symbol and type text. Typing a single-token prefix of a particular expansion is sufficient; an automatic syntactic error corrector will expand the user's input and make it syntactically valid.

When creating a new program *Poe* displays a program prototype (as depicted in Figure 11). In Poe, there are three types of symbols: required prompts (delimited by “<>”), optional prompts (delimited by “{ }”) and Pascal symbols (always shown in upper case). Cursor movement is controlled using the usual cursor control keys. Figure 12 shows the program of Figure 8 after the user types “if” with the cursor on “{STMT LIST}”.

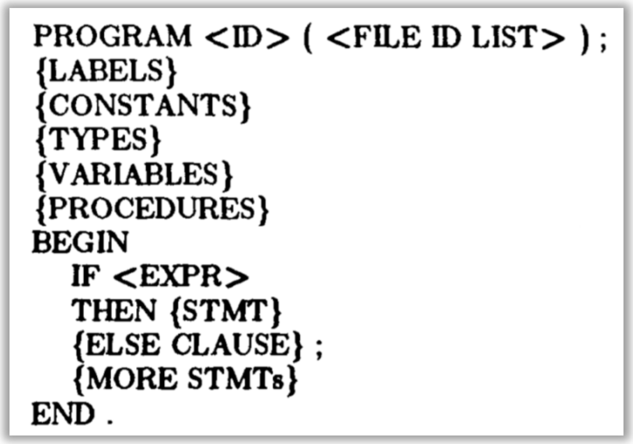
With Poe, the user can cycle through the all possible expansions of a prompt, which in fact are the productions which have the prompt as the left-hand side symbol. An elision mechanism is also provided. With the cursor on any symbol and the press of a button the user can replace the smallest structure containing the symbol with an *elision marker*. An elided "IF-THEN-ELSE" appears as "<IF-THEN-{ELSE}…>", hiding its statements.

Poe uses an algorithm to compute the locally least cost insertion sequence in order to always make user input syntactically valid. For instance when the user types "Then" with the cursor placed on a "<STMT>" the system inserts an "IF-THEN-ELSE" statement. This feature is accompanied with undo capabilities, for the cases in which the user's intent was different than the algorithm's decision. The authors also propose a solution in which a cost threshold is introduced. When the algorithm's answer has a cost above the threshold, the user will choose from a list of suggested options, as the algorithm's decision is considered to be of low confidence level.



```
PROGRAM <ID> ( <FILE ID LIST> ) ;
{LABELS}
{CONSTANTS}
{TYPES}
{VARIABLES}
{PROCEDURES}
BEGIN
   {STMT LIST}
END .
```

Figure 11: *Poe's Pascal program prototype*



```
PROGRAM <ID> ( <FILE ID LIST> ) ;
{LABELS}
{CONSTANTS}
{TYPES}
{VARIABLES}
{PROCEDURES}
BEGIN
   IF <EXPR>
   THEN {STMT}
   {ELSE CLAUSE} ;
   {MORE STMTs}
END .
```

Figure 12: *The program of Figure 8 after the user types "if" with the cursor on "{STMT LIST}"*

2.2 Block-Based Visual Programming

Block-based visual programming editors can be seen as a form of syntax-directed editors. Insertion using drag-and-drop gestures has replaced the method of keyboard typed commands, used by traditional syntax-directed editors such as those presented in the previous section. Additionally, block colors and shapes, along with shaped gaps in placeholder positions, are used to indicate a syntactic match, limiting textual information. Due to its user-friendly nature, block-based visual programming is widely used in a plethora of educational applications. Following, we present existing research in the field of block-based visual programming. For the purposes of this thesis, we prioritize discussing each block-based editor's editing features and each environment's educational applications.

Blockly

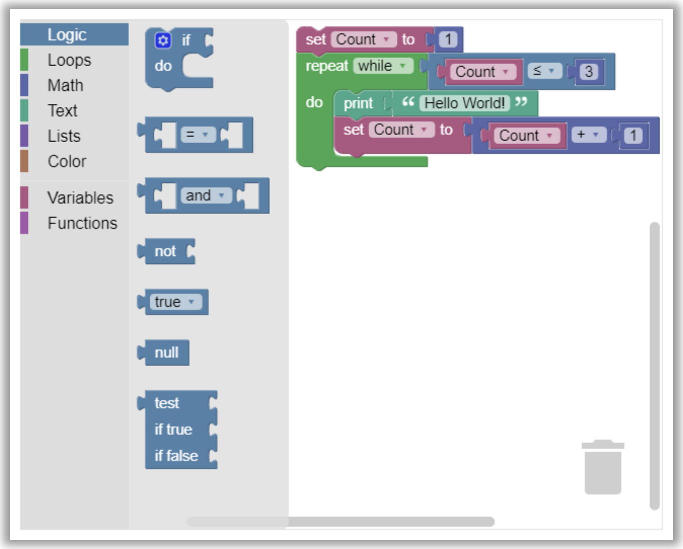


***Figure 13**: The Blockly editor*

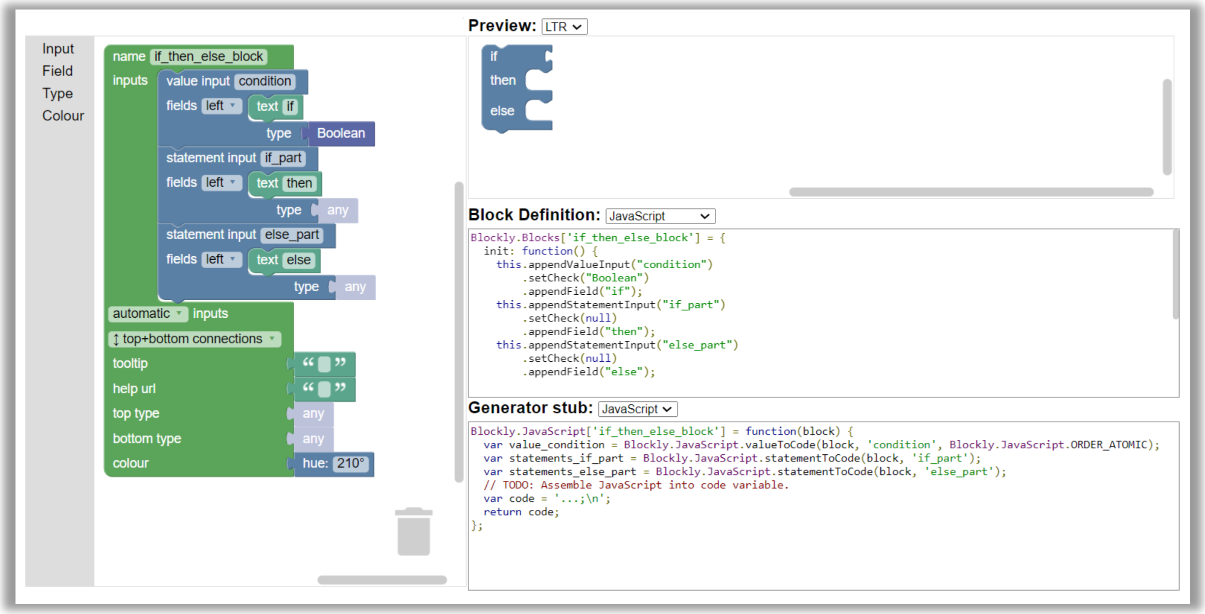
Blockly [28], first released in May 2012 by Google, is a library that allows adding visual code editors to web and mobile applications. Text is replaced by interlocking graphical blocks that enforce syntactical correctness and represent code concepts such as variables, logical expressions and loops.

The visual coding editor provided by Blockly contains a toolbox that hosts different categories of blocks. Blocks that belong in the same category are semantically related and usually share the same color. The user composes visual code by dragging blocks from the toolbox and dropping them anywhere in a canvas-like area which is positioned to the right of the toolbox and is referred to as the workspace. The user can also copy or cut and paste blocks to the workspace using keyboard shortcuts. Deleting a block is possible in four ways: by selecting it and pressing *delete*, by dropping it into a dedicated trash can positioned at the bottom-right of the workspace, by dropping it into the toolbox, and by selecting the appropriate option in the right-click context menu. Blockly supports Undo/Redo for all actions. Figure 13 depicts a simple program in the Blockly editor.

Additionally, Blockly supports collapsing blocks (used for instance to hide a function's inner blocks), as well as adding comments to blocks, both accessible through the right-click context-menu. Blocks associated with comments display a specific icon, which, when clicked, shows the comment in an overlay text area.

Blockly is not a language itself. Developers that use Blockly in higher level applications can create their own block languages based on the context of use, using the JavaScript Blockly API. Blocks can be customized in terms of their color, external inputs (connections) and internal inputs and their user-input fields. For external inputs the developer can add a left output connection point or vertical connection points. Internal user-input fields allow end-user input through a variety of interfaces such as textboxes, dropdown menus and checkboxes.

Blockly Developer Tools [29] is a web-based developer tool for facilitating Blockly's configuration process. It uses the Blockly editor and simple custom user-interfaces in order to generate ready-to-use JavaScript code. For instance, using the *Block Factory* section (depicted in Figure 14) developers can easily create new Blockly blocks for their applications.



Preview: LTR

Block Definition: JavaScript

```
Blockly.Blocks['if_then_else_block'] = {
  init: function() {
    this.appendValueInput("condition")
        .setCheck("Boolean")
        .appendField("if");
    this.appendStatementInput("if_part")
        .setCheck(null)
        .appendField("then");
    this.appendStatementInput("else_part")
        .setCheck(null)
        .appendField("else");
```

Generator stub: JavaScript

```
Blockly.JavaScript['if_then_else_block'] = function(block) {
  var value_condition = Blockly.JavaScript.valueToCode(block, 'condition', Blockly.JavaScript.ORDER_ATOMIC);
  var statements_if_part = Blockly.JavaScript.statementToCode(block, 'if_part');
  var statements_else_part = Blockly.JavaScript.statementToCode(block, 'else_part');
  // TODO: Assemble JavaScript into code variable.
  var code = '...;\n';
  return code;
};
```

Figure 14*: Block Factory in Blockly Developer Tools*

Due to its open-source nature, extensibility and high-value features such as exporting code to common programming languages, Blockly is being used by hundreds of projects, mainly active in the fields of teaching and learning. Following, we briefly describe some of these applications.

MIT App Inventor [7] is an intuitive, visual programming environment that empowers all people, and especially young people, to build fully functional smartphone and tablet apps. Using App Inventor, in school and outside of traditional educational settings, users have come together and have achieved valuable social impact to their communities.

Code.org [18] is a non-profit organization that aims to provide access to computer science in schools and particularly encourages participation by underrepresented groups such as young women. Additionally, to achieve equity and access, Code.org tries to identify and eliminate barriers that prevent the participation of students and educators with disabilities. Learning activities in Code.org include making games, apps, computer drawings with block-based programming.

Microsoft MakeCode [30] is a free, open source platform for creating engaging computer science learning experiences that support a progression path into real-world programming. Students and teachers can find material for learning and teaching programming through tutorials and courses such as "Intro to CS with the micro:bit" and "Intro to CS with

Minecraft". MakeCode includes online editors with custom application-dependent blocks and appropriate run-time simulations.

Blockly Games [17] is a series of educational games that teach programming to children with no prior-experience. It initially uses visual programming and slowly transitions to text-based programming. By the end of these games, children are ready to use conventional text-based languages.

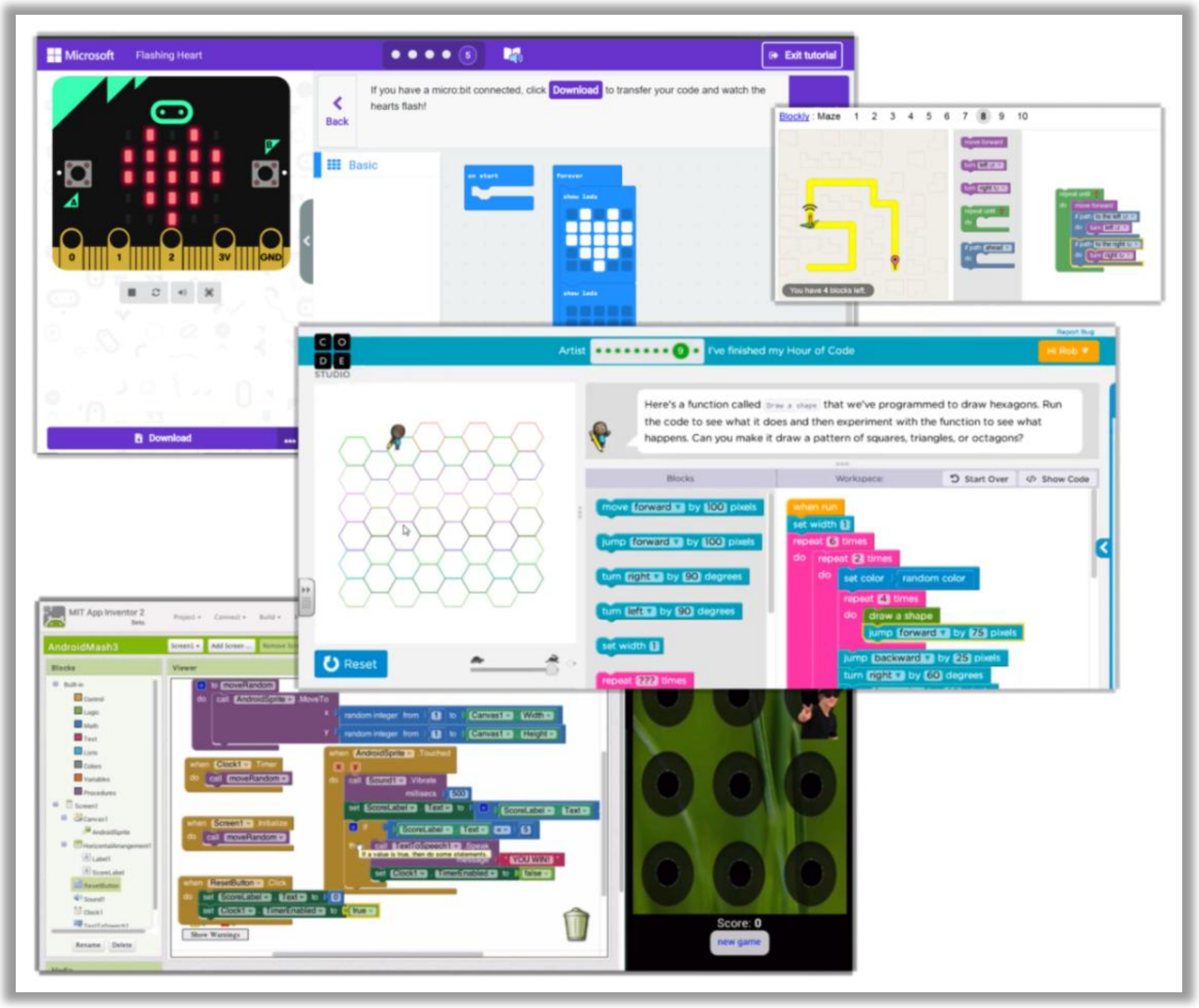
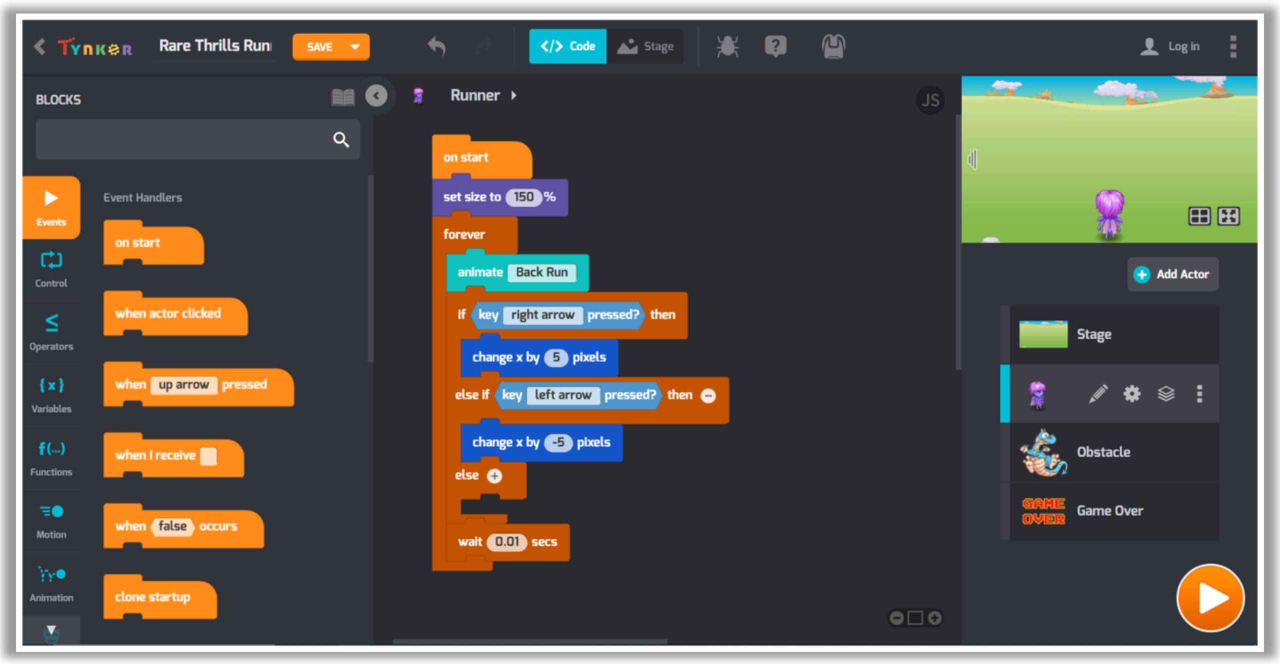


Figure 15*: MakeCode (top-left), Blockly Games (top-right), Code.org (middle) and App Inventor (bottom)*

Tynker



```
on start
set size to 150 %
forever
  animate Back Run
  if key right arrow pressed? then
    change x by 5 pixels
  else if key left arrow pressed? then
    change x by -5 pixels
  else
  wait 0.01 secs
```

Figure 16*: Tynker's visual programming environment*

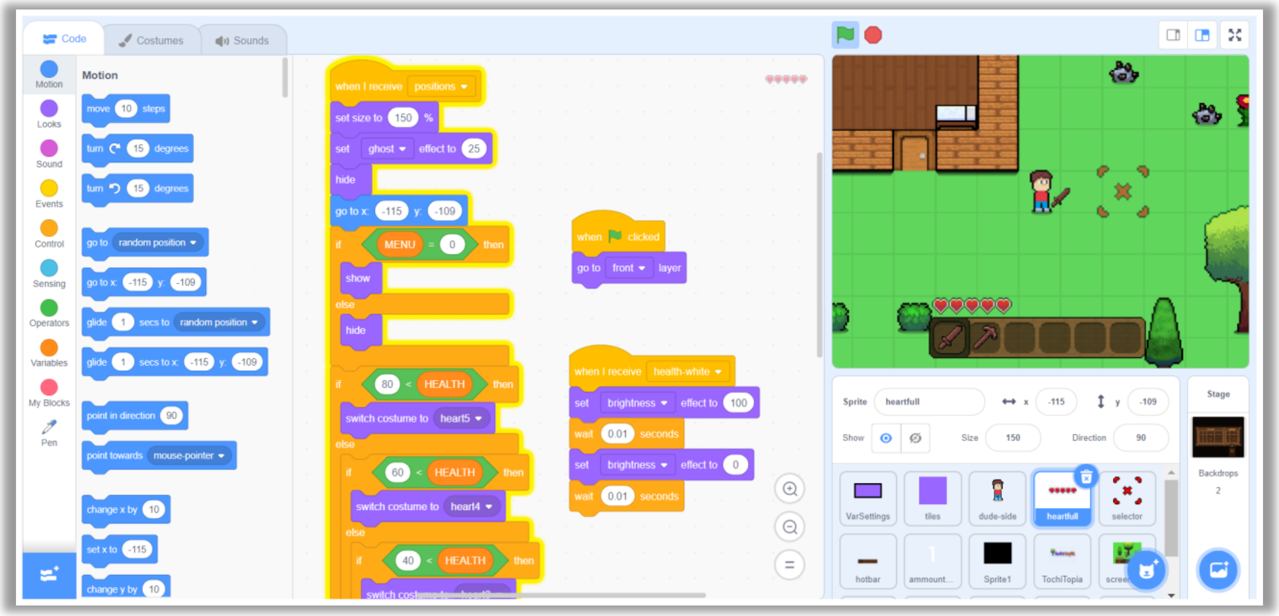
Tynker [19], founded in 2012, is a learning programming environment with immersive game-like courses and interactive lessons with built-in tutorials and has been used by over 60 million students in over 90 thousand schools. Through Tynker's intuitive visual programming language students can learn the fundamentals of programming and design without the frustrations of traditional syntax.

Lessons in Tynker are designed for children to keep advancing their skills, motivating them by offering rewards such as achievements and badges for their progression. Interactive explanations are introduced by interesting characters and short clips are provided for easy guidance on common actions such as deleting Actors or changing the background.

Tynker offers lessons suitable for children of all ages. Children under the age of 7 can be introduced to programming through a visual programming language with *iconic blocks* (i.e. blocks that rely on images instead of text). Building apps and games, controlling robots and drones, designing Minecraft mods are supported through visual programming and are recommended for children of ages 8 to 13. Children over the age of 14 can be taught more advanced computer science concepts such as data structures with popular textual programming languages such as JavaScript and Python.

The visual programming editor provided by Tynker is visually and functionally similar to Blockly. It provides a toolbox with categories of blocks, enhanced with a search bar for easier access. In the workspace blocks can be freely dropped. Blocks have shaped connections and are, themselves, shaped appropriately to annotate valid connectivity and enforce correct syntax. Blocks of the same category are given the same color to highlight semantic relevance. Additionally, the editor provides an area, called backpack, in which the user can drop sequences of blocks to save them for easy access and reuse, similar to code snippets in text-based programming. Furthermore, Tynker's editor makes the clipboard available to the user: every time a block is copied, it is inserted as the top element of a clipboard dedicated area. The user can drag blocks from this area and drop them in the workspace as they would do for blocks from the toolbox. The editor also provides an option for arranging blocks which aligns them in terms of their x coordinates and distances them appropriately. Finally, the editor is accompanied by a visual programming debugger that supports breakpoints, using a specific breakpoint block, and watchers, which are added and viewed through an appropriate user interface. Figure 16 shows Tynker's visual programming environment.

Scratch



```
when I receive positions
set size to 150 %
set ghost effect to 25
hide
go to x: -115 y: -109
if MENU = 0 then
  show
else
  hide
if 80 < HEALTH then
  switch costume to heart5
else
  if 60 < HEALTH then
    switch costume to heart4
  else
    if 40 < HEALTH then
      switch costume to heart3

when 🏴 clicked
go to front layer

when I receive health-white
set brightness effect to 100
wait 0.01 seconds
set brightness effect to 0
wait 0.01 seconds
```

***Figure 17**: A user-made game in Scratch*

Scratch [6], first released in 2007, is a free educational block-based language, developed at MIT and designed for ages 8 to 16, enabling users to program interactive stories, games, animations and more. It has over 76 million registered users and 84 million shared projects, and is used in more than 150 different countries, being available in more than 60 languages. Figure 17 depicts a user-made game in the environment of Scratch.

Scratch's visual programming is based on *Scratch Blocks* which is a collaboration between Google and MIT's Scratch Team, building on the technologies of Blockly. Again, the visual code editor is composed of a toolbox area from which the user can drag blocks and a workspace area in which the user can freely drop blocks. The blocks are colored to indicate semantic relevance and shaped to indicate the visual language's syntax. Scratch has incorporated most of Blockly's editing features, although there are functional differences and features which were not included by choice. For instance, Scratch has omitted block collapsing as well as the Blockly trash can used for deleting blocks and viewing deleted blocks, and has altered the display of comments in blocks. Additionally, Scratch's toolbox has an embedded option for adding built-in extensions such as "Music" and "micro:bit".

Each extension expands the toolbox with an additional category that contains new usable application-specific blocks.

In Scratch there are *command blocks*, *function blocks*, *trigger blocks* and *control structure blocks*. Command blocks can be chained with each other to create stacks of commands, much like statements in text-based programming. Function blocks return a value and simulate expressions such as function calls and arithmetic expressions in text-based programming. Trigger blocks have rounded top sides and thus provide only a bottom connection, allowing for a stack of command blocks that executes when the triggering event occurs. Control structure blocks have openings for nested command stacks much like an "if-then-else" statement in text-based programming. Figure 18 depicts the visual representation of the aforementioned block categories.

Scratch has *boolean*, *number* and *string* as its first-class data types which can be stored in variables and used in expressions. Function blocks for numbers and strings, as well as insertion points, have circular shaped edges while function blocks and insertion points for booleans have triangular shaped edges. This represents the fact that, in Scratch, number and string function blocks can be used interchangeably, by internally coercing the parameter to the target type if necessary. The same does not hold for boolean function blocks which may only be inserted in insertion points of boolean type.

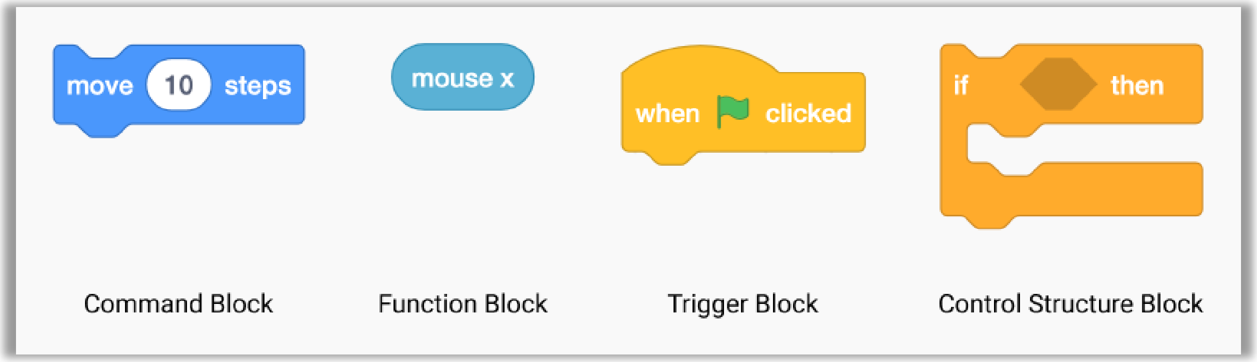
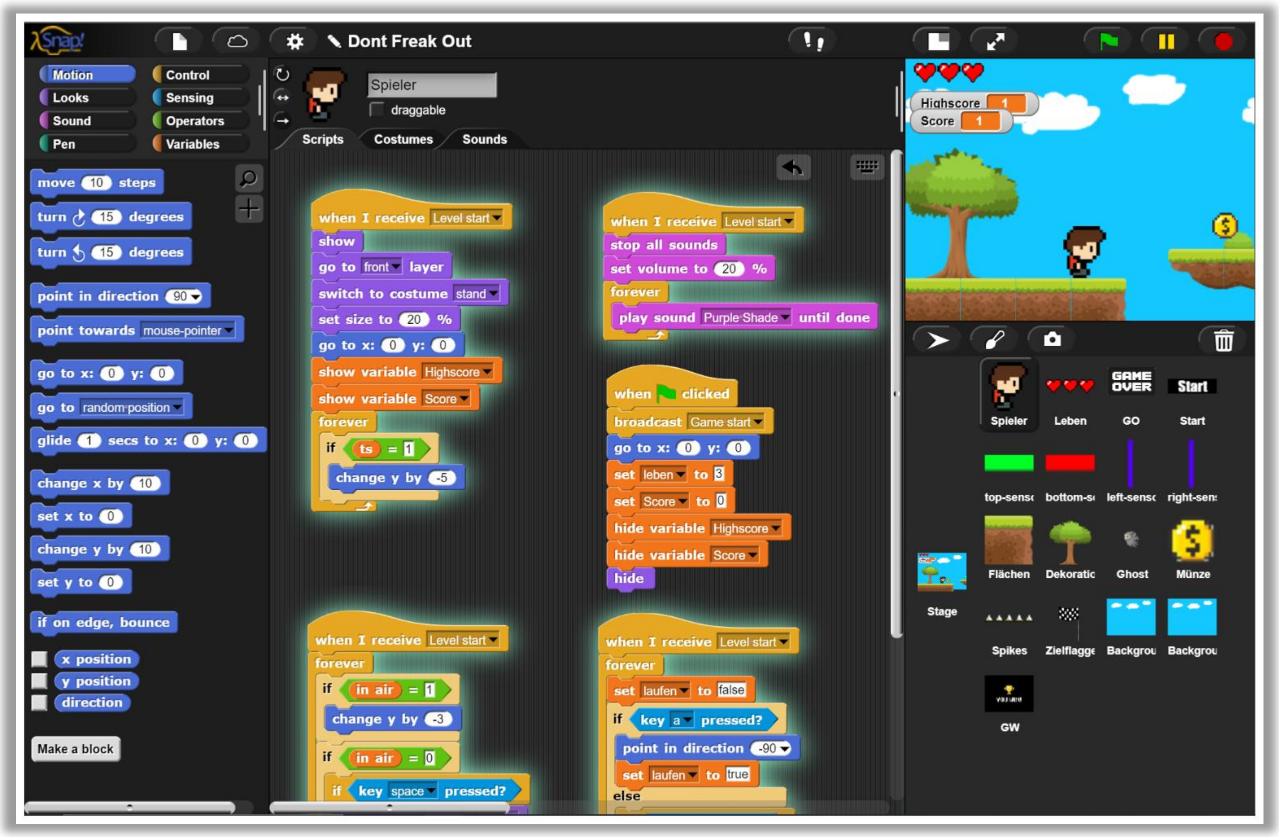


***Figure 18**: The four kinds of blocks in Scratch*

Snap! (or BYOB)



```
when I receive Level start
show
go to front layer
switch to costume stand
set size to 20 %
go to x: 0 y: 0
show variable Highscore
show variable Score
forever
    if ts = 1
        change y by -5
```

```
when I receive Level start
stop all sounds
set volume to 20 %
forever
    play sound Purple Shade until done
```

```
when clicked
broadcast Game start
go to x: 0 y: 0
set leben to 3
set Score to 0
hide variable Highscore
hide variable Score
hide
```

```
when I receive Level start
forever
    if in air = 1
        change y by -3
    if in air = 0
        if key space pressed?
```

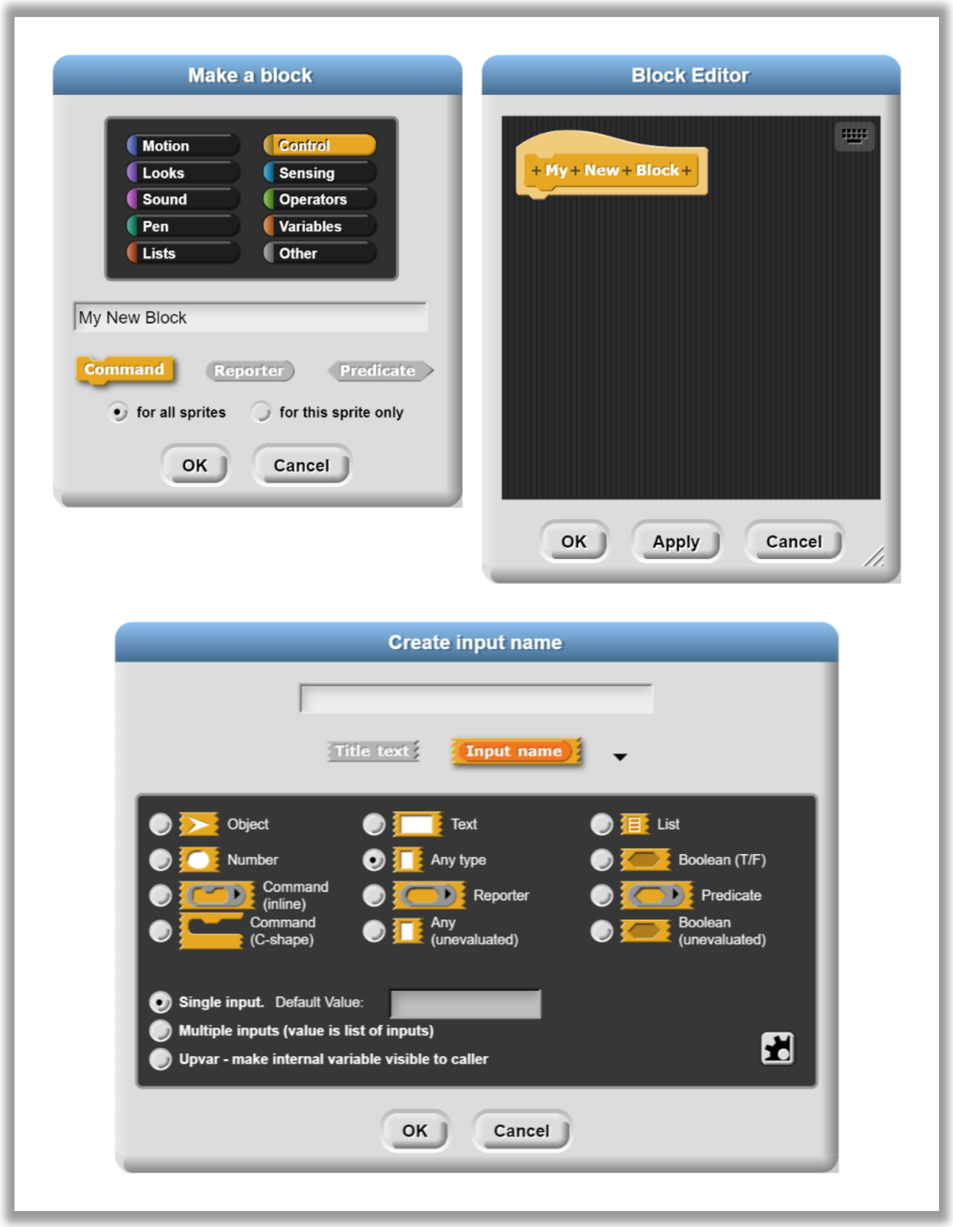
```
when I receive Level start
forever
    set laufen to false
    if key a pressed?
        point in direction -90
        set laufen to true
    else
```

***Figure 19**: A user-made game in the environment of Snap!*

Snap [31], also known as *Build Your Own Blocks* (*BYOB*), was first released in 2011 and is a block-based educational visual programming language for exploring, creating and modifying interactive animations, games, stories and more. BYOB is a reimplementation of Scratch with the extensions of first-class procedures, first-class lists and first-class sprites with inheritance. As a result of the added complexity, BYOB is targeted primarily at teenagers such as high school students, in contrast to Scratch's younger target audience.

As indicated by the name BYOB, custom block creation by end-users was a primary goal. With Snap, users are able to build a block by initially defining its toolbox category, name, type (which determines its connectivity) and scope (if it is available to all the scripts or only the current script). After that the user may add input slots and program the block's behavior using the visual programming editor. Figure 20 depicts the Snap user interface for creating a block.

Snap's visual programming editor consists of a toolbox and a workspace, similarly to the block-based editors that were previously described in this chapter. The blocks are colored and shaped similarly to Scratch and features such as Undo/Redo and adding comments to blocks are available to the end-user. In contrast to Scratch, the toolbox is resizable and a search-bar is provided for easier block access. Figure 19 shows a user-made game in the environment of Snap.



My New Block

[✓ Command]

[✓ for all sprites]

[✓ Any type]

[✓ Single input.]

***Figure 20**: User Interface for block creation in Snap!*

Snap, with its first-class procedures, lists and sprites is suitable for a serious introduction to computer science for its users. Snap visualizes procedures as grey rings that encapsulate blocks and provides a “call” block for invoking them. In this way, users may write more advanced programs such as higher order list functions that are essential in the functional programming style (Figure 21 provides an example).

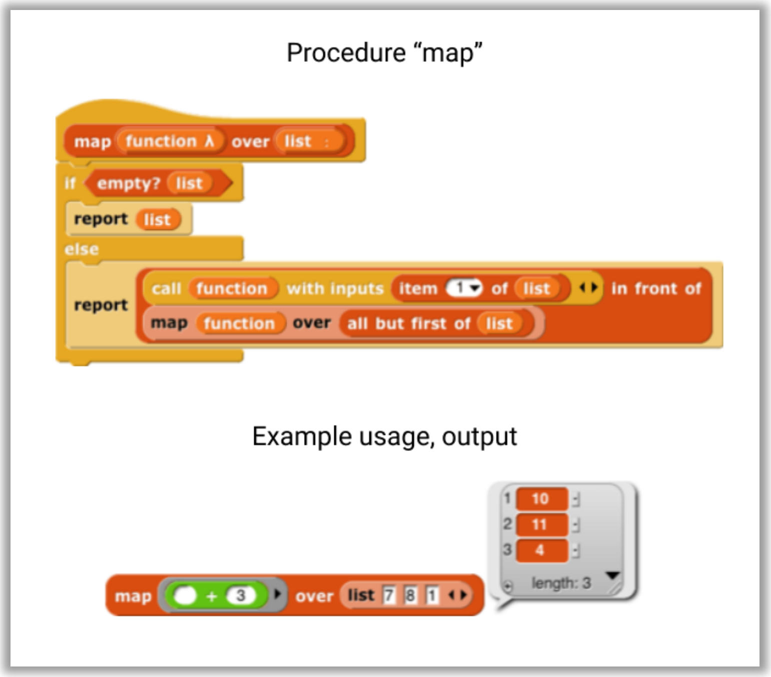

Procedure “map”
map function λ over list :
if empty? list
report list
else
report call function with inputs item 1 of list in front of map function over all but first of list
Example usage, output
map (+ 3) over list 7 8 1
1 10
2 11
3 4
length: 3

***Figure 21**: A higher order function example in Snap!*

Chapter 3

Features

Throughout this thesis, we refer to two not necessarily mutually exclusive types of users for our system: the end-user, who we commonly refer to as the user, and the language creator or language author. The end-users consist of the audience that the editing features and learning programming applications of the system are designed for, while the language author is the individual that designs a language and uses the system's infrastructure and configuration facilities in order to embed the language into the system. In this Chapter, we will analyze the features and functionality provided by Code-Chips, as well as describe the potential applications in learning programming through language exploration. A subset of the presented features requires preparation from the language author, usually in the form of filling out simple configuration files. When analyzing such features, we mention the set-up process required by the language author.

3.1 Input: Language Grammar

Code-Chips is a syntax-directed block-based visual programming editor, and therefore is suitable for a range of applications. It can be used by beginners and students for an introduction to programming as well as by experienced programmers to learn the syntax and semantics for more advanced programming language concepts, such as Java Classes or C++ templates, or explore the syntactic capabilities of a new language. Additionally, we aim for a general purpose visual programming editor and thus consider integration in applications such as game development or simple robot programming. For the above reasons Code-Chips is designed with the ability to host different programming languages.

In order to load a language into Code-Chips, the language author provides a specification of its grammar. For a successful parsing by the system, the input language grammar should be given in a well-defined form. In this context, we have authored a meta language through which language authors can define visual programming languages. For an interested reader, the Code-Chips meta language grammar specification is discussed in 4.2. Additionally, a fully operational general-purpose example language and its grammar specification are analyzed in 5.1. With a correct input language, the system automatically generates the corresponding set of interactive blocks, as well as a ready-to-use syntax-directed editor for composing programs.

3.2 Editing: Block-Based Syntax-Directed Code Manipulation

This section discusses the features and characteristics of the editing process according to our approach. We first mention and analyze the types of interactive blocks available for the user and how they are used to form programs. We then focus on how beginners and more experienced users can use this process to explore the structure of the used visual programming language and how they may benefit from such a process. Lastly, we discuss editing features inspired from block-based visual programming or textual programming and how they facilitate and enhance the editing process for a complete programming experience in our system.

3.2.1 Syntax-Directed Editing with Blocks

Before discussing the syntax-directed editing process itself, it is vital to describe all the different categories of interactive blocks that can be created through the block generation process, when the editor is provided with an input programming language grammar. These block categories are currently *selection blocks*, *simple blocks*, *input blocks*, *group blocks*, *repetition group blocks* and *optional blocks*.

Selection blocks are dropdown menus with selectable options. When clicked by the user the dropdown menu expands and presents the options that are available. When an option is selected by the user, the system replaces the selection block with a new block corresponding to the selected option. For instance, in a general purpose imperative programming language, a “statement” selection block may have different options such as “if-statement”, “while-statement” and “assign-stmt”. If the “while-statement” option is selected, the “statement” selection block is removed and an appropriate block is generated for “while-statement” - that is “statement” is expanded into “while-statement”. This example is depicted in Figure 22. In any case, each available option can be accompanied by a descriptive tooltip with appropriate information about the symbol’s semantics and usage.

Simple blocks are the least interactive of all the aforementioned categories. They are blocks that simply display text and can represent static terminal grammar symbols such as operators and keywords. For instance, the “if” keyword of an “if-statement” and the “+” operator of an arithmetic expression are all represented by simple blocks.

Input blocks are blocks that contain a text field in which the user can type using the keyboard. In Code-Chips, user input provided in such blocks can be checked for validity. The system currently supports checking for identifiers, strings, integers, floating numbers and single characters but is extensible for custom predicates. For instance, in a traditional programming language a user can type an integer into an input block that expects such an input. If the input provided by the user is not valid, the input block visually indicates that.

Code-Chips groups blocks into compound blocks to convey language specific syntactic information. These compound blocks are called group blocks and they render as containers with one or more inner blocks. For instance, a traditional “assignment” block could be a group block consisting of three inner blocks: an “identifier” input block, a “=” simple block and an “expression” selection block. While we have not yet discussed the available editing operations, it is vital to realize the importance of grouping for convenient editing and enforcing syntactically correct programs. Selecting per say an “assignment” option from a “statement” selection block would result in generating one compound “assignment” block with three inner blocks for the left-operand, the “=” operator and the right-operand. Managing this “assignment” block now becomes easier, and actions such as deletion can operate on the compound “assignment” block. Deleting just the “=” simple block would result into a syntax error and therefore is not allowed. Another important point that comes from using compound blocks is that the language author can omit tokens used to indicate grouping, such as parentheses or braces in traditional programming languages.

A repetition group block represents a metasyntactic list and allows repeating a specific block zero or more times. Repetition group blocks are block containers, like group blocks, but in contrast to them, they allow addition of elements while editing. The main way this is possible is by clicking an appropriate button which is part of the repetition group block’s rendered view. Each time the user clicks the aforementioned button, the system repeats another instance of the repetition group block's repetitive element. For instance, we can model the "statements" part of a traditional "if-statement" as a repetition group block with a "statement" selection block as its repetitive element. Each time the user clicks on the repetition group's "+" button, an additional "statement" selection block is appended to the "statements" repetition group block. Figure 23 depicts a repetition group block for the "statements" of a traditional "if-statement" block.

Optional blocks are elements that represent optional tokens in programs. For example, a traditional Java class declaration can have optional access and linkage modifiers, template type parameters, extend another class and implement interfaces. All the aforementioned characteristics of a class are optional; that is the programmer may omit them completely if they are not needed in their program. For this purpose, Code-Chips provides optional blocks which, when clicked, generate additional sequences of blocks, in order to save editing time and screen space. For instance, an "if-else-statement" that has its "else" part as an optional block can initially have no "else" part, but when this is needed, the "else" part can be easily added on the fly. With the absence of optional blocks, "if-statement" and "if-else-statement" could be separate blocks, or the "if-statement" block could be omitted and simulated by an "if-else-statement" block with its "else" part having no statements. With the first approach, more editing steps are needed to change an "if-statement" to an "if-else-statement", while the second approach always uses more screen space. Figure 24 depicts a Java class declaration block with optional blocks.

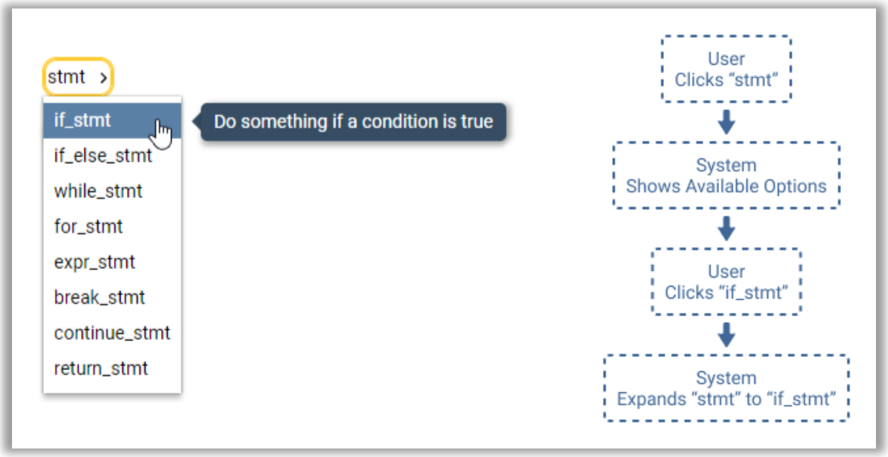


***Figure 22**: The functionality of a selection block*

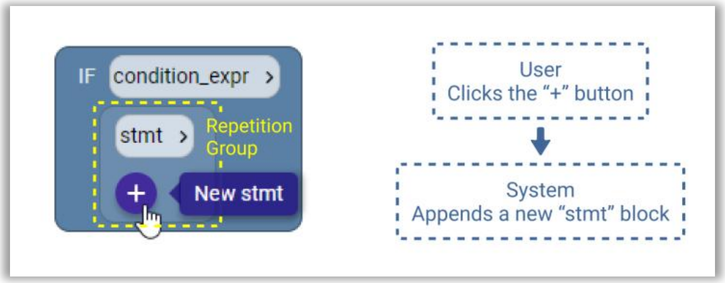


***Figure 23**: The functionality of a repetition group block*

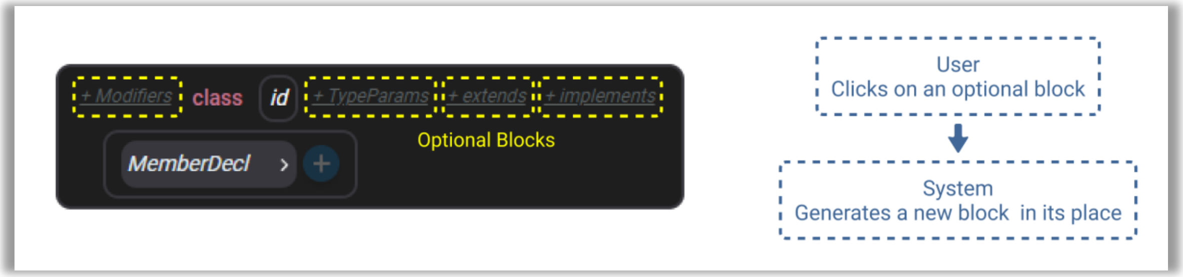
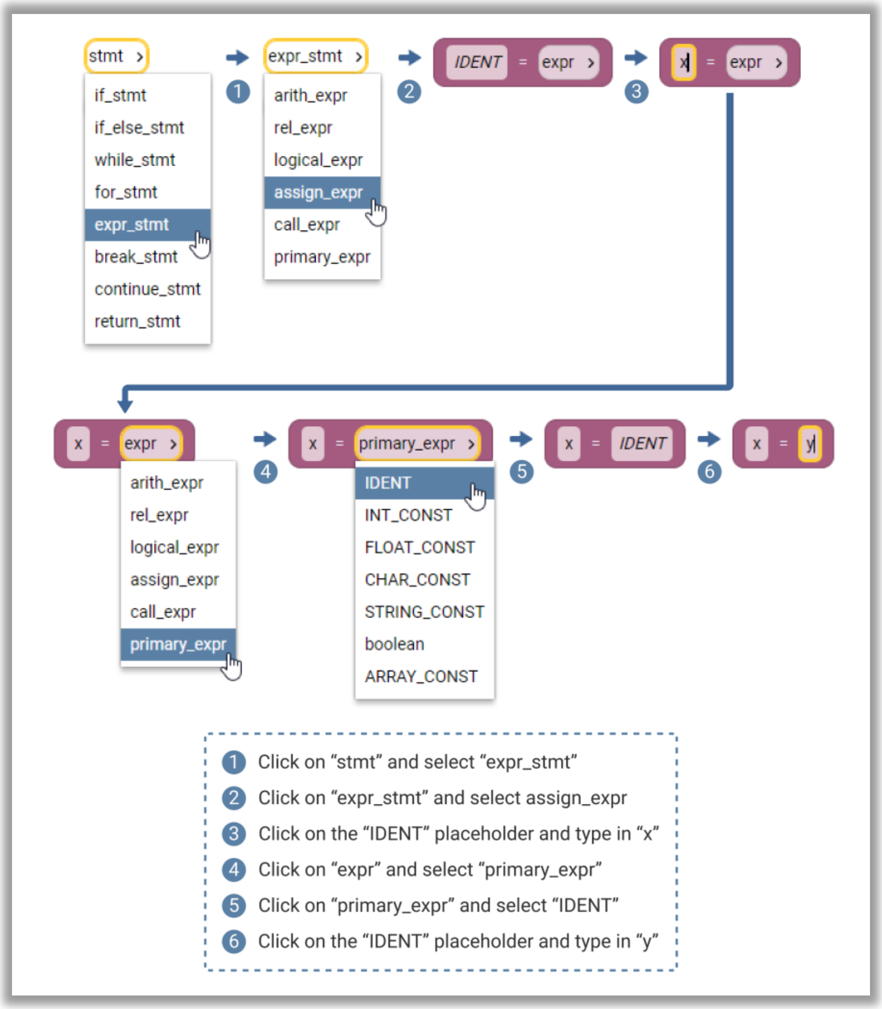


***Figure 24**: The functionality of an optional block*

After discussing each currently supported block type, we can make a deeper dive at syntax-directed editing through Code-Chips blocks. As previously mentioned, traditional syntax-directed editors use unique keyboard-typed commands for inserting code-templates at the cursor's position. However, this approach requires remembering commands and being a priori aware of possible insertions that maintain syntactic correctness at the cursor position. With our approach, the aforementioned block categories replace code-templates and insertion is explicitly handled by selection blocks. The user can click on a selection block to preview all the different blocks that are allowed to be inserted at this position. The user, thus, is not required to remember commands and can instead quickly interact with selection blocks to be reminded of all the possible insertion options, along with a descriptive tooltip for each option. Figure 25 depicts a sequence of syntax-directed insertions with Code-Chips blocks that in the end produces a simple "x=y" assignment.

Selection blocks represent language grammar non-terminals, or in simpler terms, intermediate placeholders that need to be expanded until another block type is reached in order to produce meaningful programs. For instance, a "statement" block does not have any semantics by itself – it is identical to the empty statement. It is when an expansion produces, per say, an "if-statement" group block that a meaningful program starts to form. In order to have no unpredictable program behavior, the user should have no selection blocks left in their program – that is every non-terminal grammar symbol is eliminated through consecutive expansions.



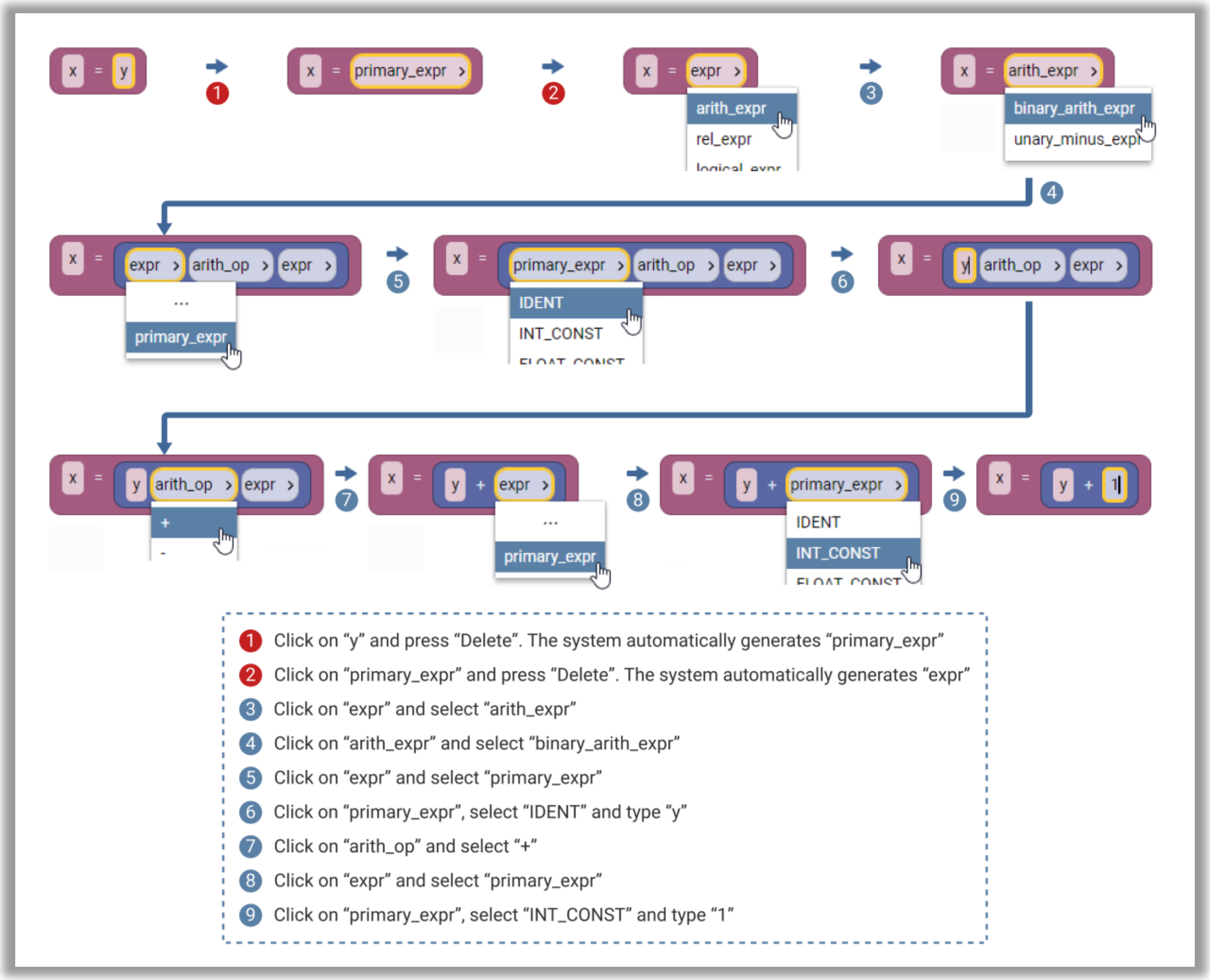
1 Click on "stmt" and select "expr_stmt"
2 Click on "expr_stmt" and select assign_expr
3 Click on the "IDENT" placeholder and type in "x"
4 Click on "expr" and select "primary_expr"
5 Click on "primary_expr" and select "IDENT"
6 Click on the "IDENT" placeholder and type in "y"

***Figure 25**: The steps of creating a simple "x=y" assignment*

Now that block insertions through selection block expansions are more apparent to the reader, let us discuss deletion. As in traditional syntax-directed editing, our approach does not allow deletion by the process of deleting one character at a time as this would result in programs that are, at least temporarily, syntactically ill-formed. Particularly it is not always safe to allow deletion, even at block level. For instance deleting the "if" keyword of an "if-statement" block results in dangling condition and statement blocks. In Code-Chips, deleting a block is only possible if the block was generated by a selection block, an optional block or a repetition group block. In the cases that the block was generated by a selection block or an optional block, a deletion operation would result in replacing the block with the block that generated it so that the program remains syntactically correct and further editing operations are allowed. For instance, imagine an “x > y” phrase represented by a group block, generated by expanding an “expression” selection block. When that “x > y” block is deleted, Code-Chips automatically generates an “expression” selection block in its place. To better understand the implications of this process, Figure 26 depicts and describes the steps of altering the “x = y” assignment of Figure 25 to a different assignment of “x = y + 1”.

Most of the editing actions supported by Code-Chips operate on a single block, selected by the user through clicking. Performing such editing operations is possible either through shortcuts and keyboard input, or through the right-click context-menu. For instance, in order to delete a block, the user can either right-click it and choose the delete option or click it, in order to select it, and press the delete keyboard button.

Although the mechanism of expanding selection blocks is a primary syntax-directed editing operation and provides enough for users to compose programs, consecutive block expansions may become repetitive and tedious for a user that already has formed an understanding of the visual programming language they are using. The example of Figure 26 demonstrates how an “x = y” phrase can be altered to “x = y + 1” in 9 compact steps. Throughout this section, we introduce editing features that can reduce user effort and generally facilitate the editing process in different ways.



1. Click on "y" and press "Delete". The system automatically generates "primary_expr"
2. Click on "primary_expr" and press "Delete". The system automatically generates "expr"
3. Click on "expr" and select "arith_expr"
4. Click on "arith_expr" and select "binary_arith_expr"
5. Click on "expr" and select "primary_expr"
6. Click on "primary_expr", select "IDENT" and type "y"
7. Click on "arith_op" and select "+"
8. Click on "expr" and select "primary_expr"
9. Click on "primary_expr", select "INT_CONST" and type "1"

Figure 26*: The steps for altering "x = y" to "x = y + 1", only using deletion and insertion by expansion*

3.2.2 Syntactic Copy-Paste

Copying, cutting and pasting are well established features for text editors and visual programming editors. Specifically, for a syntax-directed editor, copy-and-paste can save time and effort used to construct an already existing phrase. In Code-Chips, copy-and-paste can replace interacting with selection blocks and input blocks and can variably reduce user effort, depending on the case. For instance, the fifth and sixth steps of Figure 26 consist of clicking on selection blocks to display the available options for expansion, finding and choosing the wanted option and finally typing the desired identifier name "y", in order to reconstruct the already existing input block "y". These steps can be replaced by a copy operation on the input block “y”, before the user proceeds with step 1 and a paste operation targeted at the “expression” before step 5.

Copying in the syntax-directed editing environment of Code-Chips is a self-explanatory operation – the user can save an exact replica of a block in a clipboard, for later use, through pasting operations. Of course, the block, targeted by a copy operation, is not required to exist when the pasting operation is initiated; the target block for the copy operation could have been deleted in the meantime.

Pasting, on the other hand, is not trivial due to maintaining syntactic compliance relative to the language grammar. Particularly the pasting mechanism should produce no syntax errors in a non-restrictive manner for the user. The provided implementation checks whether the copied block can be pasted at the selected destination block by comparing the grammar symbols that correspond to these blocks. This comparison is not as simple as an equality check between grammar symbols; this approach would be extremely restricting as it would not allow pasting per say an identifier onto an expression, although “expression” can expand to “identifier”. Additionally, an approach that checks whether the symbol of the copied block can be produced by consecutively expanding the symbol of the destination block is not sufficient. Such an approach would not per say allow pasting a “y + 1” group block to replace a “y” identifier block, as the “y” identifier block cannot expand to “y + 1”.

The solution currently employed by Code-Chips, allows syntactic copy-and-paste when the root symbol of the destination block can expand to the symbol of the copied block. According to this, the aforementioned example of pasting “y + 1” onto “y” would work, as in this case “y” has a root symbol of type “expression” and “y + 1” is of type “binary-arithmetic-expression” which can be reached by “expression”. For another example, consider copying the left-hand-side “x” of the expression “x = y” and pasting it into an “expression” selection block placeholder. This time, “x” is not of type “expression” according to a traditional programming language grammar (this would allow illegal expressions such as “x + 1 = y”). In this case, pasting “x” into an “expression” placeholder still works and the system automatically generates the correct production path from “expression” to “x”, including all intermediate non-terminal symbols, so that deleting “x” will work as expected.

Copying a block is performed either by clicking on it in order to select it and pressing the keyboard shortcut “Ctrl + C”, or selecting “Copy” from the block’s right-click context-menu. Equivalently, pasting is available by the shortcut “Ctrl + V” or through the context-menu. When pasting the clipboard into the selected block would result in syntax errors, the pasting operation is not performed and the corresponding context-menu option is shown as disabled.

3.2.3 Quick Replace and Reverting Production Steps

Although copy-and-paste improves the editing capabilities of our system and generally reduces user-effort, there are still scenarios in which syntax-directed editing requires too much effort to perform tasks that are trivial in the context of a textual editor. An example commonly used in syntax-directed editing related research is the altering of an if-statement with defined conditions and statements to a while-statement with identical condition and statements. In traditional text editors, the user would simply be able to erase the keyword “if” one character at a time and insert the keyword “while” by typing one character at a time. Of course, this is not possible in syntax-directed editing due to the program being in an ill-formed state during the process of deleting and inserting. With the currently presented features, this simple operation would need four discrete steps. First a while-statement has to be inserted. Secondly, the condition of the if-statement has to be copied and pasted onto the condition of the while-statement. Thirdly, the if-statements’ statements have to be copied and pasted onto the statements of the while-statement. Lastly, the if-statement has to be deleted. Figure 27 depicts the steps of this process in Code-Chips.

For the purposes of simplifying such replacements in terms of user effort and time required, we introduce a “Quick Replace” feature that empowers the programming language author to specify simple conversions from a grammar symbol to another, allowing existing, already-produced symbols to be kept in the converted result. These available replacement symbols, then, can be displayed and presented to the user through the right-click context-menu of the block for conversion. With this feature, altering per say an “if-statement” to a “while-statement” is accomplished with a single simple step.

According to the input language grammar, there can be different applications of "Quick Replace". For a general purpose visual programming language, especially when aimed at beginners, the language author should distinguish between arithmetic, relational and Boolean expressions and provide different visual representations for each one, for instance by having different colored blocks. This is more important for a syntax-directed editor, such as Code-Chips, which aims to communicate language-specific structural information through interactive syntax. As a result, it is not recommended that arithmetic, relational and Boolean operators are under the same category which for the end-users means that changing per say an "x + y" arithmetic expression to "x > y" requires the extra steps of constructing a new relational expression, copying-and-pasting the operands and deleting the old arithmetic expression. With "Quick Replace", conversions such as the aforementioned become trivial while not polluting the input language grammar.

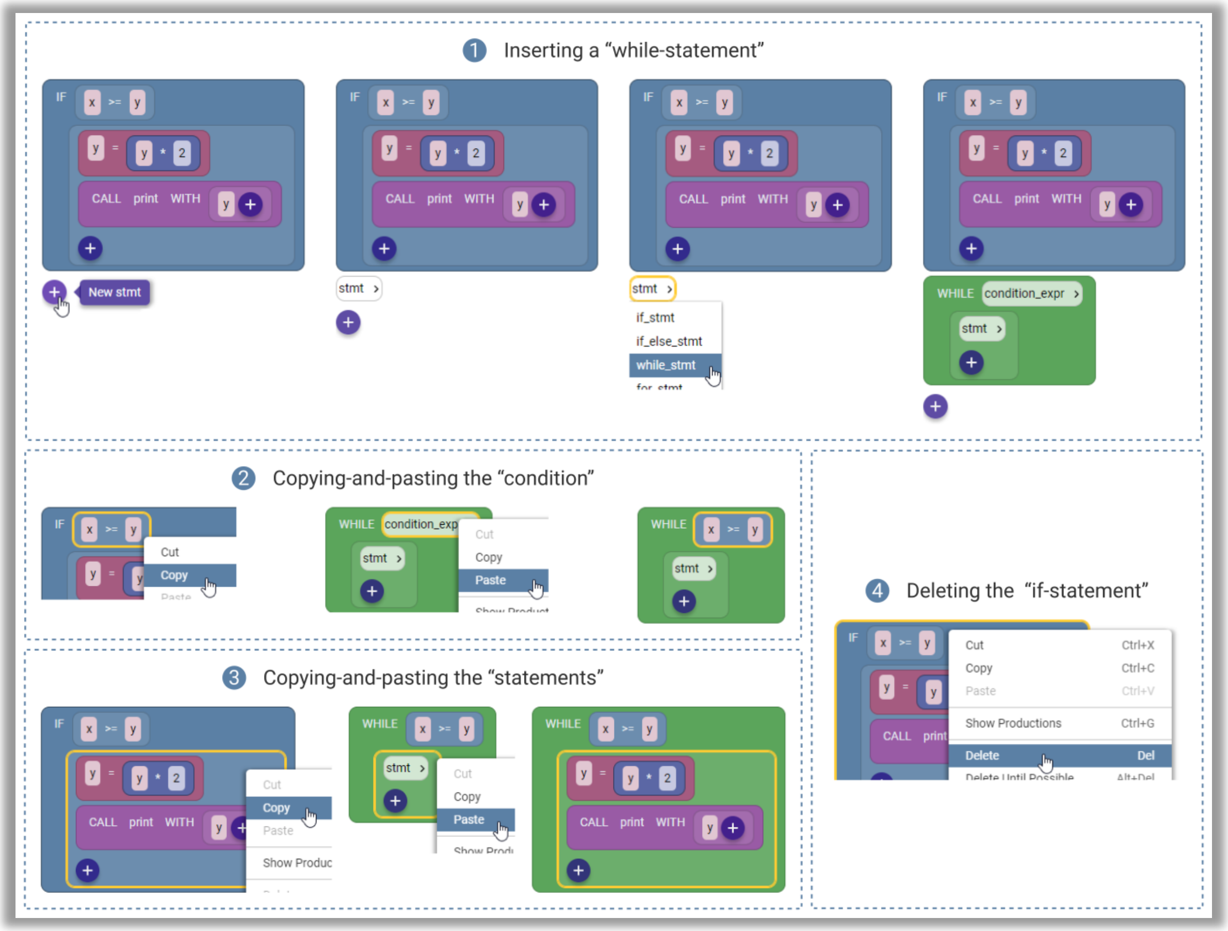


Figure 27*: Replacing an "if-statement" with a "while-statement", without the use of "Quick Replace"*

In the context of making syntax-directed deletion quicker, Code-Chips allows reverting multiple production steps at once, instead of using consecutive deletion operations. By right-clicking a block and selecting the “Reduce To” option, the system displays all the previously chosen symbols, allowing the user to replace the block with one of them. This is particularly useful in language grammars that have phrases with high depth as these phrases would need multiple selection block expansions in order to produce and thus would need multiple deletion steps to revert. Figure 28 depicts various simple examples.

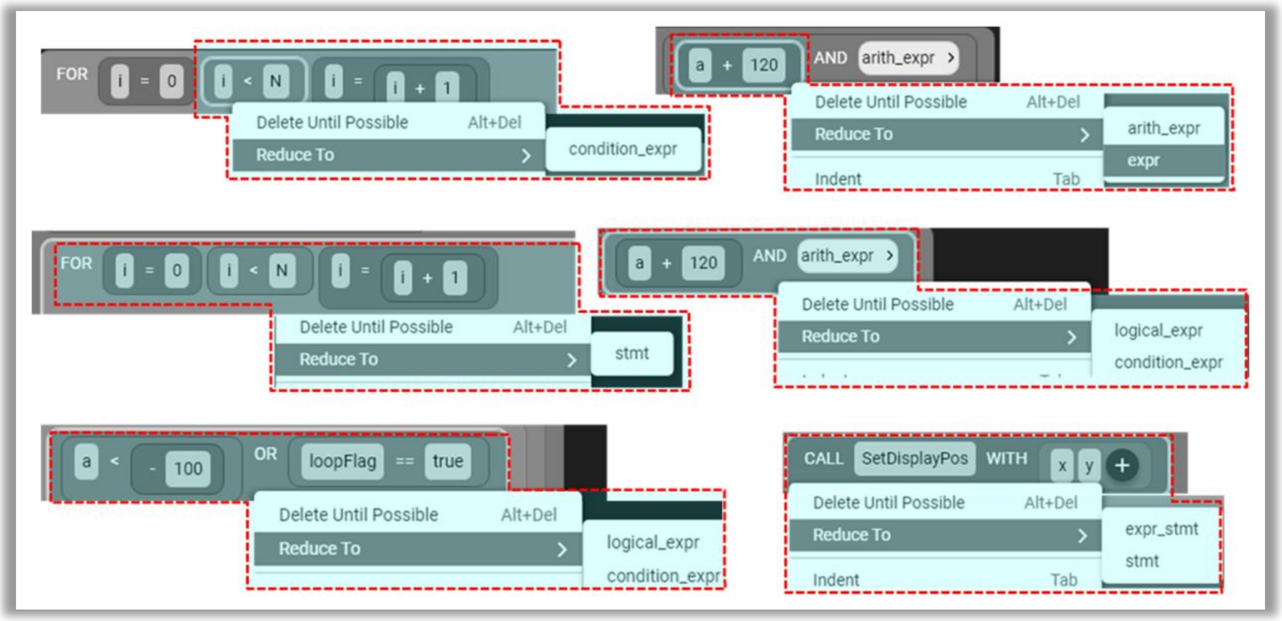

FOR i = 0 i < N i = i + 1
Delete Until Possible Alt+Del
Reduce To
condition_expr
a + 120 AND arith_expr
Delete Until Possible Alt+Del
Reduce To
Indent Tab
arith_expr
expr
FOR i = 0 i < N i = i + 1
Delete Until Possible Alt+Del
Reduce To
stmt
a + 120 AND arith_expr
Delete Until Possible Alt+Del
Reduce To
logical_expr
condition_expr
a < - 100 OR loopFlag == true
Delete Until Possible Alt+Del
Reduce To
logical_expr
condition_expr
CALL SetDisplayPos WITH x y +
Delete Until Possible Alt+Del
Reduce To
Indent Tab
expr_stmt
stmt

Figure 28: *Various examples of reverting blocks to their higher-level grammar symbols*

3.2.4 Toolbox, Drag-and-Drop and Visual Code Snippets

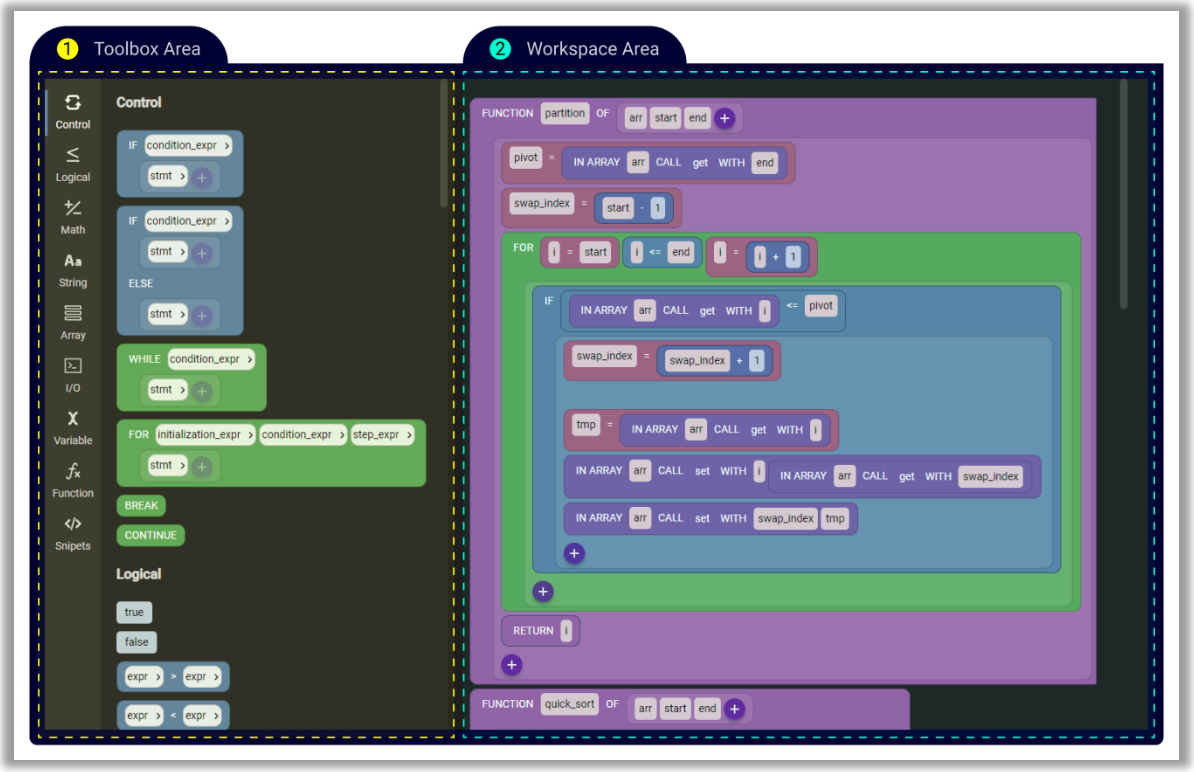

1 Toolbox Area
2 Workspace Area
Control
Logical
Math
String
Array
I/O
Variable
Function
Snipets
Control
IF condition_expr stmt
IF condition_expr stmt ELSE stmt
WHILE condition_expr stmt
FOR initialization_expr condition_expr step_expr stmt
BREAK
CONTINUE
Logical
true
false
expr > expr
expr < expr
FUNCTION partition OF arr start end
pivot = IN ARRAY arr CALL get WITH end
swap_index = start - 1
FOR i = start i <= end i = i + 1
IF IN ARRAY arr CALL get WITH i <= pivot
swap_index = swap_index + 1
tmp = IN ARRAY arr CALL get WITH i
IN ARRAY arr CALL set WITH i IN ARRAY arr CALL get WITH swap_index
IN ARRAY arr CALL set WITH swap_index tmp
RETURN i
FUNCTION quick_sort OF arr start end

Figure 29: *The toolbox area and the workspace area in Code-Chips*

Up to this point, the discussion was centered at syntax-directed editing and the features that we introduced were aimed at improving user experience in terms of structured editing. The only previously presented characteristic that refers to block-based visual programming is the rectangular colored visual representation we have chosen for templates. As previously discussed, block-based visual programming editors rely on the drag-and-drop gesture for visual-code insertions and modifications. In such environments, the common approach of supporting the drag-and-drop interaction is by providing a toolbox area with all the available blocks, categorized by their usage and semantics. Through this toolbox area, the user can drag any block and drop it into the workspace. The workspace usually consists of a canvas-like area in which blocks may be freely dropped to form programs. In the following paragraphs we will discuss features that enable block-based visual programming in the syntax-directed environment of Code-Chips.

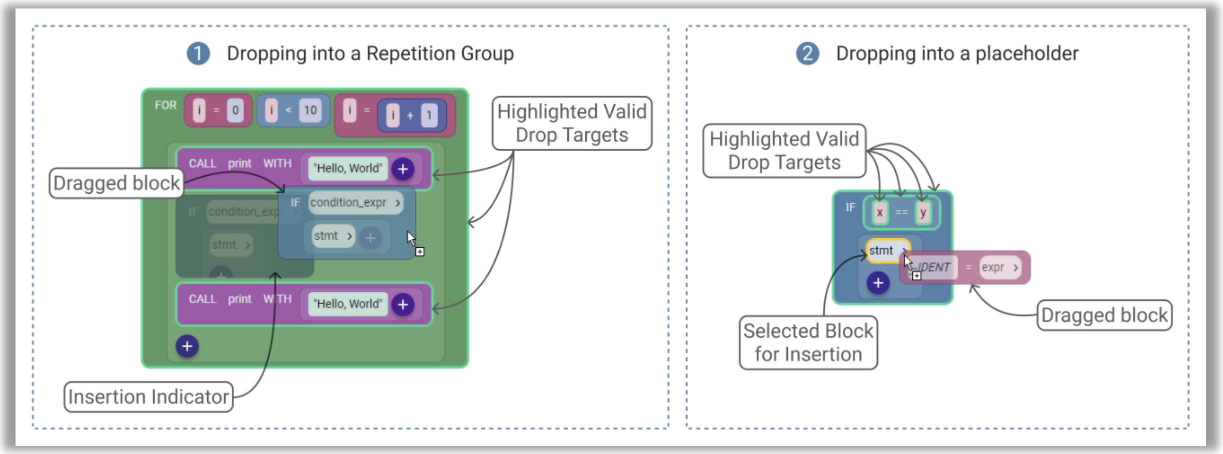

1 Dropping into a Repetition Group
Highlighted Valid Drop Targets
Dragged block
Insertion Indicator
FOR
CALL print WITH "Hello, World"
IF condition_expr
stmt
2 Dropping into a placeholder
Highlighted Valid Drop Targets
IF x == y
stmt
IDENT = expr
Dragged block
Selected Block for Insertion

Figure 30*: Two examples of drag-and-drop usage in Code-Chips*

Code-Chips follows the conventional block-based visual programming editor approach of dividing the user interface in two parts: the toolbox and the workspace (as depicted in Figure 29). The toolbox area in Code-Chips, is able to host categories of blocks. The system requires each category to be given a name and an icon which are rendered, and when clicked by the user, display the registered blocks for this category. The language author can set up the toolbox by defining the category names and icons in a configuration file. After this process, the language author can use the syntax-directed features of the workspace to form any phrase-block and include it in the toolbox by dragging it from the workspace and dropping it in the desired toolbox category. It is worth mentioning that the syntax-directed editing approach we have discussed only makes use of interacting with workspace blocks and hence does not require use of the toolbox.

For the end-user, the toolbox is a means through which ready-made language productions can be found and utilized. In order to use it, the user can drag blocks from the toolbox and drop them onto placeholder blocks in the workspace. Dropping uses the previously discussed rules for pasting, in order to avoid ill-formed programs and syntax-errors. Note that the workspace in Code-Chips does not allow dropping blocks in arbitrary positions – the system's structured line-based layout will be discussed in more detail later, in section 3.3. Despite that, the system allows drag-and-drop block insertions in-between blocks that are directly into repetition group blocks, without requiring a block placeholder to be present. For instance, inserting a ready-made "assignment" group block "x = y" as a new statement into the "statements" repetition group block of a while-statement, can be done by simply dragging the “x = y” block and dropping it into the “statements” block, in the desired position. The same task, without the usage of drag-and-drop, would require copying the “x = y” block, creating a new “statement” block by clicking the button of the “statements” repetition group block and pasting the copied block onto the newly created “statement” block. Figure 30 depicts two examples of drag-and-drop interaction in Code-Chips.

The toolbox does not impose requirements for its blocks – in fact any valid language grammar production can be hosted in it. Additionally, end-users are not only allowed to drag blocks from the toolbox, but are also empowered to expand its collection of blocks by dropping workspace block productions into the toolbox. According to these two statements, the toolbox can easily host an initially empty category with the name of “Snippets”, in which end-users are encouraged to drop their own formed blocks, empowering them to save reusable segments of code, for easier and quicker access. For instance, using this feature, a user can save a conventional “two-dimensional for-loop” with variables “i” and “j” for later uses.

3.3 Layout: Row-Based Indentation

The currently prevalent block-based visual programming editors use workspaces which operate in a canvas-like manner. With this approach, users are allowed to drop blocks anywhere within the workspace, without necessarily connecting them with each other. For Code-Chips we have chosen a row-based layout which additionally supports indentation. In order to accomplish this, we introduced two new block types, the new-line block and the tab block. The new-line block, similarly to a text-editor line-break, signifies the end of a line and causes blocks after it to start in a new line. The tab block is a simple transparent block with a fixed width, and can be used to indent the blocks that follow it.

For a better understanding of this row-based indented layout, consider the familiar layout of a text-editor: the user can insert a new-line character or a tab character in the position of the cursor, between any two consecutive characters. In Code-Chips, as in other block-based visual programming editors, editing operations are performed based on a user-selected block and there is no available cursor. As a result, inserting a new-line or tab, is performed at the position of the selected block: pressing the "Enter" key places the currently selected block on a new-line and pressing the "Tab" key increases its indentation. Similarly, the inverse operations of placing the selected block in the previous line or decreasing its indentation can be done by pressing the "Backspace" key.

Now, let us consider how the multidimensionality of syntax-directed editing affects the row-based layout. In contrast to text-editors, which process text as a one-dimensional stream of characters, syntax-directed editors process code as hierarchical compositions of computational structures. In our block-based environment, this hierarchical nature is visualized with the aid of group blocks. In this context, in order to support the aforementioned layout, new-line blocks and tab-blocks cannot be inserted globally but must be hosted by group blocks. When group blocks are the target of indentation or new-line commands, their inner blocks move along with them. Additionally, when an inner block is a target of such commands, the size of the parent group block is automatically altered vertically and horizontally to accommodate the change. Furthermore, all lines in a group share the same leftmost x-coordinate but can have different heights, determined by the block with the greatest height in the line and different length, determined by the total length of all blocks in the line. For instance, a traditional "if-statement" group block in Code-Chips, as in Figure 23, contains five blocks in this order: an "if" keyword simple block, a "condition" selection block, a new-line block, a tab block and a "statements" repetition group block.

Up to this section, the new-lines and tabs were not mentioned for simplicity, but the editing features and available block types of our system are designed to complement the row-based indented layout. For instance, the drag-and-drop gesture is allowed to be initiated by dragging a block and dropping it either onto a workspace block or in-between blocks of a repetition group block. Now that the reader has a better understanding of the underlying layout, the phrase *in-between blocks of a repetition group block* implies that the two blocks are separated by a new-line block. In this case, when dropping a block the system also inserts a separate new-line block in order to keep the, now three, blocks in separate lines. In the same manner, a "statements" repetition group block, per say, inserts a "statement" block along with a new-line block when its button is clicked.

In the remainder of this section we discuss the benefits of choosing a line-based layout with indentation. In particular, we discuss its advantages in structure, compared to the common canvas-like layout, demonstrate how it can be used to enable expressive user-defined code patterns and introduce a simple pretty print mechanism for Code-Chips blocks.

3.3.1 More Structured than Floating Blocks

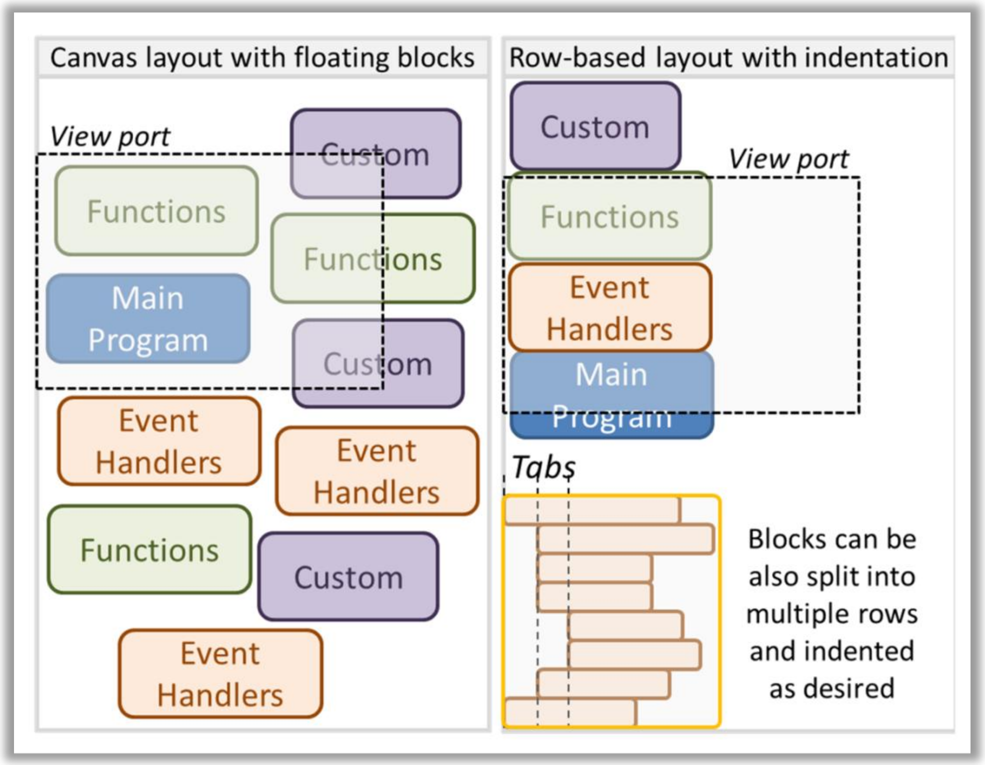


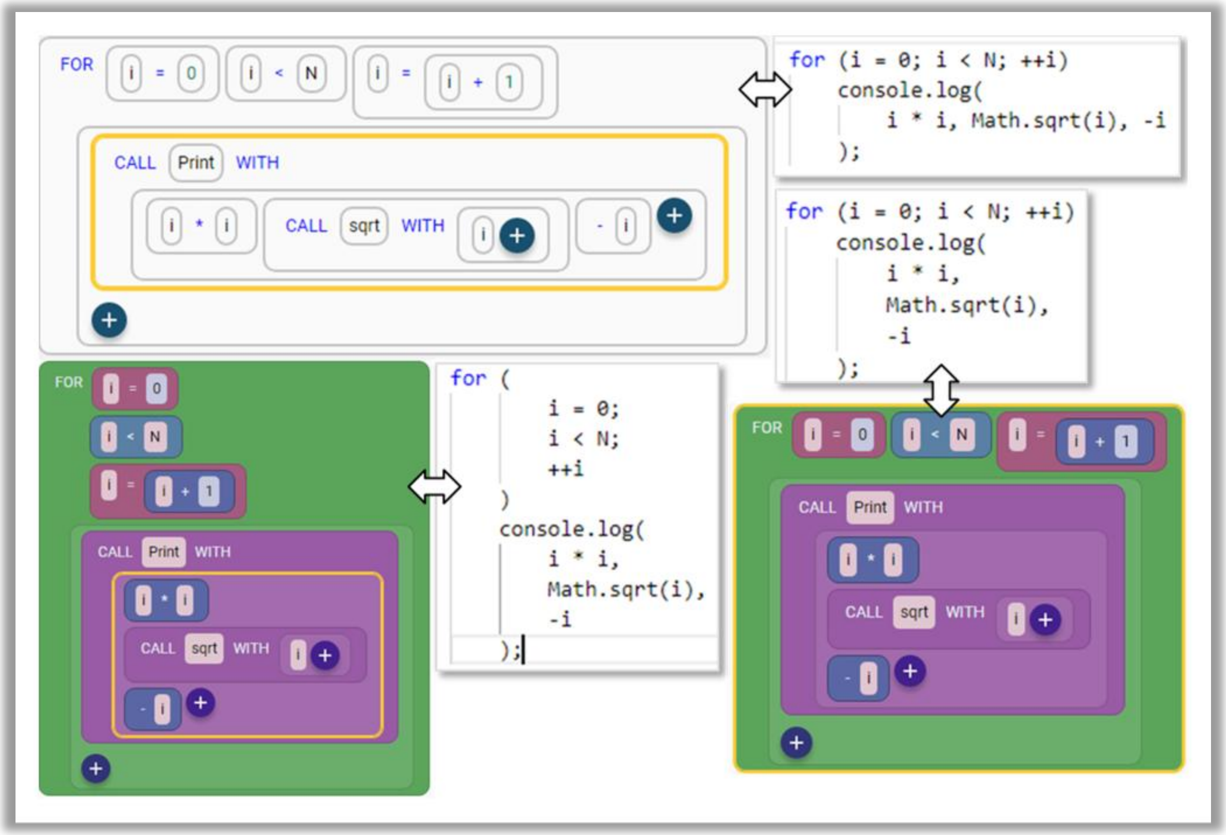
Figure 31*: A comparison, in terms of code organization, between the row-based layout with indentation (right) and the open canvas with floating blocks (left) currently adopted by most visual programming tools*

As previously stated, most block-based visual programming tools currently incorporate an open two-dimensional canvas-like area that allows placing blocks anywhere with enough space to fit them. With this approach, it is very easy to create more chaotic programs that lack in terms of code structure and organization. For a better understanding of this statement, let us consider the example of textual programming. Textual environments encourage users to author programs that have a limited character count per line and expand vertically. Even when following this direction, novices and even more experienced programmers, can easily derail from code organization guidelines, and author hard to read programs. When a textual program has no particular structure, locating per say a function definition can be as difficult as scanning the program vertically until the function is spotted. In the common canvas model of visual programming, semantically different program components such as event handlers and functions can be located in any arbitrary position in the two-dimensional canvas area. In this context, a program can be particularly more chaotic since scanning the program in a vertical manner is not enough to spot per say a specific function block – the user needs to examine the entire two-dimensional area. The left-part of Figure 31 illustrates this organizational issue. In addition, aligning blocks in terms of their x-coordinate is not always provided, even by widespread visual programming editors and environments. As a result, aligning per say two function definitions is a task that requires the end-users to move blocks in a free-hand manner.

The row-based layout of our system does not share these disadvantages. In contrast to the canvas approach, and in similarity to the text-based layout, each block can have a visually identifiable left-neighbor block and right-neighbor block, as well as a line it belongs to. This enables the end-user to navigate in group blocks by simply using the arrow-keys of their keyboard. With the addition of a “step-in” command and a “step-out” command, which navigate into and out of compound blocks respectively, the user can navigate to and select any block in the program even without using the mouse.

With our approach programs expand in a vertical manner, similarly to textual programming, facilitating the intrinsic organization of simple programs. In this context, our system can make further improvements in structured visual program design, utilizing its syntax-directed portion. In particular, with a language grammar designed with such aims, it is possible to enforce per say functions to be defined before any other visual code segment. This approach, combined with visual indications through grouping and block colorization, allows for more structurally clear visual programs, without necessarily appearing restrictive in the perspective of the end-users.

3.3.2 Enabling User-Defined Indented Code Patterns



```
for (i = 0; i < N; ++i)
    console.log(
        i * i, Math.sqrt(i), -i
    );
```

```
for (i = 0; i < N; ++i)
    console.log(
        i * i,
        Math.sqrt(i),
        -i
    );
```

```
for (
        i = 0;
        i < N;
        ++i
    )
    console.log(
        i * i,
        Math.sqrt(i),
        -i
    );
```

***Figure 32**: Examples of applying various indentation style patterns directly on visual blocks and their respective source-text JavaScript equivalent (the top-left block is shown under a different display theme)*

Textual programming editors empower users to employ custom code patterns by utilizing line-breaks and tabs, according to their personal preferences. For instance, a programmer can write an if-statement in the same line in its entirety, with no new-line or tab characters, when it contains a short single inner statement. A different programmer may still choose the more traditional option of placing the inner statement at a new-line and indenting it by one tab character. On the other hand, current visual programming editors employ blocks that have a predefined visual structure and do not allow any layout customization. As a result, personalized visual code patterns are not supported by such visual programming environments.

In Code-Chips, the addition of new-line blocks and tab blocks allows a variety of personalized user-defined code patterns. Similarly to textual editors, users can press

"Enter" or "Tab" to insert a new-line block or a tab block, respectively, at the position of the selected block, modifying the compound block's visual structure according to the situation and their personal preferences. Figure 32 depicts examples of custom indented code patterns and their JavaScript source-text equivalent.

3.3.3 Pretty Print

As previously mentioned, supporting the system's row-based layout requires group blocks and repetition group blocks to be able to host new-line blocks and tab blocks. In this context, the system can support blocks such as "if-statement" which usually require more than one line of code and need indentation. For a general purpose visual programming language, an "if-statement" block can be a group with three inner blocks: an "if" simple block, a "condition" selection block and a "statements" repetition group block. The user can then achieve its traditional form by placing a new-line block and a tab block before the "statements" block.

Although having blocks in their desired indented layout form is possible by hosting them in the toolbox, without any added support, syntax-directed insertions would produce blocks without any new-lines and tabs, resulting in one-line blocks. Additionally, repetition group blocks need information about whether to insert blocks in a new line, for program elements such as statements, or insert blocks in the same line, for program elements such as function call arguments. A solution to this would be to embed new-line and tab specific information into the input language grammar, but we value keeping the language grammar simple and believe that such an approach pollutes it.

A better solution, and what was implemented in Code-Chips, is a simple pretty print system based on a language-specific configuration file. In this configuration file, the language author can specify in which groups and in which positions the system needs to generate new-lines and tabs as well as which repetition group blocks generate new-lines along with their repetitive element. For instance, using this configuration file, the language author can specify that an "if-statement" group block needs a new-line block after its condition and a tab-block before its statements, as well as that a "statements" repetition group block needs a new-line between consecutive "statement" blocks.

This simple pretty print system can be used in two ways: automatically and on-demand. In its automatic usage, the editor applies it directly on every syntax-directed block insertion, resulting in visual code that automatically conforms into the provided pretty print settings. In its on-demand usage, the system applies it on the whole program - useful for the cases that the program has an unwanted structure due to mistakenly placed user tab and new-line insertions.

3.4 Reviewing: Code and Productions

Although syntax-directed editing can lay the foundation for an introduction to computer science and learning a programming language, we believe that beginner programmers and learners can significantly benefit from reviewing ready-made programs, not only in the context of problem solving but also in terms of syntactic language expressiveness. In current block-based visual programming editors, the user is given the ability to review a translation of their visual script to a textual programming language, usually JavaScript, to increase familiarity with text-based programming. In this section, we will discuss features that enable end-users to review programs and language grammar productions in Code-Chips.

3.4.1 Selectively Visualizing Productions

Syntax-directed editing, in Code-Chips, is based on expanding grammar symbols to their production right-hand-side and typing simple input through keyboard, by interacting with selection blocks and input blocks respectively. We introduce a new feature that empowers the user, to view the production chain of any block at any point – that is to review the steps needed to produce a given phrase. In Figure 25, we show how an “x = y” assignment can be produced with discrete simple steps using custom hand-made arrows and screenshots of Code-Chips blocks. With the introduction of production path visualization, the user can obtain a similar result through the system, accompanied by an interactive tree-view. Figure 33 depicts the production path for an “x = y” assignment, as visualized in Code-Chips. This feature can be utilized by tutors in the process of teaching a new language, or by end-users themselves when applied on ready-made programs, constructed by others.

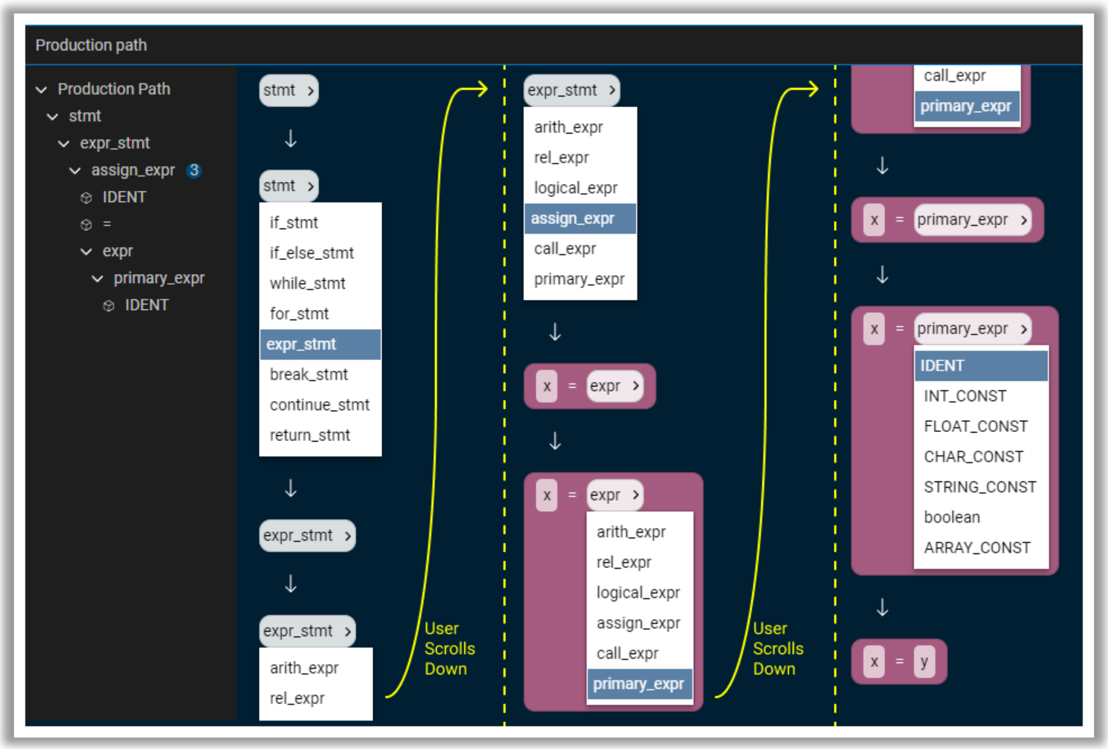
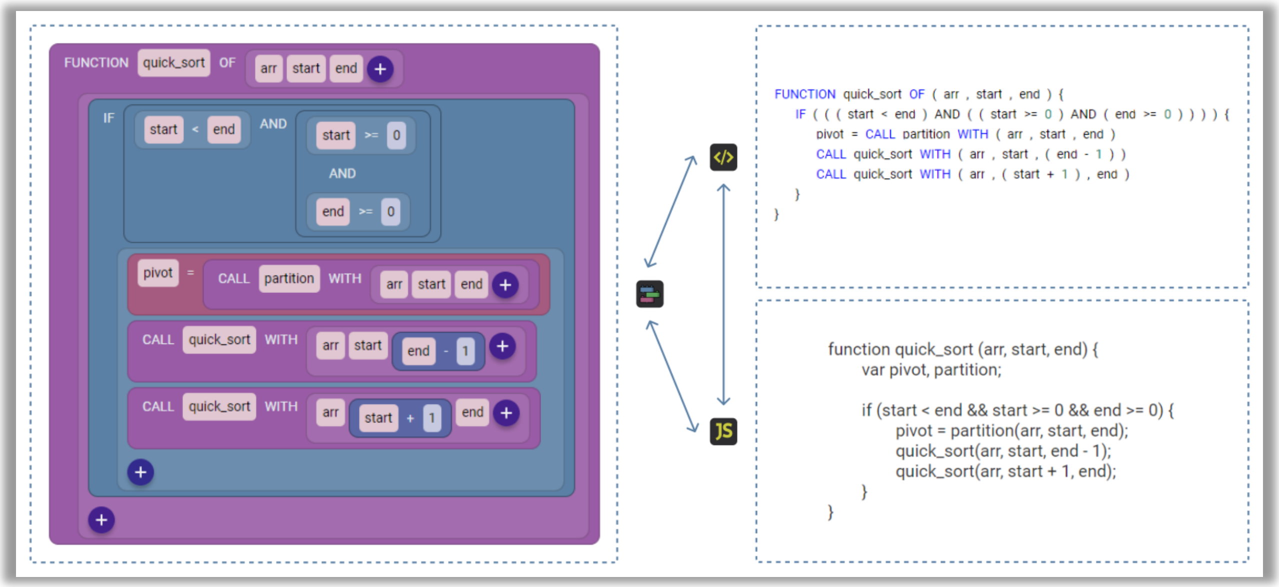


Figure 33*: An "x=y" production visualization in Code-Chips.*

3.4.2 Viewing Blocks as Source Text or JavaScript

Although current visual programming editors usually provide translations to textual programming languages, such as JavaScript, they do not provide functionality for viewing the source-text representation of the visual program. This cannot simply be a mere conversion from blocks to their inner text, as the extra dimension of grouping allows visual programming languages to omit tokens such as braces and parentheses. For example, the expression "(2*2) + 1" does not require parentheses in its block representation, as "2*2" is a visually distinct group block. In such cases, converting to source-text requires adding parentheses to the block's output source-text. As stated previously, we prefer keeping the language grammar simple and thus opt to a separate configuration file for storing this information. In this configuration file, the language author can specify, per group block, any extra characters and their positions for correct source-text translation. The same configuration file also serves for defining the source-text pretty print settings, which can be different from the pretty print settings for blocks, since source-text translation may need additional characters such as parentheses.


FUNCTION quick_sort OF arr start end
IF start < end AND start >= 0 AND end >= 0
pivot = CALL partition WITH arr start end
CALL quick_sort WITH arr start end - 1
CALL quick_sort WITH arr start + 1 end
JS

```
FUNCTION quick_sort OF ( arr , start , end ) {
   IF ( ( ( start < end ) AND ( ( start >= 0 ) AND ( end >= 0 ) ) ) ) {
      pivot = CALL partition WITH ( arr , start , end )
      CALL quick_sort WITH ( arr , start , ( end - 1 ) )
      CALL quick_sort WITH ( arr , ( start + 1 ) , end )
   }
}
```

```
function quick_sort (arr, start, end) {
      var pivot, partition;

      if (start < end && start >= 0 && end >= 0) {
            pivot = partition(arr, start, end);
            quick_sort(arr, start, end - 1);
            quick_sort(arr, start + 1, end);
      }
}
```

***Figure 34**: Switching between viewing blocks, embedded language source-text and JavaScript*

With the aforementioned setup by the language author, the end-user then can, at any time, view the program in a textual form within the system by clicking an appropriate button. By using this feature, the user can increasingly become more familiar with textual programming and eventually transition to text-based environments. Obtaining the source-text becomes particularly interesting when the language embedded into Code-Chips is an existing programming language. For instance, a teacher that serves the role of the language author can provide a grammar specification for C, or a subset of it, and host it in Code-Chips. Their students then can benefit from the syntax-directed and visual programming capabilities of the system and avoiding syntax-errors, but can also view their programs in pure C. Additionally, the language author can setup the system to make a request to an available C compiler, when the program is run, meaning that execution is possible without implementing a runtime environment.

The system also supports translating blocks to JavaScript. For this feature, similarly to current visual programming editors, the system requires the language author or an experienced programmer to specify the translation through code. As with viewing the source-text of the embedded language, the end-user can inspect the JavaScript equivalent of their visual script with the click of a button. Figure 34 depicts a program viewed in all available modes: blocks, embedded language source-text and JavaScript.

3.5 Syntax-Driven Language Exploration

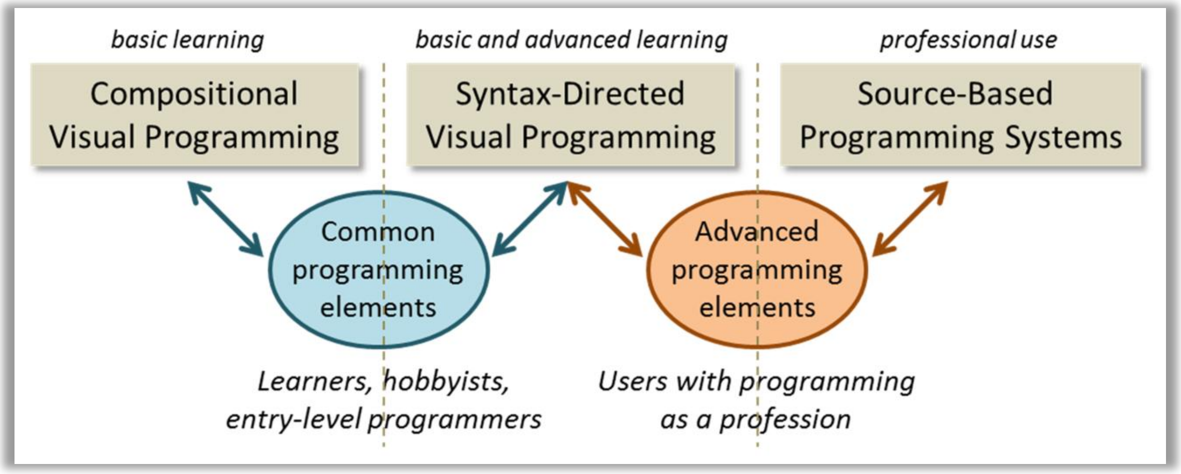

basic learning
basic and advanced learning
professional use
Compositional Visual Programming
Syntax-Directed Visual Programming
Source-Based Programming Systems
Common programming elements
Advanced programming elements
Learners, hobbyists, entry-level programmers
Users with programming as a profession

Figure 35: *The place of syntax-directed visual programming, bridging the gap between compositional visual programming and source-text-based programming systems*

In this chapter we have focused on introducing and analyzing features of our system that empower users to edit, review and structure programs, combining elements and characteristics from the disciplines of syntax-directed editing and block-based visual programming. Although, we have presented how users may benefit from such features in terms of efficient editing, we have only briefly demonstrated how the system and its functionality can be beneficially used in terms of teaching and what is the impact for the end-user's learning programming experience.

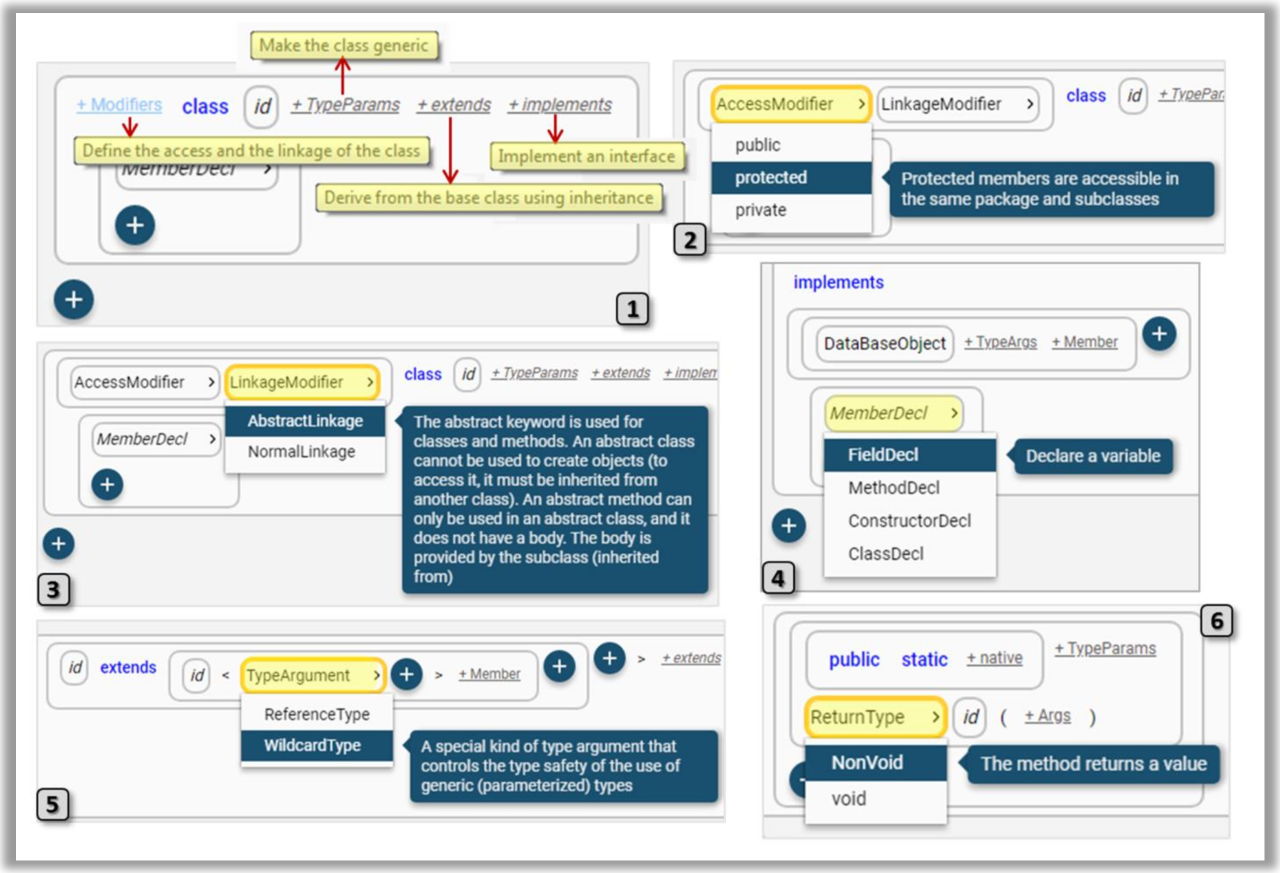
As stated before, learning programming is a demanding process and it requires as well as teaches and trains important cognitive skills that transfer to other disciplines and everyday life. Visual programming is commonly considered as a tool primarily targeted to learners, entry-level programmers and generally non-professional programmers. Through visual programming, young people can improve their creative thinking, problem solving and reasoning skills. Additionally, visual programming commonly serves as a stepping stone for easing the transition to more elaborate text-based programming environments. We discuss how our syntax-directed visual programming approach can assist beginners and students as well as more experienced developers.

In the context of introducing beginners to programming, although the current visual programming systems offer syntactic safety, they fail to directly communicate the underlying syntax of the used visual programming language, resulting in incomplete understanding of available language expressiveness. With Code-Chips, syntax is embedded and integrated into editing. With the help of a tutor, a beginner can be taught about an imperative language's structure and comprehend its semantics in depth, all while editing and reviewing programs. In this way, there can be absolute certainty about the productions that are allowed or forbidden. Through interacting with selection blocks, the user can explore all available possible expansions and read short but descriptive tooltips that communicate their semantics. Through deleting blocks, reverting productions and undoing-redoing expansions and deletions the user can easily navigate forward and backward, without being punished for making mistakes. As a result, the user is encouraged to explore the used visual programming language in its entirety. By reviewing productions in ready-made programs, a user can observe how common program elements are constructed and a tutor can teach their students the structure of a language. On this basis, a more insecure user can start from reviewing programs and slowly progress into composing their own.

The benefits of using a more structured, syntax-directed approach are more apparent when end-users are pursuing a transition to textual programming for personal or professional use. As previously stated, according to Chomsky, the study of a language expands the individual's universal language understanding and facilitates subsequent learning processes for different languages. In this context, learning a visual programming language's underlying syntax will assist the end-user in their later steps of text-based software development. Code-Chips' features of reviewing the source-text equivalent of a visual program, as well as the JavaScript equivalent, aim in further facilitating the transition from visual programming to textual programming. In addition, the row-based and indented layout for blocks that we have introduced is very similar to that of a textual code editor. Experimenting with indented patterns, navigating to the previous or next line, or to the previous or next block as well as reordering lines of blocks with simple shortcuts, all make Code-Chips look and feel closer to code than traditional visual programming editors.

Last but not least, diving into the structure of a programming language can be of significance to more experienced programmers and even professional software developers.

A programmer usually learns about a new language's syntactic specifications from online documentation, code examples, as well as by practical experimentation. While the latter two are able to produce quicker results, they can result in an incomplete understanding, at least for complicated and demanding language features. For instance, a programmer might know the basics of using C++ templates but may lack a deeper understanding, which in practice limits their potent use. In this context, syntax-driven visual programming systems, such as Code-Chips, can be used to assist professional developers to more easily experiment with and ultimately learn new advanced programming language features. Figure 36 depicts Java class definitions hosted by Code-Chips.



Make the class generic

Define the access and the linkage of the class

Implement an interface

Derive from the base class using inheritance

Protected members are accessible in the same package and subclasses

The abstract keyword is used for classes and methods. An abstract class cannot be used to create objects (to access it, it must be inherited from another class). An abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from)

Declare a variable

A special kind of type argument that controls the type safety of the use of generic (parameterized) types

The method returns a value

***Figure 36**: Interactive visual editing of class definitions in Java, with syntax-driven assistance: (1) the initial block with all tooltips on grammar symbols shown; (2) access modifiers; (3) linkage modifiers; (4) class members; (5) base classes and generic bases; and (6) methods, in particular return types.*

3.6 Block and GUI Theme Configuration

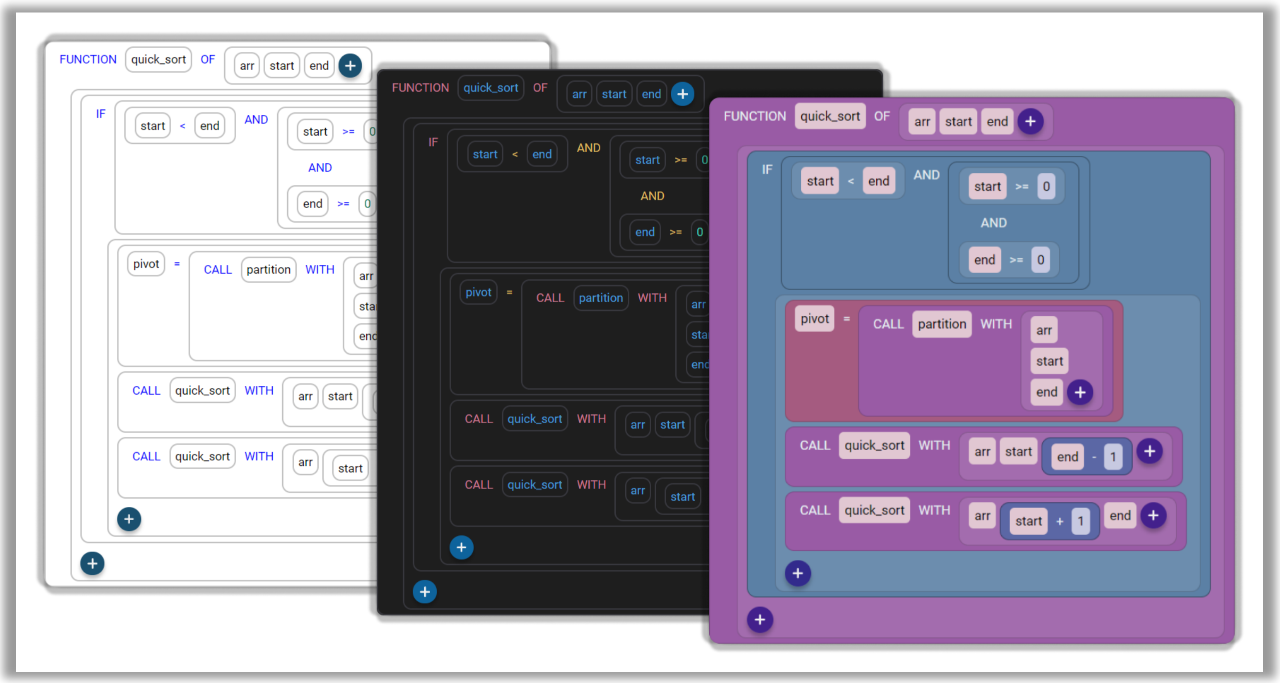


Figure 37: *Displaying blocks in the default white and light themes and in a custom colorful theme*

Due to the high count and variance of visual programming applications, ranging from educational to professional use, Code-Chips provides block and user interface customization. Through altering values in configuration files, the language author is able to customize each user interface component, including the toolbox and the syntax-directed editor. Customization properties include background colors, borders, font-sizes, font-colors and font-styles, padding and margins, scrollbar styling and more. Using these facilities, Code-Chips allows custom user interface themes, which make it adaptable and able to be integrated into and blend in with any visual programming environment.

Other than altering the appearance of the system's user interface, Code-Chips provides configuration capabilities for block customization. In contrast to the user interface themes, which are agnostic to the embedded language and can be reused by different language authors, block customization is language dependent. The system, upon embedding a language, can export a configuration file which includes customization settings for each grammar symbol. Again, these settings include general properties for background colors, borders, font-styles, padding and margins, etc. Additionally, each different block type is configurable in different ways. For instance, selection blocks offer customization properties for their dropdown menu that can have, per say, a different background color and font-style than the main part of the block. In this context, elements such as the dropdown menus of selection blocks have customizable properties on user interaction such as on-mouse-hover change of background color. Similarly, repetition group blocks offer separate customization for their "+" button which includes altering it on-mouse-hover.

In order to facilitate block customization, the language author does not have to fill-in the properties for every grammar symbol. The configuration file not only contains properties per grammar symbol, but also allows customization per block category. Using this infrastructure the language author can configure settings that apply to all the grammar symbols of the same type, for instance, changing the background-color for the dropdown menus of every symbol that corresponds to a selection-block. Specific grammar symbols, then, can use different customization properties or override properties that are already defined by their block category. For instance the language author can define common properties for every group block, including their padding, inner margins and border radius. Then, to differentiate, per say, a "for-statement" and an "if-statement" group block, the language author can skip specifying their common properties and only specify per say different background and border colors. Figure 37 depicts a visual code example in three different themes.

Chapter 4

Implementation

In this chapter we will present an overview of our system's software architecture and provide information about our implementation. For the purposes of this thesis, we do not mention and analyze every component and do not provide implementation details for every feature. Instead we focus on components and features that we consider important for forming a better understanding of the system.

4.1 System Overview and Software Architecture

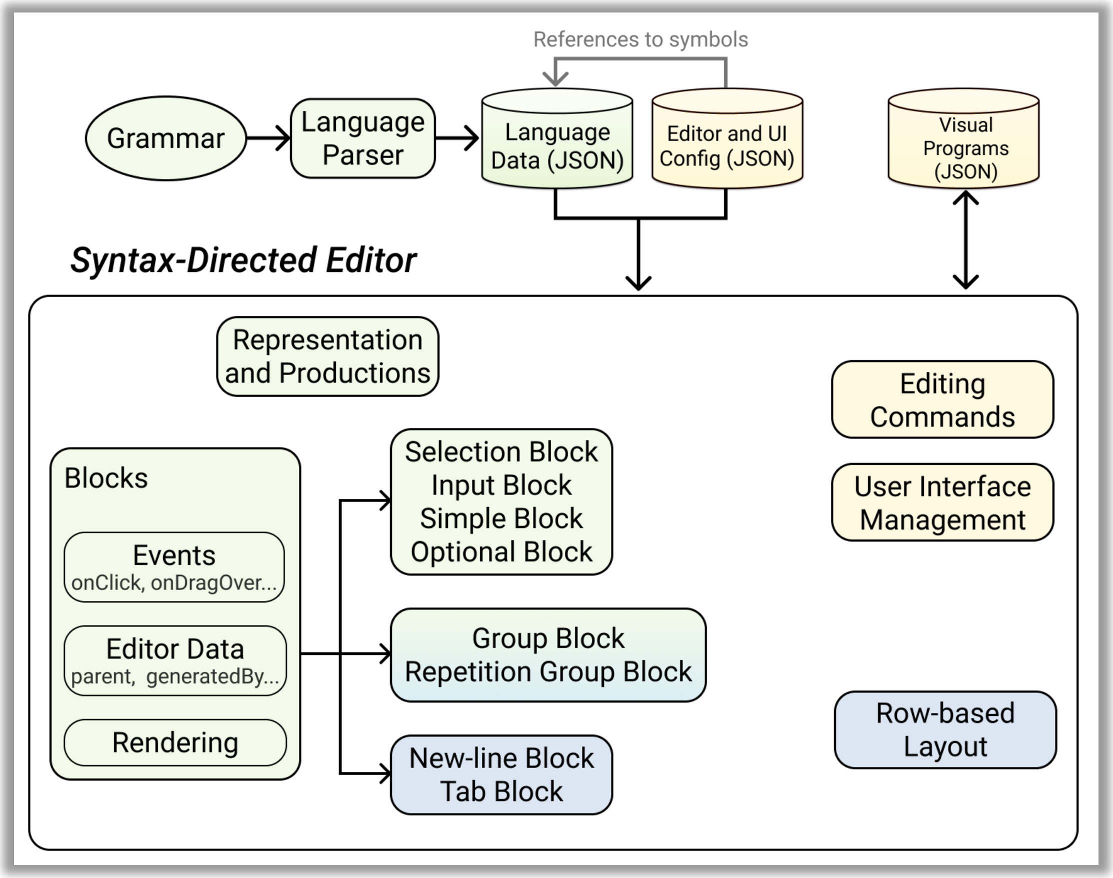


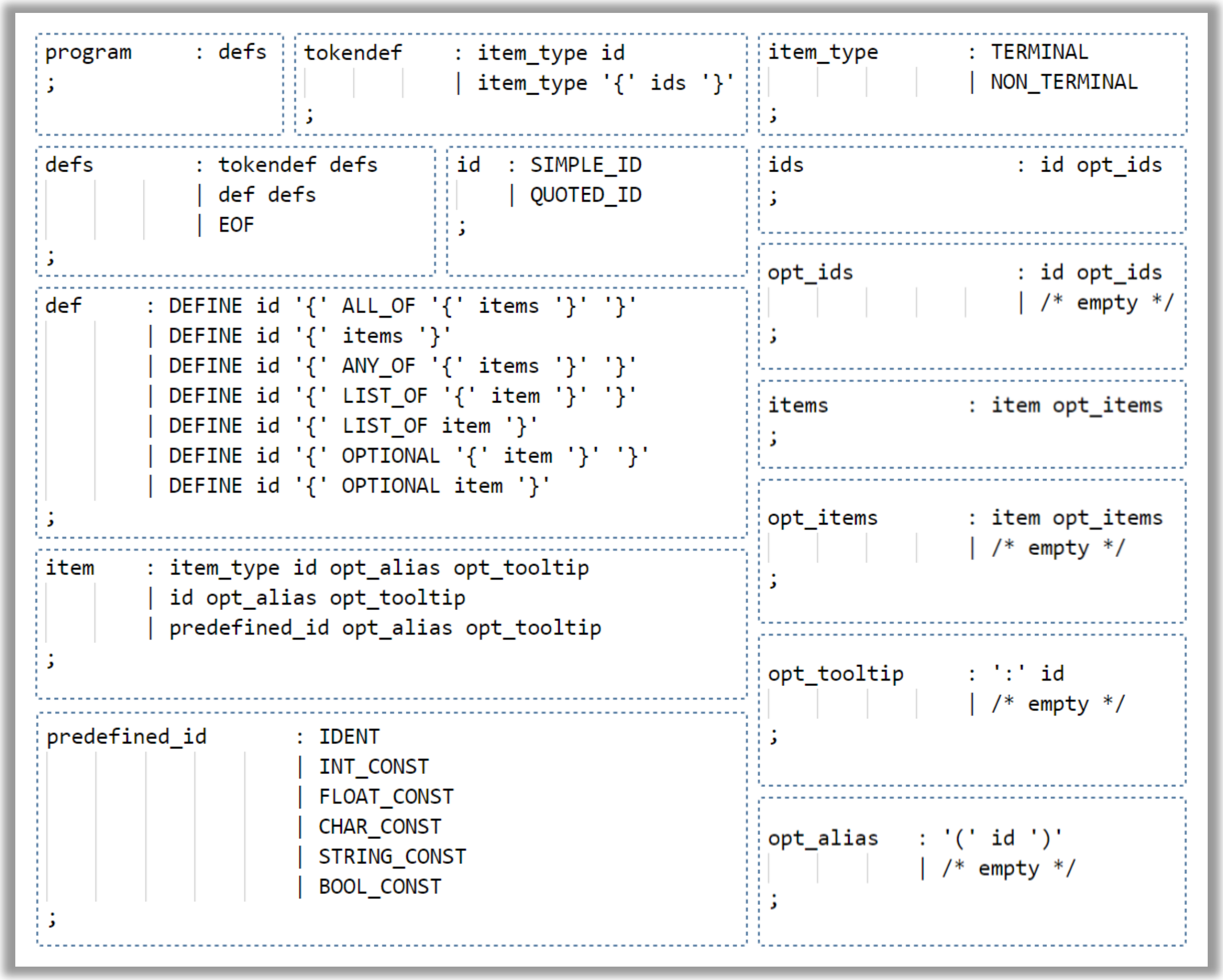
***Figure 38**: An overview of the system's software architecture*

In this section the reader can learn more about our system from an engineering and software development perspective. The system is a web application written mainly in vanilla JavaScript, HTML and CSS. For assistance with DOM element manipulation, we have utilized jQuery [32]. It was our intent to use as little external libraries as possible for easier system maintenance. In this context, we are considering removing jQuery and proceeding entirely with vanilla JavaScript in future versions.

Figure 38 provides an overview of our system's software architecture, depicting the primary editor components, as well as language related configuration. As previously stated, the system can host any programming language given a grammar specification according to the Code-Chips meta language. For this purpose we have implemented a language parser that validates the input programming language and outputs a corresponding JSON file, readable by the syntax-directed editor. The system, given the input language, also outputs configuration files that can be filled in by the language author and are used to configure the user interface. Configuration files are mostly language specific and contain grammar symbol references in order to provide adequate customization, for instance, different themes or pretty-print settings for blocks that represent specific grammar symbols.

The system, when provided with JSON files for the input language and its configuration, generates a corresponding syntax-directed visual programming editor. Grammar symbols are represented by a variety of interactive blocks, such as input blocks and selection blocks. Each of these blocks extends a generic block class, implements methods for being rendered and accepts handlers for events triggered by user actions. With this functionality the editor can generate, render, and connect blocks to user-actions by specifying their appropriate handlers. Group blocks can host any block and additionally, provide functionality for inserting, deleting and generally manipulating inner blocks. Visual code is represented in the form of an AST, with its root being a group block. The editor, when the end-user interacts with the user interface, executes editing commands which manipulate the visual code, utilizing the API provided by group blocks. Each editing command provides functionality for executing, undoing and redoing it. The editor's row-based layout is simply accomplished by inserting and deleting tab blocks and new-line blocks, which when rendered modify the page appropriately. The editor, using the API provided by blocks, can issue a command to obtain the visual code in JSON format, and export it for the end-user. Of course, the editor supports the inverse command of importing code in JSON format. For code execution and translation to JavaScript, we provide an AST visitor class and an AST host class, which the language author or an experienced programmer can use In order to map out visual code to JavaScript. In the following sections of this chapter we will discuss the implementation of our system in more detail and provide relevant code samples.

4.2 Meta Language and Parser

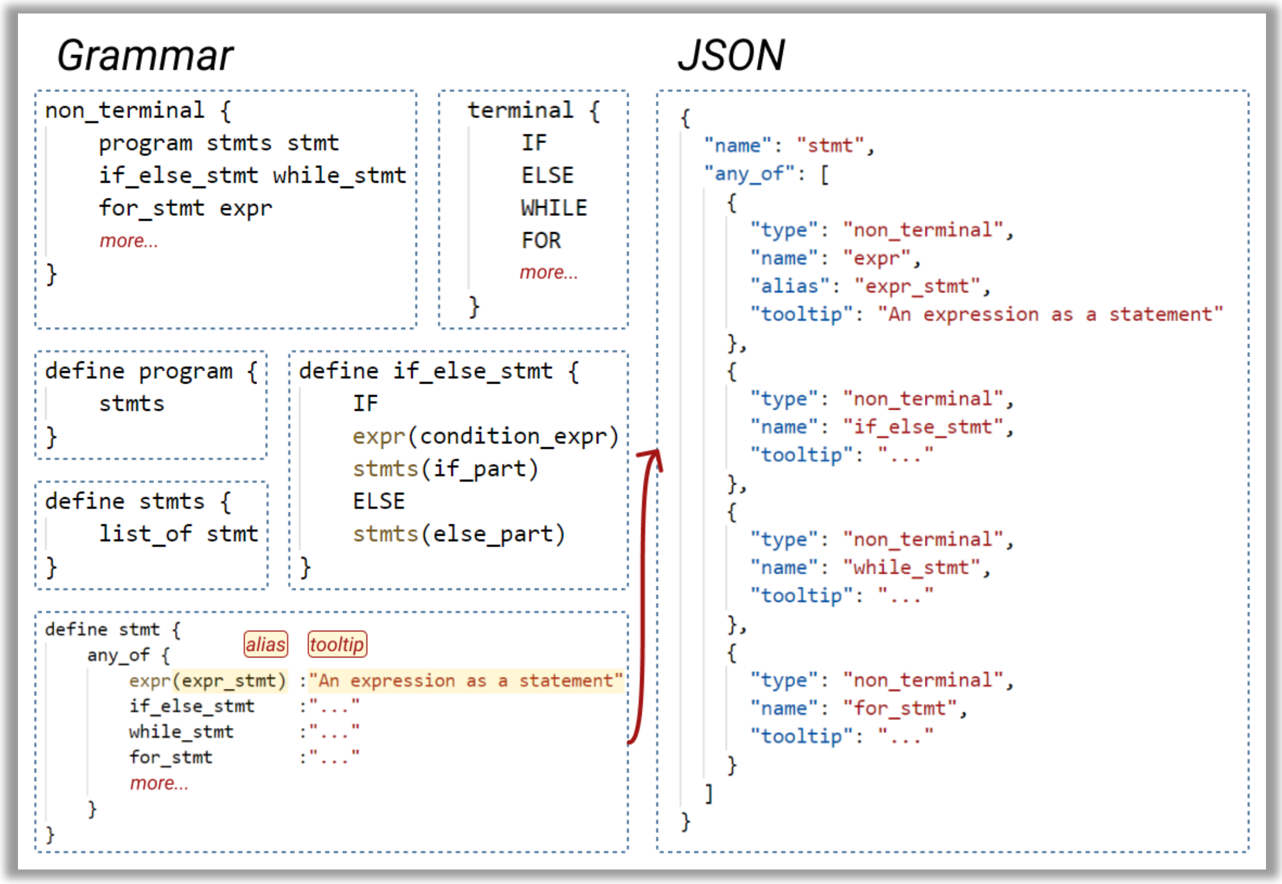


```
program         : defs
;

tokendef        : item_type id
                | item_type '{' ids '}'
;

item_type       : TERMINAL
                | NON_TERMINAL
;

defs            : tokendef defs
                | def defs
                | EOF
;

id    : SIMPLE_ID
      | QUOTED_ID
;

ids                     : id opt_ids
;

opt_ids                 : id opt_ids
                        | /* empty */
;

def     : DEFINE id '{' ALL_OF '{' items '}' '}'
        | DEFINE id '{' items '}'
        | DEFINE id '{' ANY_OF '{' items '}' '}'
        | DEFINE id '{' LIST_OF '{' item '}' '}'
        | DEFINE id '{' LIST_OF item '}'
        | DEFINE id '{' OPTIONAL '{' item '}' '}'
        | DEFINE id '{' OPTIONAL item '}'
;

items               : item opt_items
;

opt_items           : item opt_items
                    | /* empty */
;

item    : item_type id opt_alias opt_tooltip
        | id opt_alias opt_tooltip
        | predefined_id opt_alias opt_tooltip
;

opt_tooltip         : ':' id
                    | /* empty */
;

predefined_id           : IDENT
                        | INT_CONST
                        | FLOAT_CONST
                        | CHAR_CONST
                        | STRING_CONST
                        | BOOL_CONST
;

opt_alias       : '(' id ')'
                | /* empty */
;
```

Figure 39*: The Code-Chips meta language grammar, in which terminals are denoted in capital letters or are in quotes (SIMPLE_ID and QUOTED_ID are defined in a different lexer file and accept textual input)*

Integrating a programming language into Code-Chips requires specifying its grammar in a valid form. For this purpose, we provide an expressive but simple meta language (Figure 39), that contains type definitions for each terminal or non-terminal grammar symbol and the corresponding non-terminal symbol productions. Each production definition consists of the “define” keyword followed by an identifier for the symbol and the definition right-hand-side inside braces. Each definition right-hand-side consists of its type and its right-hand-side symbols, which we call items. The definition type can be any of the tokens “all_of”, “any_of”, “list_of” and “optional”. The definition type can also be left empty, in which case it is interpreted as “all_of”. While the types “all_of” and “any_of” can accept multiple items in braces, the types “list_of” and “optional” accept just a single item and it is valid to omit braces. These types will be later used by the editor to generate the appropriate block for each symbol. A definition item is an identifier, which provides a reference to another defined terminal or non-terminal symbol, and is accompanied by an optional alias and an optional tooltip, which allow the language author to provide context specific information. For instance, an “expression” symbol that represents the condition of an if-statement can be aliased as “condition” and be given an appropriate tooltip, but grammatically it remains an expression for the purposes of expansion.

After the language author provides an input grammar according to the Code-Chips meta language, the system generates a JSON file that can be used by the editor to host the given language. This process is handled by a parser written with Jison [33], which is a JavaScript version of Bison [34] and generates bottom-up parsers in JavaScript. The output file contains all the information provided by the input grammar, converted into a JSON object. In particular, this JSON object contains just a single key, “defs”, with a value of an array of objects, each representing a non-terminal definition. Theoretically, the language author can define the input language directly in JSON form, since the parser’s output is still humanly readable, but using the meta language is easier, takes less typing and is more elegant. Figure 40 depicts an example input grammar and a sample corresponding generated JSON object for one of its definitions. In section 5.1, the reader can find additional examples based on a fully-working general-purpose visual programming language we have integrated into the system.



```
non_terminal {
    program stmts stmt
    if_else_stmt while_stmt
    for_stmt expr
    more...
}
```

```
terminal {
    IF
    ELSE
    WHILE
    FOR
    more...
}
```

```
define program {
    stmts
}
```

```
define stmts {
    list_of stmt
}
```

```
define if_else_stmt {
    IF
    expr(condition_expr)
    stmts(if_part)
    ELSE
    stmts(else_part)
}
```

```
define stmt {
    any_of {
        expr(expr_stmt) :"An expression as a statement"
        if_else_stmt    :"..."
        while_stmt      :"..."
        for_stmt        :"..."
        more...
    }
}
```

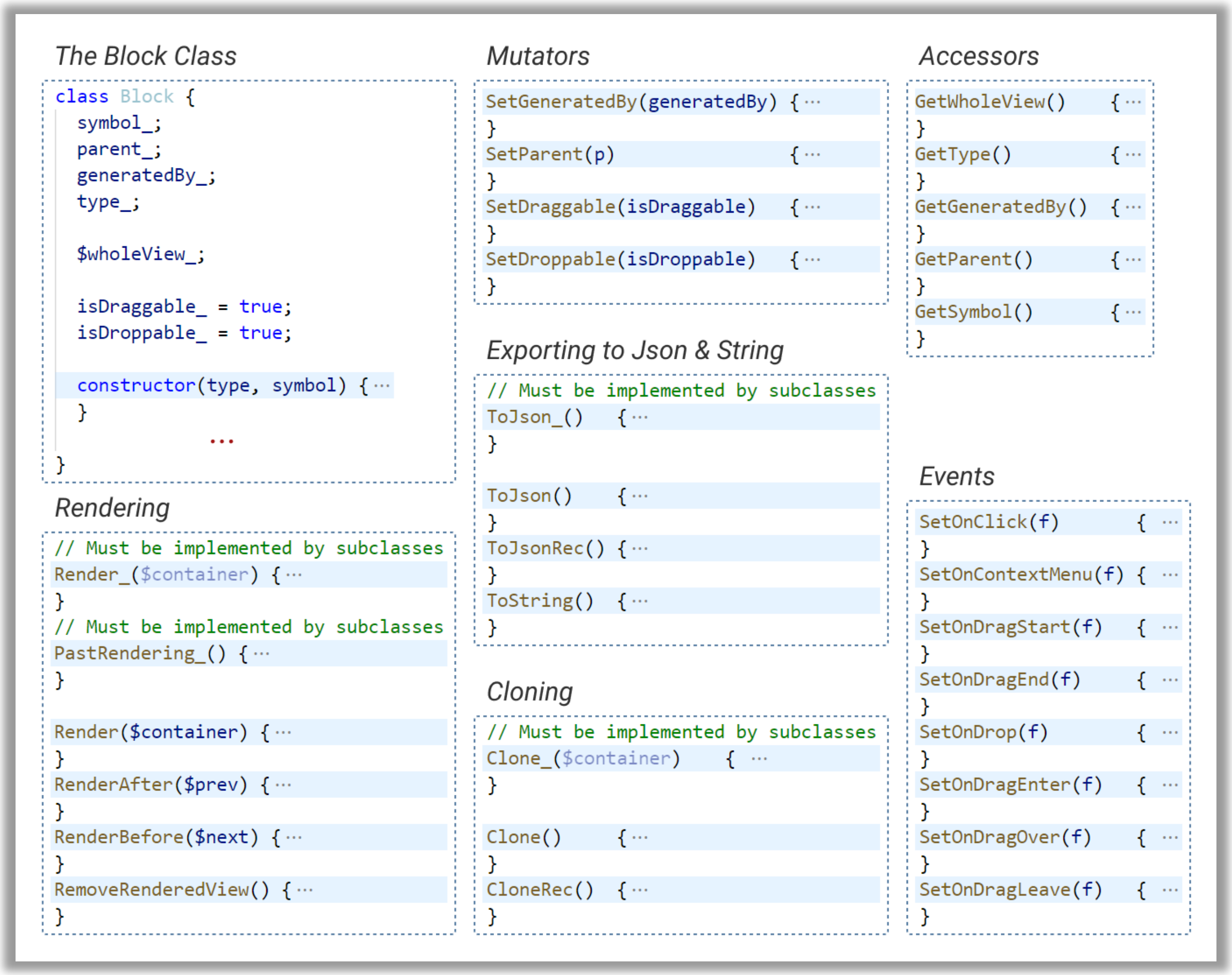
```
{
  "name": "stmt",
  "any_of": [
    {
      "type": "non_terminal",
      "name": "expr",
      "alias": "expr_stmt",
      "tooltip": "An expression as a statement"
    },
    {
      "type": "non_terminal",
      "name": "if_else_stmt",
      "tooltip": "..."
    },
    {
      "type": "non_terminal",
      "name": "while_stmt",
      "tooltip": "..."
    },
    {
      "type": "non_terminal",
      "name": "for_stmt",
      "tooltip": "..."
    }
  ]
}
```

***Figure 40**: An example conversion from grammar to JSON*

4.3 Interactive Blocks

In Code-Chips, blocks are designed to allow syntax-directed editing operations such as insertion and deletion, as well as block-based visual programming operations such as drag-and-drop. Particularly, each block type has a significant role in the process of syntax-directed editing and represents a different set of grammar symbols. In this section, we will discuss the implementation behind blocks, as well as the functionality and the API that is provided to the editor. Before focusing on each block separately, we will review the Block base class, which mainly provides functionality for rendering, adding event handlers, exporting to JSON format and cloning.

4.3.1 The Block Base Class



The Block Class

```
class Block {
  symbol_;
  parent_;
  generatedBy_;
  type_;

  $wholeView_;

  isDraggable_ = true;
  isDroppable_ = true;

  constructor(type, symbol) {…
  }
             …
}
```

Rendering

```
// Must be implemented by subclasses
Render_($container) {…
}
// Must be implemented by subclasses
PastRendering_() {…
}

Render($container) {…
}
RenderAfter($prev) {…
}
RenderBefore($next) {…
}
RemoveRenderedView() {…
}
```

Mutators

```
SetGeneratedBy(generatedBy) {…
}
SetParent(p)                {…
}
SetDraggable(isDraggable)   {…
}
SetDroppable(isDroppable)   {…
}
```

Exporting to Json & String

```
// Must be implemented by subclasses
ToJson_()    {…
}

ToJson()     {…
}
ToJsonRec() {…
}
ToString()   {…
}
```

Cloning

```
// Must be implemented by subclasses
Clone_($container)     { …
}

Clone()      {…
}
CloneRec()   {…
}
```

Accessors

```
GetWholeView()     {…
}
GetType()          {…
}
GetGeneratedBy()   {…
}
GetParent()        {…
}
GetSymbol()        {…
}
```

Events

```
SetOnClick(f)          { …
}
SetOnContextMenu(f) { …
}
SetOnDragStart(f)     { …
}
SetOnDragEnd(f)       { …
}
SetOnDrop(f)          { …
}
SetOnDragEnter(f)     { …
}
SetOnDragOver(f)      { …
}
SetOnDragLeave(f)     { …
}
```

***Figure 41**: The Block Base class, the API it provides and the methods it requires from subclasses*

The *Block* base class (Figure 41) provides a collection of fields and methods that can be used by the syntax-directed editor and its commands, as well as any other component that requires block functionality. Every block category should inherit from the block base class and provide implementation for its pure virtual methods. For the purposes of this thesis we will discuss the primary fields and methods, as well as the requirements that must be met by subclasses.

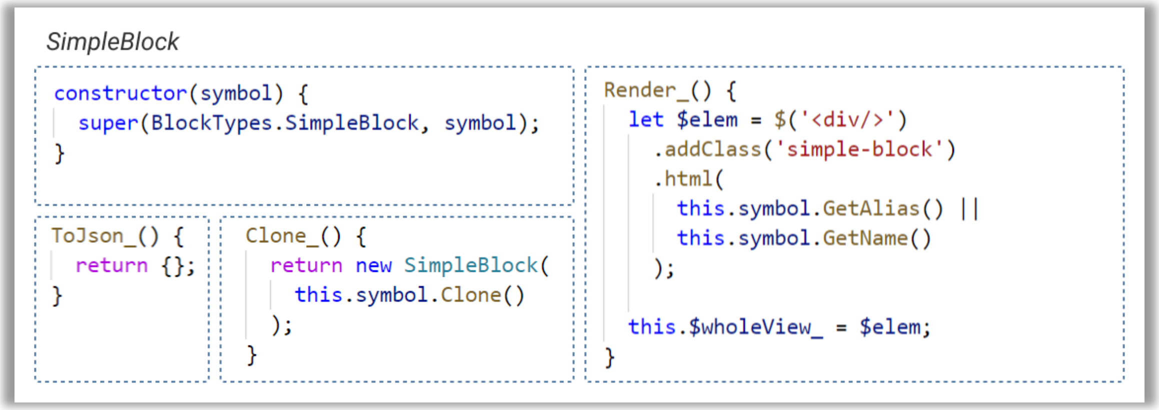
In order to construct a valid block, the creator needs to provide a *type* string, which can be later accessed to invoke methods that are specific to a subclass, as well as a *symbol* object, which stores data as provided in the language grammar specification. Particularly, the symbol field contains data for the grammar symbol's type (terminal or non-terminal), as well as its name, alias and tooltip. The *generatedBy* field is used by the editor to hold predecessor blocks in a linked list manner. Predecessor blocks are previous selection blocks, which were expanded through syntax-directed insertions to generate the current block. For instance a binary-arithmetic-expression block could have a generatedBy arithmetic-expression block which could have a generatedBy expression block. This information is used in editing operations such as deletion, in order to retrieve and re-render previous blocks in the expansion sequence. The *parent* field can be accessed by the editor in order to retrieve the group block which is one level lower in the visual code AST. For instance the parent field of a "condition" block could be an "if-statement" block. The *$wholeView* field contains the DOM element which corresponds to the block. The *isDraggable* and *isDroppable* fields are used by components to control the drag-and-drop block interaction. For instance isDroppable is set to false for blocks in the Code-Chips toolbox, to prevent replacing them by dropping other blocks onto them.

In the context of rendering, the Block base class provides four primary methods: *Render*, *RenderBeofre*, *RenderAfter* and *RemoveRenderedView*. These methods allow the editor to render the block inside a container, before or after another DOM element, or completely remove its rendered view respectively. In order to achieve this, each subclass provides a *Render_* method that constructs and saves the corresponding DOM element in the $wholeView field without attaching it to the page. The rendering methods, then, append the block to the document and apply the appropriate event handlers, which were attached by the editor, to the rendered view. Finally, the rendering methods invoke *PastRendering_*, optionally provided by subclasses, in order to handle tasks that cannot be completed before the block is a part of the page.

It is essential for the Block base class to support functionality for cloning and exporting to JSON or string, as this facilitates features such as exporting the whole visual code AST to JSON format, or copying-and-pasting blocks onto other blocks. In order to achieve conversion into JSON, the base class explicitly handles its data, which are shared by all block subclasses and requires from each subclass to implement a *ToJson_* method that exports data specific to the subtype. Similarly, for cloning, subclasses provide a *Clone_* method, through which the base class obtains a clone, to which it attaches event handlers.

The methods *ToJsonRec* and *CloneRec* recursively apply *ToJson* or *Clone* for each of the generatedBy blocks.

4.3.2 Current Block Collection


SimpleBlock

```
constructor(symbol) {
    super(BlockTypes.SimpleBlock, symbol);
}
```

```
ToJson_() {
    return {};
}
```

```
Clone_() {
    return new SimpleBlock(
        this.symbol.Clone()
    );
}
```

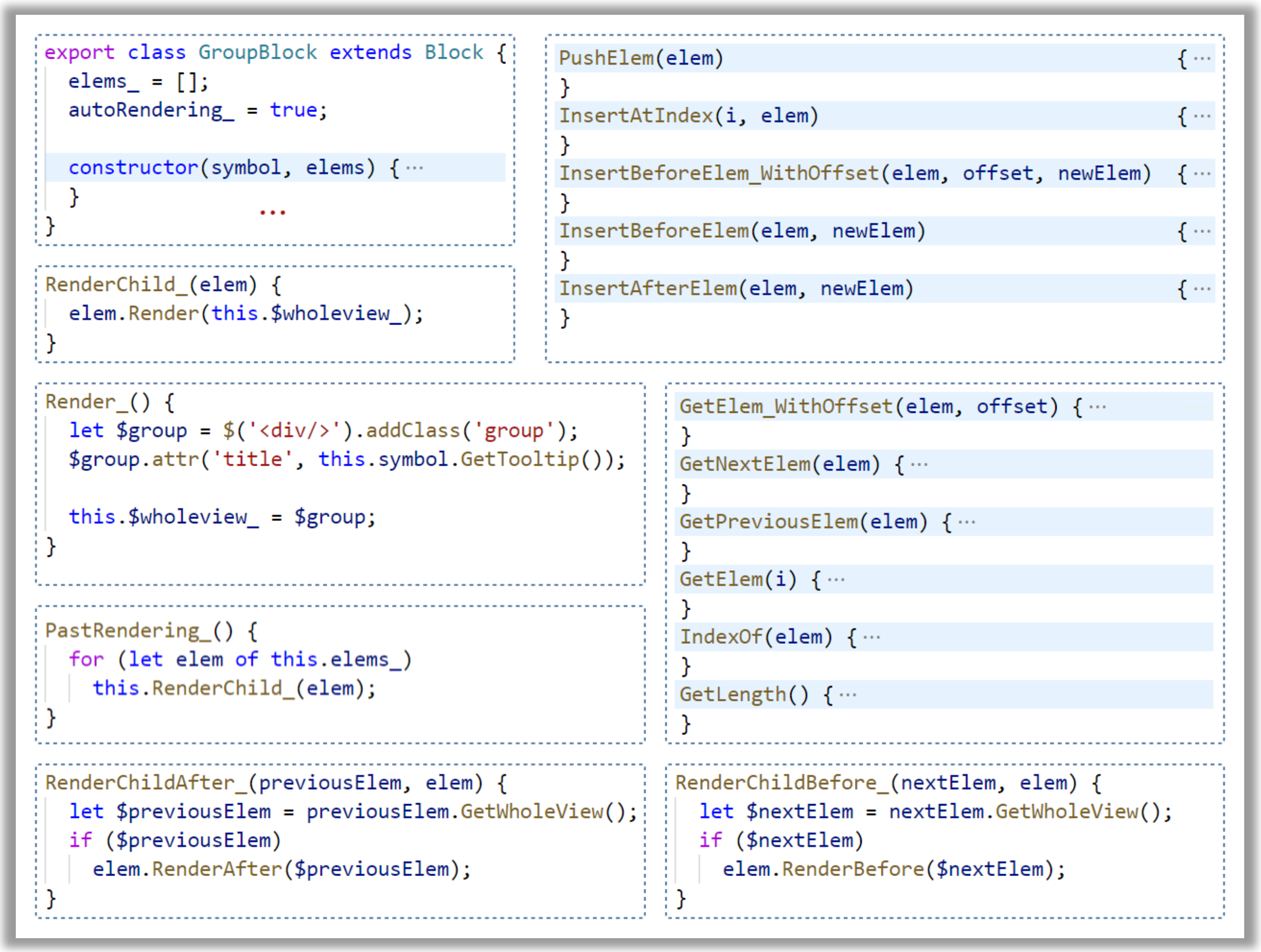
```
Render_() {
    let $elem = $('<div/>')
        .addClass('simple-block')
        .html(
            this.symbol.GetAlias() ||
            this.symbol.GetName()
        );

    this.$wholeView_ = $elem;
}
```

Figure 42*: The core elements of the SimpleBlock class*

The current block collection includes classes for simple blocks, selection blocks, input blocks, optional blocks, group blocks, repetition group blocks, new-line blocks and tab blocks. Figure 42 depicts the core of the most trivial of these: the *SimpleBlock* class. As previously stated, the simple block category simulates language grammar terminal symbols such as the "if" keyword or the "=" operator. In its core, it only provides implementation for ToJson_, Clone_ and Render_ which are required base class methods. In its ToJson_ method, it returns an empty object since it does not use any extra data. For its rendering, it creates a div DOM element, attaches the class "simple-block" to it and sets its text to the symbol's alias, if the language author provided one, otherwise sets it to the symbol's name.

At their core, the *SelectionBlock* and *InputBlock* classes are simple but unlike SimpleBlock, provide interactive rendered views. As a result, they provide methods for setting their interaction event handlers. In particular, a selection block is constructed given a symbol which represents the left-hand-side of a production and a symbol array which represents its alternate right-hand-side expansion choices. For decoupling reasons, selection blocks do not have embedded logic for creating and inserting the new block upon user selection. Instead they accept an event handler, called *onSelection*, and are responsible for invoking it when the user selects one of the available alternate options for expansion.

Similarly, input blocks accept an *onInput* event handler which they are responsible to invoke each time the user types a character. The editor uses this to validate the input, based on the symbol's type (e.g. integer, string etc) and provide visual feedback.



```
export class GroupBlock extends Block {
  elems_ = [];
  autoRendering_ = true;

  constructor(symbol, elems) {…
  }
                ...
}
```

```
RenderChild_(elem) {
  elem.Render(this.$wholeview_);
}
```

```
PushElem(elem)                                    {…
}
InsertAtIndex(i, elem)                            {…
}
InsertBeforeElem_WithOffset(elem, offset, newElem)  {…
}
InsertBeforeElem(elem, newElem)                   {…
}
InsertAfterElem(elem, newElem)                    {…
}
```

```
Render_() {
  let $group = $('<div/>').addClass('group');
  $group.attr('title', this.symbol.GetTooltip());

  this.$wholeview_ = $group;
}
```

```
GetElem_WithOffset(elem, offset) {…
}
GetNextElem(elem) {…
}
GetPreviousElem(elem) {…
}
GetElem(i) {…
}
IndexOf(elem) {…
}
GetLength() {…
}
```

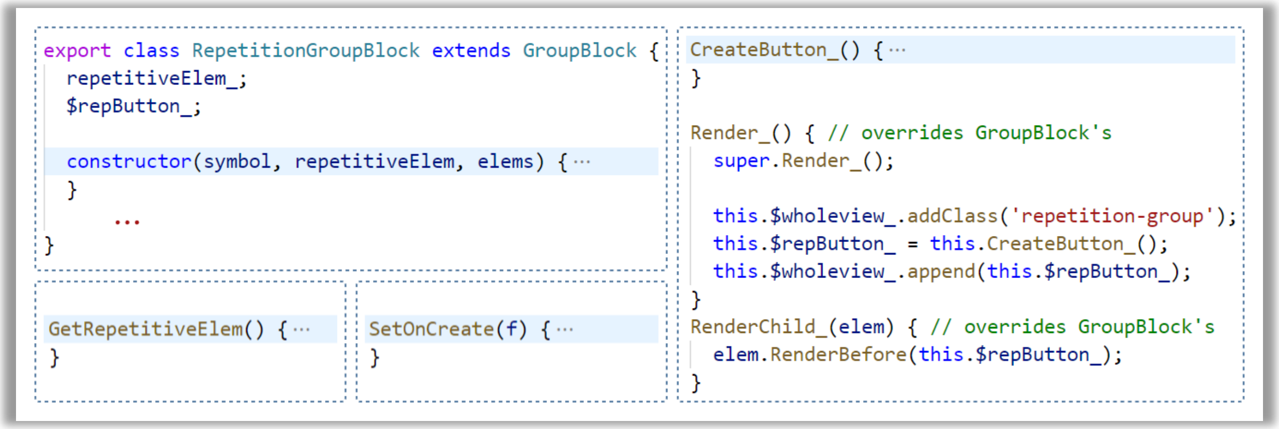
```
PastRendering_() {
  for (let elem of this.elems_)
    this.RenderChild_(elem);
}
```

```
RenderChildAfter_(previousElem, elem) {
  let $previousElem = previousElem.GetWholeView();
  if ($previousElem)
    elem.RenderAfter($previousElem);
}
```

```
RenderChildBefore_(nextElem, elem) {
  let $nextElem = nextElem.GetWholeView();
  if ($nextElem)
    elem.RenderBefore($nextElem);
}
```

Figure 43: *The GroupBlock class, along with its primary API and rendering implementation details*

The *GroupBlock* class Figure 43 emulates grammar non-terminal symbols with a production right-hand-side that contains more than one symbol. For instance, an "if-statement" group block consists of inner blocks for an "if" symbol, an "expression" symbol and a "statements" symbol. The *elems_* field is an array that holds blocks of any category. The GroupBlock class provides methods for manipulating this array, which the editor uses in its various editing commands. The *autoRendering* field denotes whether the GroupBlock automatically updates its rendered view, when manipulating the array with one of the methods it provides. Of course, this does not require producing all the group DOM elements, but builds on the existing rendered view using the methods *RenderChildAfter_* or *RenderChildBefore_*. Figure 43, also depicts the rendering sequence for a GroupBlock in which a div container is created and attached to the page, followed by rendering the inner blocks by invoking Render of the Block base class.



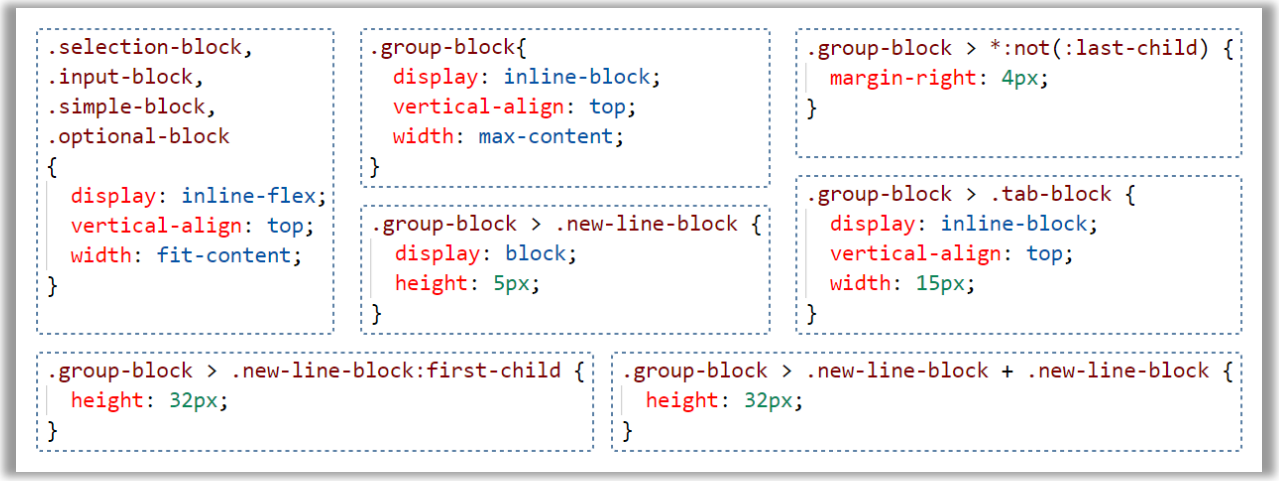
```
export class RepetitionGroupBlock extends GroupBlock {
  repetitiveElem_;
  $repButton_;

  constructor(symbol, repetitiveElem, elems) {···
  }
     ...
}
```

```
GetRepetitiveElem() {···
}
```

```
SetOnCreate(f) {···
}
```

```
CreateButton_() {···
}

Render_() { // overrides GroupBlock's
  super.Render_();

  this.$wholeview_.addClass('repetition-group');
  this.$repButton_ = this.CreateButton_();
  this.$wholeview_.append(this.$repButton_);
}
RenderChild_(elem) { // overrides GroupBlock's
  elem.RenderBefore(this.$repButton_);
}
```

***Figure 44**: The RepetitionGroup class' basic API and rendering based on GroupBlock*



```
.selection-block,
.input-block,
.simple-block,
.optional-block
{
  display: inline-flex;
  vertical-align: top;
  width: fit-content;
}
```

```
.group-block{
  display: inline-block;
  vertical-align: top;
  width: max-content;
}
```

```
.group-block > .new-line-block {
  display: block;
  height: 5px;
}
```

```
.group-block > *:not(:last-child) {
  margin-right: 4px;
}
```

```
.group-block > .tab-block {
  display: inline-block;
  vertical-align: top;
  width: 15px;
}
```

```
.group-block > .new-line-block:first-child {
  height: 32px;
}
```

```
.group-block > .new-line-block + .new-line-block {
  height: 32px;
}
```

***Figure 45**: CSS related to the system's row-based indented layout*

RepetitionGroupBlock (Figure 44) extends the GroupBlock class with the addition of a repetitive element. This repetitive element, *repetitiveElem_*, is generated with every press of the button *$repButton_*. Similarly to selection blocks and input blocks, event handling is decoupled from the class and delegated to the syntax-directed editor, with the method *SetOnCreate*. In terms of rendering, repetition group blocks use the infrastructure of group blocks, but also add a "repetition-group" CSS class, append a button to the group's rendered view and override the method *RenderChild_*, in order to render the elements at the appropriate position.

Group blocks, and thus repetition group blocks, are partly responsible for the Code-Chips row-based layout and indentation, as they are able to host new-line blocks and tab blocks. Particularly, group blocks are designed to offset their inner elements by a margin, and due to their inline display, they are rendered in a horizontal manner. When a group block hosts a new-line block, the new-line block occupies a whole horizontal line by itself and ensures the following elements are rendered below it, using an appropriate CSS property. When two new-lines are rendered in a row, the last new-line occupies more vertical space in order to simulate an empty line, similarly to a text-editor line. Tab blocks are simply DOM elements with a fixed width and inline display, so that they offset the elements that follow them. Figure 45 depicts the CSS code that is primarily responsible for rendering according to the system's row-based indented layout.

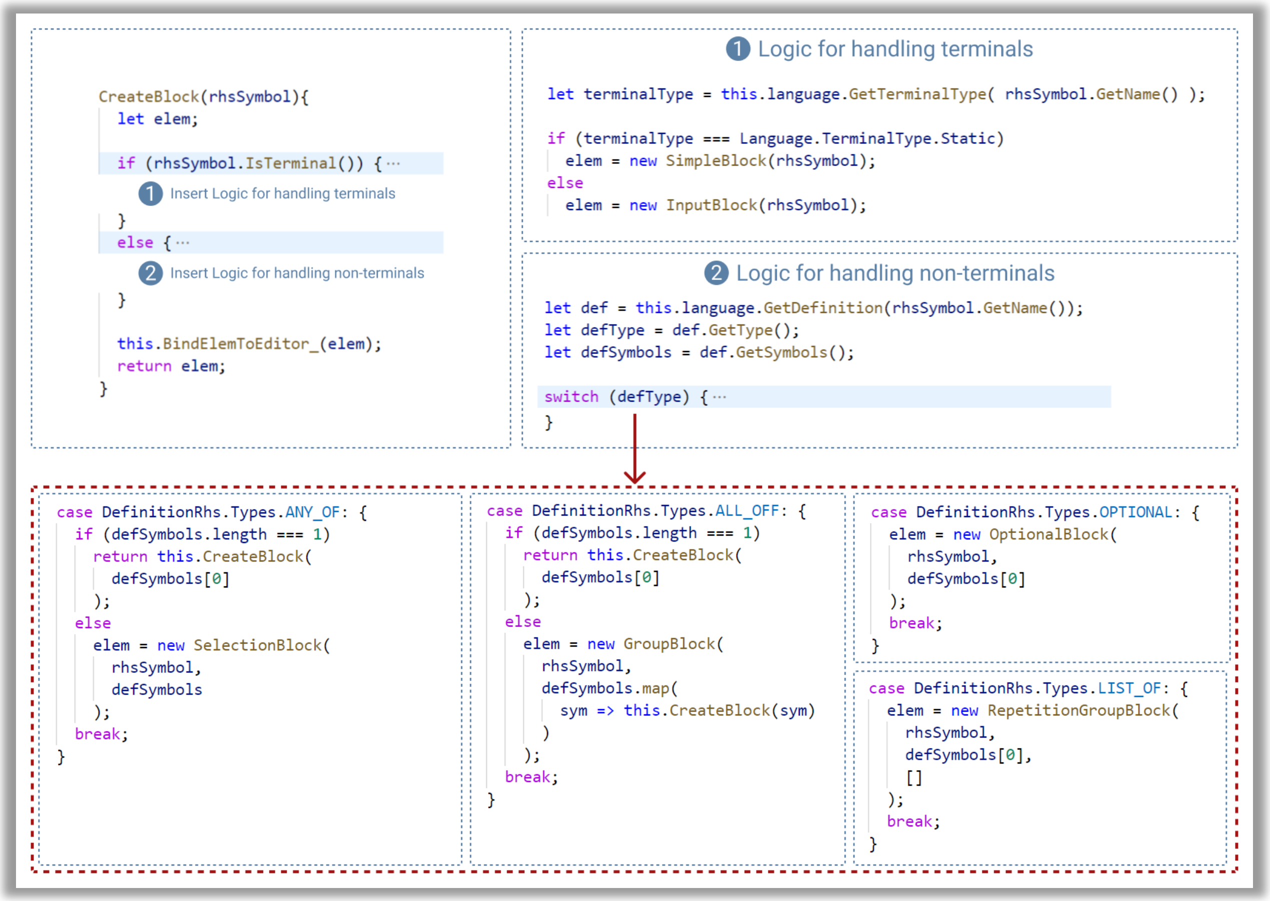
4.4 Editor Operations

In this section we will discuss the infrastructure behind editing commands, designed to allow undo-and-redo functionality and we will provide appropriate code segments and examples. Before that, having discussed our approach's meta language grammar and how it is converted to JSON format, as well as the available blocks in our system and their implementation, we will analyze the process of converting the input grammar definitions to interactive blocks.

4.4.1 Converting Grammar to Blocks

As we have previously stated, our system's syntax-directed editor accepts the input language grammar specification in JSON form, as it is output by the parser. In order to allow easier manipulation and access, the editor converts the JSON object, output by the parser, into an instance of the *Language* class, which provides a simple API, mainly for registering and accessing symbols and their productions. Figure 46 provides an overview of the algorithm for converting symbols into blocks. Non-terminal grammar symbols can produce either simple blocks for static terminals such as the "if" keyword and the "+"

operator, or input blocks for dynamic terminals such as an integer or a string. Non-terminal grammar symbols can produce group blocks, repetition group blocks, selection blocks and optional blocks, according to the item type used by the language author on the grammar specification. The aforementioned block types correspond to the meta language tokens “all_of”, “any_of”, “list_of” and “optional” respectively. In the special case of having a non-terminal grammar symbol that is of type “all_of” or “any_of” with a production right-hand-side that consists of only one grammar symbol, the editor can skip the symbol and proceed with generating a block for its production right-hand-side symbol. In the end of the block creation process, the editor must bind the created block by setting generic event handlers such as opening the context-menu when a block is right-clicked or highlighting the block when it is clicked as well as block type specific event handlers, such as syntax-directed insertion when a selection block option is clicked or input validation when the user types a character in an input block.


1 Insert Logic for handling terminals
2 Insert Logic for handling non-terminals
1 Logic for handling terminals
2 Logic for handling non-terminals

```
CreateBlock(rhsSymbol){
  let elem;

  if (rhsSymbol.IsTerminal()) {…
      1 Insert Logic for handling terminals
  }
  else {…
      2 Insert Logic for handling non-terminals
  }

  this.BindElemToEditor_(elem);
  return elem;
}
```

```
let terminalType = this.language.GetTerminalType( rhsSymbol.GetName() );

if (terminalType === Language.TerminalType.Static)
  elem = new SimpleBlock(rhsSymbol);
else
  elem = new InputBlock(rhsSymbol);
```

```
let def = this.language.GetDefinition(rhsSymbol.GetName());
let defType = def.GetType();
let defSymbols = def.GetSymbols();

switch (defType) {…
}
```

```
case DefinitionRhs.Types.ANY_OF: {
  if (defSymbols.length === 1)
    return this.CreateBlock(
      defSymbols[0]
    );
  else
    elem = new SelectionBlock(
      rhsSymbol,
      defSymbols
    );
  break;
}
```

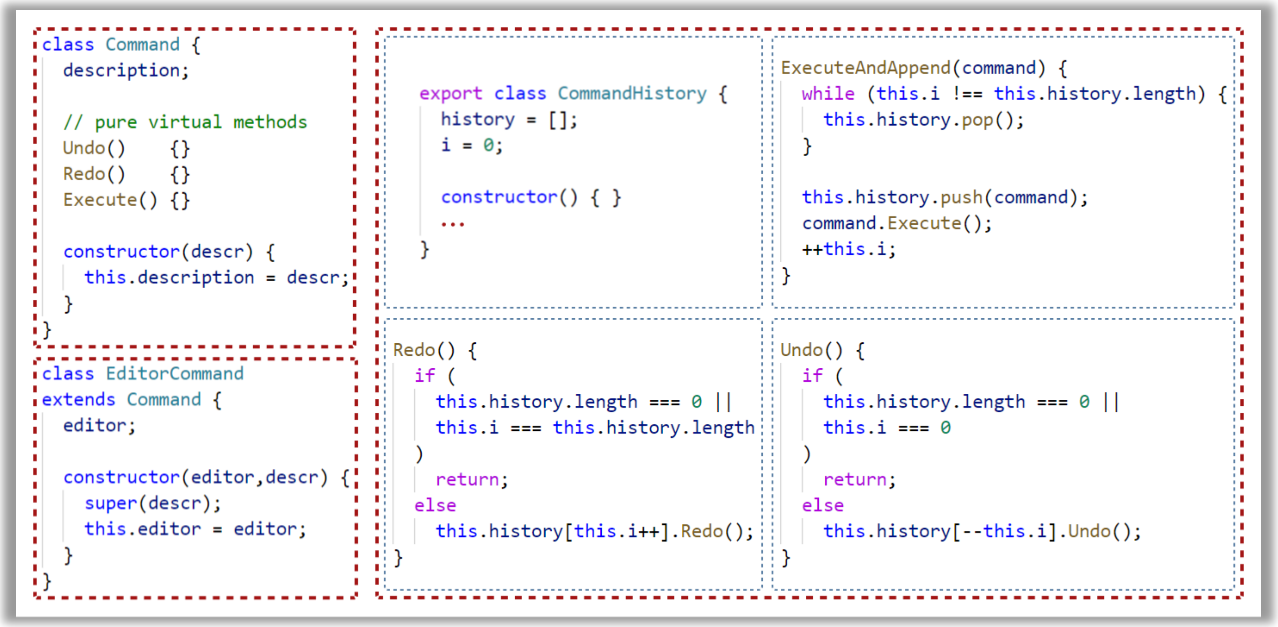
```
case DefinitionRhs.Types.ALL_OFF: {
  if (defSymbols.length === 1)
    return this.CreateBlock(
      defSymbols[0]
    );
  else
    elem = new GroupBlock(
      rhsSymbol,
      defSymbols.map(
        sym => this.CreateBlock(sym)
      )
    );
  break;
}
```

```
case DefinitionRhs.Types.OPTIONAL: {
  elem = new OptionalBlock(
    rhsSymbol,
    defSymbols[0]
  );
  break;
}
```

```
case DefinitionRhs.Types.LIST_OF: {
  elem = new RepetitionGroupBlock(
    rhsSymbol,
    defSymbols[0],
    []
  );
  break;
}
```

***Figure 46**: Primary logic for block creation*

4.4.2 Editing Commands and Undo-Redo



```
class Command {
  description;

  // pure virtual methods
  Undo()    {}
  Redo()    {}
  Execute() {}

  constructor(descr) {
    this.description = descr;
  }
}
```

```
class EditorCommand
extends Command {
  editor;

  constructor(editor,descr) {
    super(descr);
    this.editor = editor;
  }
}
```

```
export class CommandHistory {
  history = [];
  i = 0;

  constructor() { }
  ...
}
```

```
ExecuteAndAppend(command) {
  while (this.i !== this.history.length) {
    this.history.pop();
  }

  this.history.push(command);
  command.Execute();
  ++this.i;
}
```

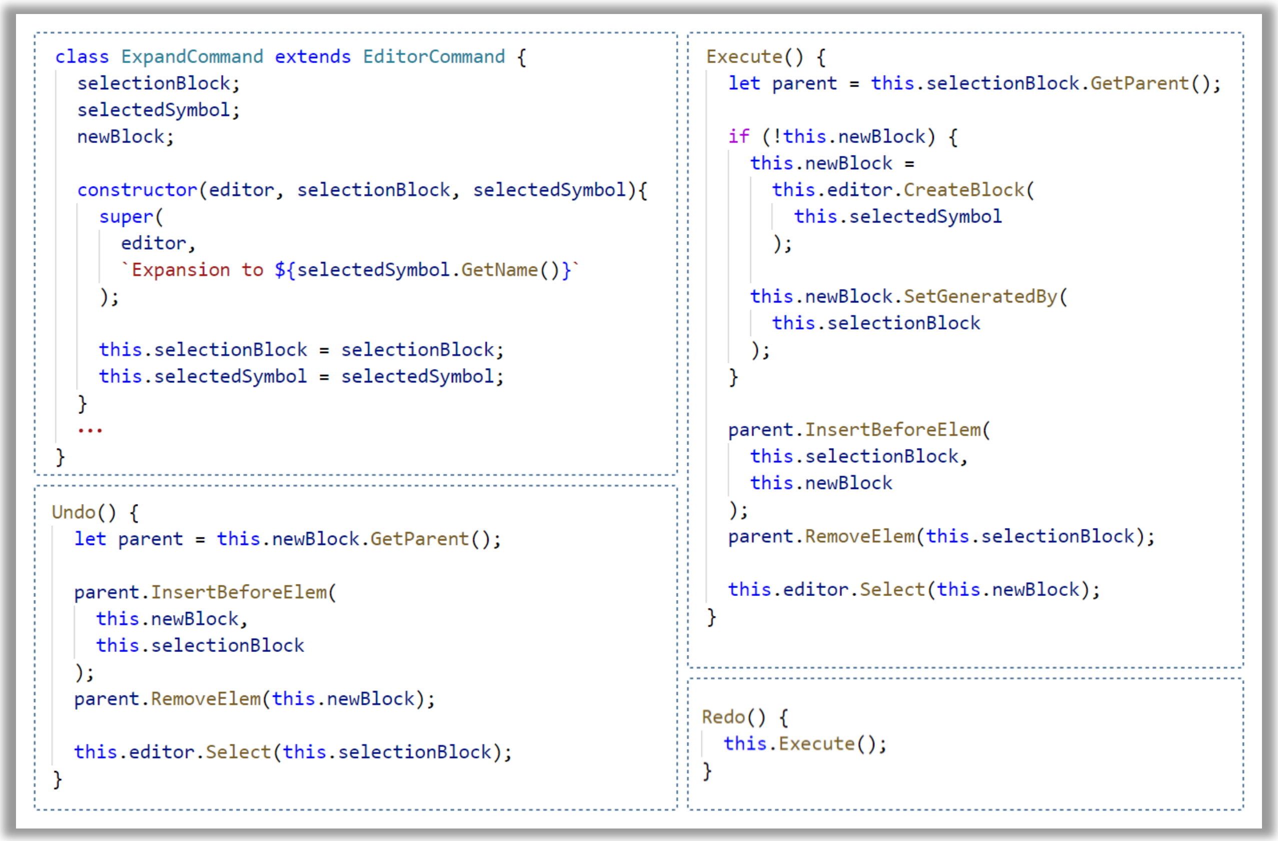
```
Redo() {
  if (
    this.history.length === 0 ||
    this.i === this.history.length
  )
    return;
  else
    this.history[this.i++].Redo();
}
```

```
Undo() {
  if (
    this.history.length === 0 ||
    this.i === 0
  )
    return;
  else
    this.history[--this.i].Undo();
}
```

Figure 47*: The Command base class, the EditorCommand class and the CommandHistory class*

Code-Chips, as other visual programming and syntax-directed editors, provides undo-and-redo functionality for its editing commands. In order to accomplish this, we have used the command design pattern. Every editor command derives from the class *EditorCommand*, which itself derives from the *Command* base class (left part of Figure 47). Every editor command must implement the three pure virtual methods of the Command base class - *Undo*, *Redo* and *Execute* – and thus provide functionality for undoing and redoing itself.

In order to keep track of the commands that are available for undoing and redoing, the syntax-directed editor has an instance of the *CommandHistory* class (right part of Figure 47), which uses an array of commands, named “history”, and a position in the array, named “i”. Commands that are available for undoing are positioned to the left of “i” with it always pointing at the index of the leftmost command available for redoing. As a result, “i” is decreased when undoing and increased when redoing. When a new command is executed through *ExecuteAndAppend*, the available for redoing commands are removed from the array and the new command is inserted as the rightmost command, meaning it is the most recent command and the first one available for undoing.



```
class ExpandCommand extends EditorCommand {
  selectionBlock;
  selectedSymbol;
  newBlock;

  constructor(editor, selectionBlock, selectedSymbol){
    super(
      editor,
      `Expansion to ${selectedSymbol.GetName()}`
    );

    this.selectionBlock = selectionBlock;
    this.selectedSymbol = selectedSymbol;
  }
  ...
}
```

```
Undo() {
  let parent = this.newBlock.GetParent();

  parent.InsertBeforeElem(
    this.newBlock,
    this.selectionBlock
  );
  parent.RemoveElem(this.newBlock);

  this.editor.Select(this.selectionBlock);
}
```

```
Execute() {
  let parent = this.selectionBlock.GetParent();

  if (!this.newBlock) {
    this.newBlock =
      this.editor.CreateBlock(
        this.selectedSymbol
      );

    this.newBlock.SetGeneratedBy(
      this.selectionBlock
    );
  }

  parent.InsertBeforeElem(
    this.selectionBlock,
    this.newBlock
  );
  parent.RemoveElem(this.selectionBlock);

  this.editor.Select(this.newBlock);
}
```

```
Redo() {
  this.Execute();
}
```

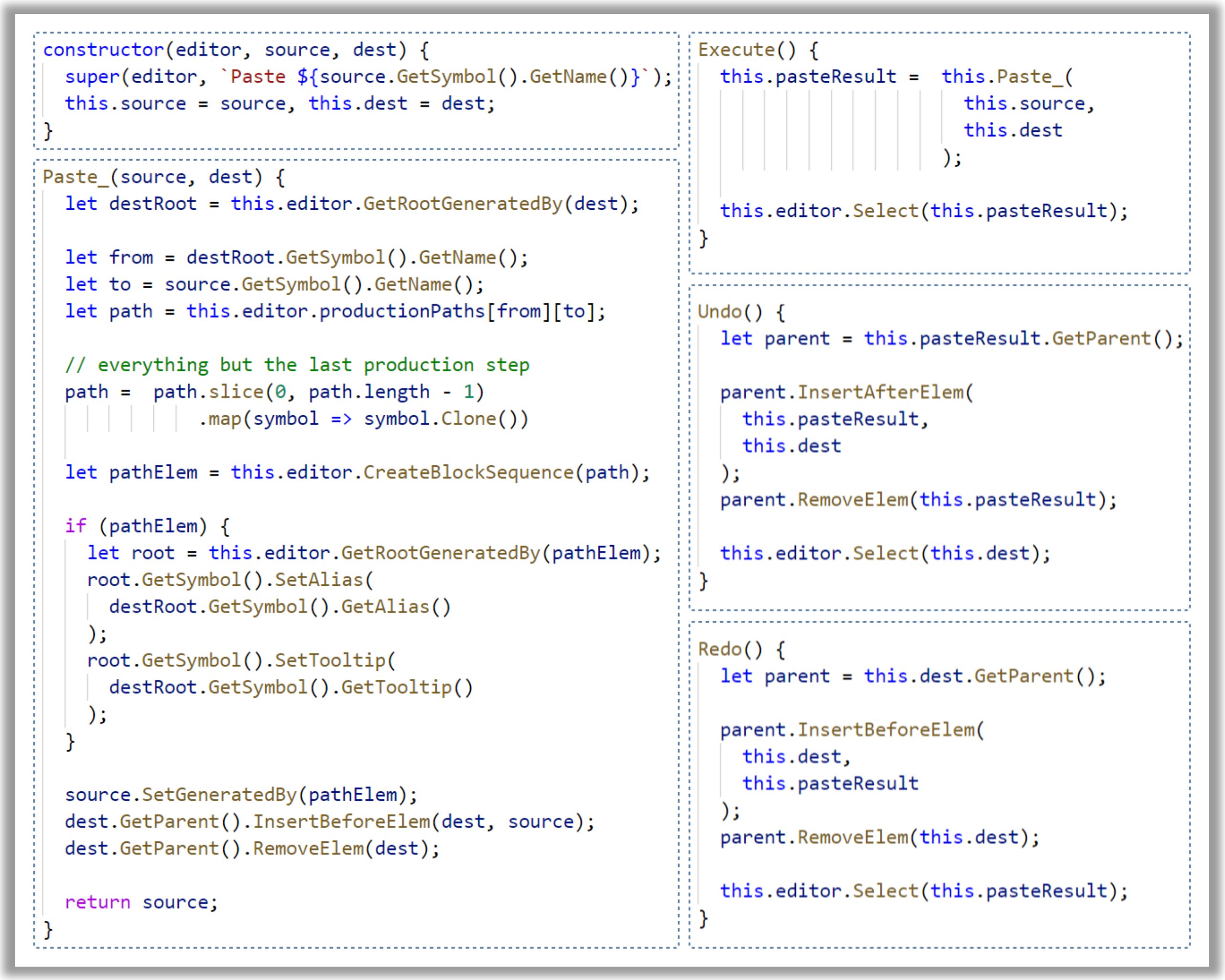
***Figure 48**: The ExpandCommand class*

For a better understanding of the mechanism of editing commands and visual code manipulation using the group block API, let us look at the implementation of the primary operation of expanding a selection block into the production right-hand-side symbol chosen by the user. In order to expand a symbol, the user clicks on a selection block, which opens the selection block's dropdown menu with all the possible options. The user action of selecting one of the options triggers the selection block's onSelect event handler, which was registered by the syntax-directed editor. As a result, a new instance of the *ExpandCommand* class is created and used as an argument to invoke the ExecuteAndAppend method of CommandHistory and execute the command.

The ExpandCommand constructor accepts, as its arguments, the editor, the selection block that caused the event and the symbol chosen by the user. In its Execute method, ExpandCommand creates a new block using the previously analyzed *CreateBlock* editor method and sets its generatedBy field to the selection block which caused the expansion. Afterwards, using the group block API, it replaces the selection block with the new block by inserting it to the place of the selection block and then removing the selection block. Finally, it invokes the editor method *Select*, which selects the block and highlights it with a yellow border to visually indicate that it is selected. Note that there is no need for the editor to re-render the whole group block as group blocks automatically update their rendered view on insertion and deletion.

The method Undo reverses the above insertion and deletion sequence, meaning that it re-inserts the selection block, removes the new block and highlights the selection block. The method Redo simply invokes Execute to repeat the aforementioned steps of insertion and deletion. An important observation is that Execute saves the newly created block as a class field, in order to be reused in subsequent calls. This is not done purely for optimization reasons; it is required for proper undo-and-redo functionality, as commands such as this use references to blocks. If the block was re-created upon redoing the command, then following invocations of Redo for other command instances that use the originally created block would result in invalid access to memory and improper execution.

As a second and more interesting example, let us discuss the implementation for syntactic copy-and-paste (Figure 49). We concentrate on pasting, as copying can be handled by simply invoking the Block base class method CloneRec of the selected block and saving the result to an editor field. As stated previously, pasting is not trivial due to maintaining syntactic compliance relative to the language grammar and thus producing no syntax-errors. Code-Chips allows syntactic copy-and-paste when the root symbol of the destination block can expand to the symbol of the copied block, meaning that there is a chain of productions (production path), starting from the destination block's initial placeholder, that produces the copied block. In order to answer this question, the system precomputes a mapping of each symbol valid copy-paste pair *<i, j>* to the production path that leads from *i* to *j*. This is implemented by initiating a depth-first search from each non-terminal grammar symbol and saving the current path for every node that is reachable by the process of following its productions. These data is held by the editor in a field named *productionPaths* and is used by commands such as the *PasteCommand* and the *DropCommand*.



```
constructor(editor, source, dest) {
  super(editor, `Paste ${source.GetSymbol().GetName()}`);
  this.source = source, this.dest = dest;
}

Paste_(source, dest) {
  let destRoot = this.editor.GetRootGeneratedBy(dest);

  let from = destRoot.GetSymbol().GetName();
  let to = source.GetSymbol().GetName();
  let path = this.editor.productionPaths[from][to];

  // everything but the last production step
  path =  path.slice(0, path.length - 1)
              .map(symbol => symbol.Clone())

  let pathElem = this.editor.CreateBlockSequence(path);

  if (pathElem) {
    let root = this.editor.GetRootGeneratedBy(pathElem);
    root.GetSymbol().SetAlias(
      destRoot.GetSymbol().GetAlias()
    );
    root.GetSymbol().SetTooltip(
      destRoot.GetSymbol().GetTooltip()
    );
  }

  source.SetGeneratedBy(pathElem);
  dest.GetParent().InsertBeforeElem(dest, source);
  dest.GetParent().RemoveElem(dest);

  return source;
}
```

```
Execute() {
  this.pasteResult =  this.Paste_(
                        this.source,
                        this.dest
                      );

  this.editor.Select(this.pasteResult);
}
```

```
Undo() {
  let parent = this.pasteResult.GetParent();

  parent.InsertAfterElem(
    this.pasteResult,
    this.dest
  );
  parent.RemoveElem(this.pasteResult);

  this.editor.Select(this.dest);
}
```

```
Redo() {
  let parent = this.dest.GetParent();

  parent.InsertBeforeElem(
    this.dest,
    this.pasteResult
  );
  parent.RemoveElem(this.dest);

  this.editor.Select(this.pasteResult);
}
```

Figure 49*: The PasteCommand class' core implementation*

Particularly, PasteCommand requires the editor, the copied block, called *source*, and the destination block, called *dest*. Its core functionality is performed by the method *Paste_*, which initially uses the aforementioned editor field productionPaths to acquire the chain of expansions that produces the source grammar symbol, when expanding starts from the destination grammar symbol. Note that the syntax-directed editor has already ensured that pasting is possible before constructing an instance of PasteCommand and thus checking for existence is not required. After acquiring the production path, a block is generated, using the editor method *CreateBlockSequence*. The last expansion step is excluded from the generation, as the command can use the source block instead. CreateBlockSequence simply invokes CreateBlock and connects the blocks in a simply-linked list using their SetGeneratedBy method. A detail is that, due to having aliases and tooltips only on right-hand-side symbols of the language, the root of the produced chain does not have a proper alias and tooltip. For this purpose, the appropriate alias and tooltip are copied from the root of the original destination block. Finally, the source block is connected to the produced block and replaces the destination block in the AST.

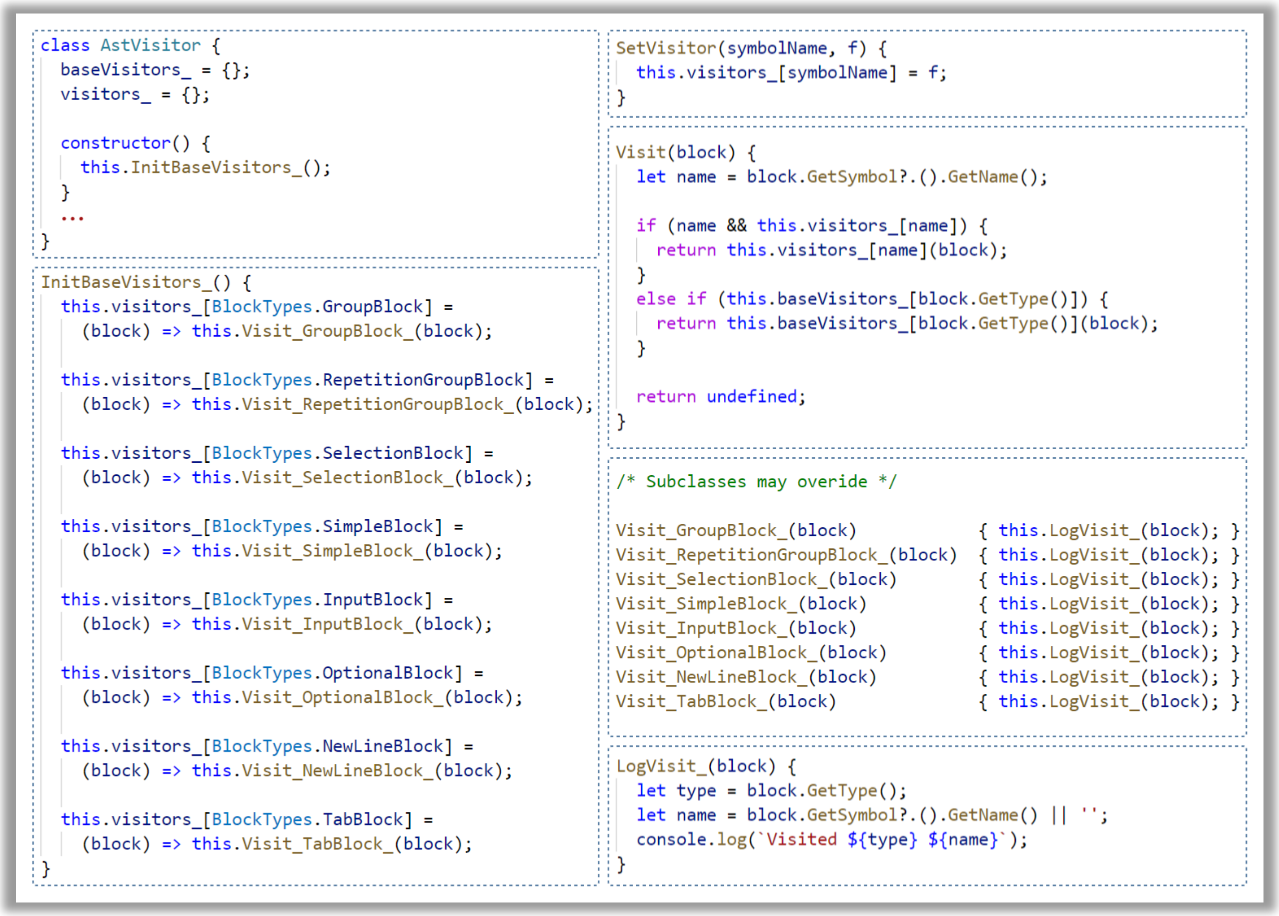
Similarly to ExpandCommand, PasteCommand marks the new block as selected by invoking the editor method Select, and saves the new block for reusing. Using the field *pasteResult*, the Undo and Redo methods are trivial and similar to the ones of ExpandCommand.

4.5 Runtime Support

In this section we will discuss how the infrastructure of our system can be utilized by the language author, in order to create a runtime environment for the language they have developed. In particular, we will analyze two classes, *AstHost* and *AstVisitor*, which can be used in combination to traverse the block AST, convert it to JavaScript and execute it.

Of course, conversion to JavaScript is specific to the input language grammar and its semantics, so providing a global runtime environment is not possible. Instead, our system currently provides the AstVisitor class and the AstHost class. The AstVisitor class (Figure 50) provides functionality for registering visitors per block category, such as simple blocks and group blocks, and per symbol type, such as expression or statement. When the method *Visit* is invoked with a block as its argument, it invokes the handler which is registered for the specific grammar symbol or the appropriate generic block category handler if the former is not provided. These handlers accept blocks as their arguments, meaning the language author has access to the entire block API.

The AstHost class (Figure 51) accepts a visitor instance as its constructor argument and is responsible for traversing through the block AST and invoking the visitor's Visit method in the correct order. The AstHost of Figure 51 traverses the AST and invokes Visit in post-order traversal, meaning a group block will be visited after all its children-blocks are visited in an order from the left-most to the right-most.



```
class AstVisitor {
  baseVisitors_ = {};
  visitors_ = {};

  constructor() {
    this.InitBaseVisitors_();
  }
  ...
}
```

```
InitBaseVisitors_() {
  this.visitors_[BlockTypes.GroupBlock] =
    (block) => this.Visit_GroupBlock_(block);

  this.visitors_[BlockTypes.RepetitionGroupBlock] =
    (block) => this.Visit_RepetitionGroupBlock_(block);

  this.visitors_[BlockTypes.SelectionBlock] =
    (block) => this.Visit_SelectionBlock_(block);

  this.visitors_[BlockTypes.SimpleBlock] =
    (block) => this.Visit_SimpleBlock_(block);

  this.visitors_[BlockTypes.InputBlock] =
    (block) => this.Visit_InputBlock_(block);

  this.visitors_[BlockTypes.OptionalBlock] =
    (block) => this.Visit_OptionalBlock_(block);

  this.visitors_[BlockTypes.NewLineBlock] =
    (block) => this.Visit_NewLineBlock_(block);

  this.visitors_[BlockTypes.TabBlock] =
    (block) => this.Visit_TabBlock_(block);
}
```

```
SetVisitor(symbolName, f) {
  this.visitors_[symbolName] = f;
}
```

```
Visit(block) {
  let name = block.GetSymbol?.().GetName();

  if (name && this.visitors_[name]) {
    return this.visitors_[name](block);
  }
  else if (this.baseVisitors_[block.GetType()]) {
    return this.baseVisitors_[block.GetType()](block);
  }

  return undefined;
}
```

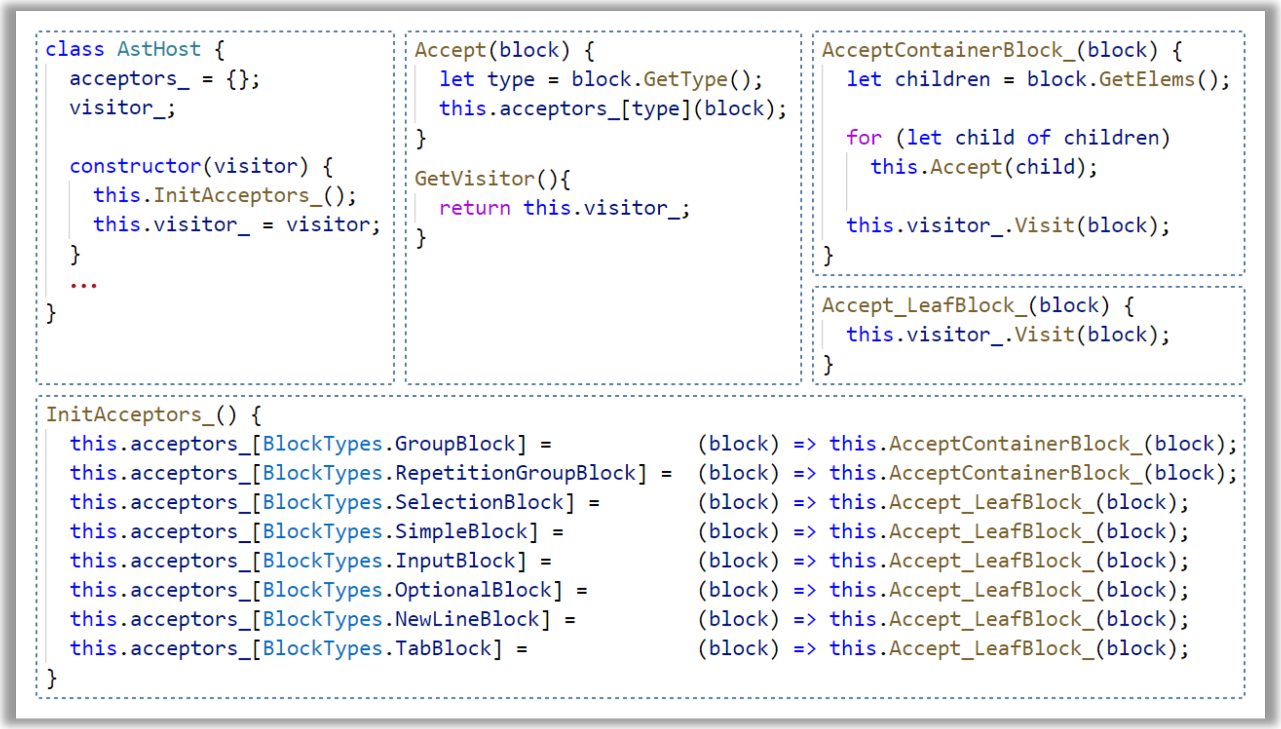
```
/* Subclasses may overide */

Visit_GroupBlock_(block)             { this.LogVisit_(block); }
Visit_RepetitionGroupBlock_(block)   { this.LogVisit_(block); }
Visit_SelectionBlock_(block)         { this.LogVisit_(block); }
Visit_SimpleBlock_(block)            { this.LogVisit_(block); }
Visit_InputBlock_(block)             { this.LogVisit_(block); }
Visit_OptionalBlock_(block)          { this.LogVisit_(block); }
Visit_NewLineBlock_(block)           { this.LogVisit_(block); }
Visit_TabBlock_(block)               { this.LogVisit_(block); }
```

```
LogVisit_(block) {
  let type = block.GetType();
  let name = block.GetSymbol?.().GetName() || '';
  console.log(`Visited ${type} ${name}`);
}
```

***Figure 50**: The AstVisitor class' basic functionality*

The language author or a different programmer can utilize this infrastructure by implementing a subclass of the visitor class and defining the language specific handlers for each grammar symbol. For instance, a language author can provide a visitor subclass that converts the block AST to JavaScript by gradually converting each block, when possible, storing the results and connecting them in an appropriate way when visiting group blocks. After obtaining JavaScript source-code, execution is easily handled by using JavaScript's *eval* function. This is the approach that we have chosen when implementing the runtime environment for the example language of Chapter 5. Section 5.3 provides details about its implementation, including the handling of indentation and parenthesization for producing readable source-code.

Although the system provides the aforementioned infrastructure, the editor can accept any handler for JavaScript conversion and visual code execution. In this context, language authors are free to explore other options, such as implementing a custom interpreter and not explicitly converting to JavaScript for execution.



```
class AstHost {
  acceptors_ = {};
  visitor_;

  constructor(visitor) {
    this.InitAcceptors_();
    this.visitor_ = visitor;
  }
  ...
}
```

```
Accept(block) {
  let type = block.GetType();
  this.acceptors_[type](block);
}
GetVisitor(){
  return this.visitor_;
}
```

```
AcceptContainerBlock_(block) {
  let children = block.GetElems();

  for (let child of children)
    this.Accept(child);

  this.visitor_.Visit(block);
}
```

```
Accept_LeafBlock_(block) {
  this.visitor_.Visit(block);
}
```

```
InitAcceptors_() {
  this.acceptors_[BlockTypes.GroupBlock] =           (block) => this.AcceptContainerBlock_(block);
  this.acceptors_[BlockTypes.RepetitionGroupBlock] = (block) => this.AcceptContainerBlock_(block);
  this.acceptors_[BlockTypes.SelectionBlock] =       (block) => this.Accept_LeafBlock_(block);
  this.acceptors_[BlockTypes.SimpleBlock] =          (block) => this.Accept_LeafBlock_(block);
  this.acceptors_[BlockTypes.InputBlock] =           (block) => this.Accept_LeafBlock_(block);
  this.acceptors_[BlockTypes.OptionalBlock] =        (block) => this.Accept_LeafBlock_(block);
  this.acceptors_[BlockTypes.NewLineBlock] =         (block) => this.Accept_LeafBlock_(block);
  this.acceptors_[BlockTypes.TabBlock] =             (block) => this.Accept_LeafBlock_(block);
}
```

***Figure 51**: The AstHost class' basic functionality*

4.6 Configuration Facilities

In this section, we will briefly discuss our implementation, regarding the system's configuration facilities as well as the form of expected input configuration files.

4.6.1 Block and GUI themes

Supporting customizable block themes requires a lengthy and more complicated to understand configuration file, when compared to the simpler structure of our system's other configuration files. The block theme configuration file consists of two parts: the general, per block category theme configuration settings and the specific per grammar symbol configuration settings. Each category of blocks, due to its different rendered representation, provides different theme customization properties. Simple blocks provide the simplest theme-able configuration, since they render as rectangles that contain text. In this context, they provide customization properties for background color, left, right, top and bottom padding, font-color and size, border width, border color and border radius. However, not every block category is as simple to configure. A selection block's rendered view consists of a rectangle that contains text and an arrow, which when clicked, expands to a rectangular dropdown list that includes selectable options which, on mouse-hover, display tooltips. Each of these elements is customizable, meaning that selection blocks provide configuration properties for the main block (background color, left, right, top and bottom padding, font size and font color, width of the gap between the text and the arrow, border width, border color and border radius), the arrow (color, width, and height), the option container (background color, left, right, top and bottom padding), each inner option (background color, left, right, top and bottom padding, font size and font color), each inner option on-mouse-hover (background color and font color) as well as the option tooltip (background color, font size and font color).

In order to achieve this in a flexible way, each block category chooses, for each of its configurable elements, a subset from the set of all the configuration properties available through the system. For instance, selection blocks, as mentioned in the previous paragraph have 6 configurable rendered views: the main block view, the arrow, the option container, the inner options, the inner options on-mouse-hover and the option tooltip. For each one of these elements, there is a different supported subset of configuration properties, as they are mentioned in the previous paragraph. Furthermore, each block provides methods which return the corresponding rendered DOM elements of its configurable components. For applying the chosen configuration theme, a different component translates the properties to CSS, obtains the corresponding DOM elements and applies the properties directly to their style attribute. Furthermore, because interactive configuration such as on-hover is not applicable to the style attribute of DOM elements, configurable properties such as the on-mouse-hover background color of a selection block's inner option, are applied by obtaining the properties' CSS translation and utilizing it in specific JavaScript event handlers, or appending it directly in HTML <style> elements.

The input configuration file not only accepts configurable settings per block category, but also configurable settings per specific grammar symbol. These settings consist of configuration properties derived by the grammar symbol's block category and when specified, override them. For instance, an “if-statement” block can be configured by properties specified in the general group block category as well as in the language dependent “if-statement” grammar symbol category, with configuration properties of the latter being prioritized. In this context, the language author can specify a general background color for all group blocks and override it in the case of a specific symbol. For automatically generating a template for the language specific part of the block theme configuration file, the editor loops through all the input language’s terminal and non-terminal symbols, calculates their block category and outputs the appropriate configurable components and properties. When the system is provided with a complete theme configuration file, the editor precomputes and saves the theme for each language grammar symbol by initially assigning it its general category properties and then overriding the ones which are also provided in the specific per-grammar-symbol configuration. When a block is inserted or generated through the editor, it is automatically provided with its corresponding input theme, which is applied when the block is rendered.

The user interface theme configuration facilities use the same backbone mechanism, meaning that each configurable component provides the editor with a list of its configurable subparts and their chosen configuration properties. The editor then can export a theme template, and when given a valid theme configuration file, it can apply it to the user interface.

4.6.2 Pretty Print

As previously stated, the system uses a pretty print configuration file in order to automatically construct blocks that are structured according to a desired layout. With the absence of such a mechanism, all blocks would be one-line elements. Using pretty print, the language author can define, per say, “if-statement” blocks to have their statements in a new-line and indented by one tab character. The pretty print configuration JSON has two distinct parts, one related to group blocks, and one related to repetition group blocks.

Generating the first part of the pretty print configuration file requires looping through all the language grammar symbols that lead to producing group blocks. These are the non-terminal grammar symbols that have a right-hand-side of multiple grammar symbols and are defined using the “all_of” meta language grammar token. The editor, then, outputs, for each of these symbols, a JSON object that contains a key-value pair with the symbol’s name as its key and, as its value, an array containing all of its production’s right-hand-side symbol names. The language author can then alter the array with the right-hand-side symbols, by adding the special prefixed tokens *$$_newline* and *$$_tab* in the wanted positions. This configuration JSON object is validated and saved by the editor. In each syntax-directed insertion operation, the block generation process is followed by consulting the pretty print configuration JSON and inserting any new-line or tab blocks in the specified positions.

The second part of the pretty print configuration file is related to insertions in repetition group blocks and is used to indicate whether the editor should generate new-line blocks between their repetitive elements. While, per say, a repetition group block for statements requires each statement in a new-line, an “arguments” block for function formal parameters usually has each argument in the same line. Similarly to the previous part of the pretty print configuration file, this part is generated by looping through all the non-terminal grammar symbols which produce repetition group blocks. Each of these grammar symbols has its own entry, which has a value of true if the symbol requires generating new-line blocks and false otherwise. When the end-user generates a new block using the “+” button of a repetition group block, the editor consults the input pretty print configuration JSON and inserts a new-line block, only if required.

4.6.3 Source-Text View

Visual code has a hierarchical nature and displays grouping in a visual way without requiring the use of specific tokens such as parentheses and braces. For converting visual code to its source-text, the language author can specify any additional tokens, required for valid conversion. The structure of the system’s source-text view configuration file, which allows the addition of such tokens, is identical to that of the pretty print configuration file, with the difference that the editor accepts any *$$_* prefixed token such as *$$_(* for left-parenthesis or *$$_{* for left-brace, instead of accepting only *$$_tab* and *$$_newline*. Additionally, the configuration file allows specifying any prefixed token for inserting between the repetitive elements of repetition group blocks, instead of allowing a Boolean answer of whether to insert a new-line block or not. This is particularly useful for converting traditionally comma separated repetitive elements, such as function formal parameters. The language author has to redefine newline and tab placement, since the additional tokens of source-text conversion impact the group blocks' structure.

When the system is provided with a source-text configuration JSON object, the editor validates it and saves it. When the user issues a command to enter into the source-text view mode, the editor converts the visual code to source-text by performing a traversal in the visual code AST. In this traversal, the editor applies a specific theme to each block, which mainly erases its background-color and borders in order to simulate pure text, generates simple blocks corresponding to the tokens specified in the source-text view configuration file and places them into their correct position into the visual code AST. Any new-line or tab blocks that were defined before the conversion are saved and erased. When reentering block-based editing mode, the editor reapplies the user-defined color theme, removes all the tokens added by the source-text view mode and reinserts the new-line and tab blocks as they were before entering the source-text view mode. The language author has access to an additional configuration file, for defining the source-text color theme. This is mainly used to define the font color for the source-text view's special tokens, but also to define different font colors than those specified in the block theme configuration file for already existing grammar symbols.

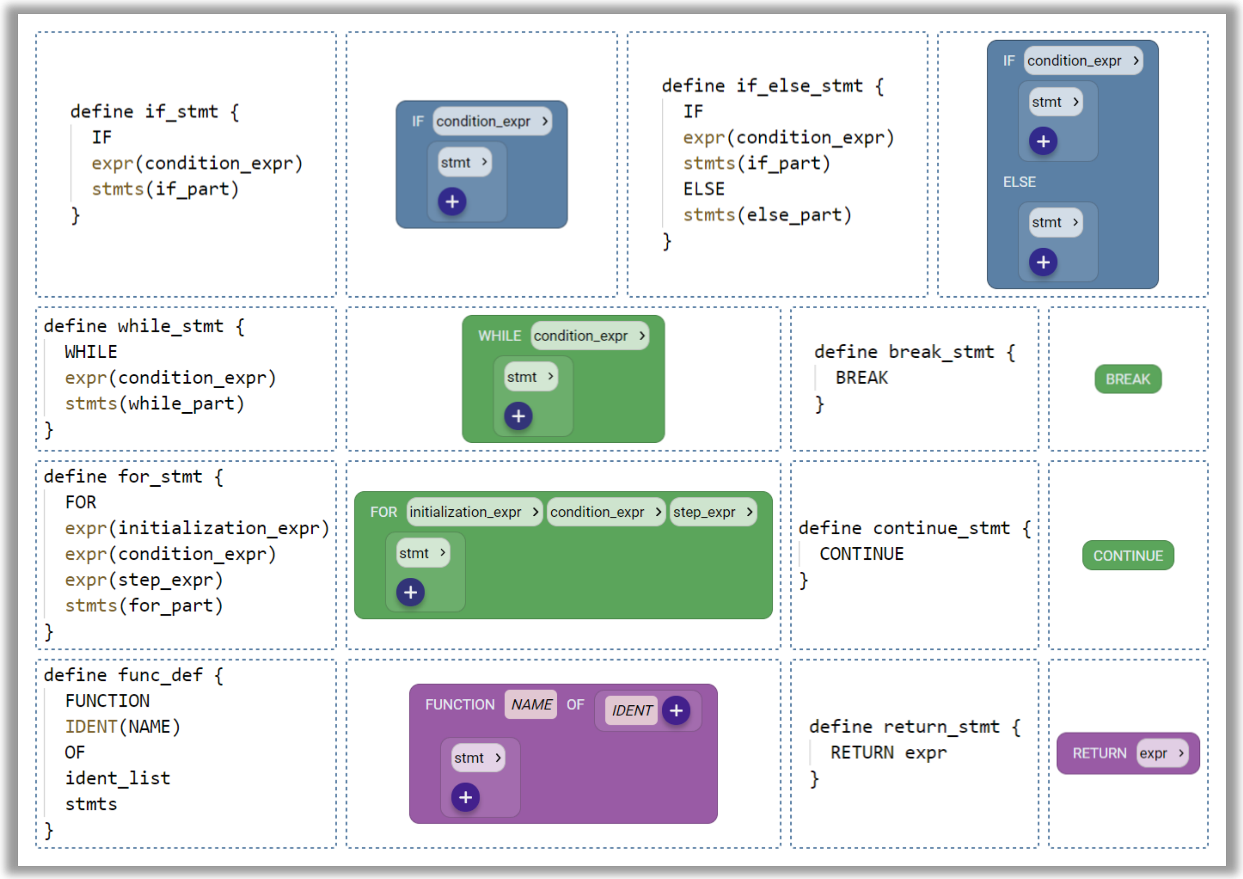
Chapter 5

Example Language

In this chapter, we will describe an example general purpose visual programming language primarily targeted at learners, which we have authored while developing the system. Throughout this process, we aim to further elaborate on the process of integrating a visual programming language into our system. Particularly, we provide the complete language grammar specification written in the Code-Chips meta language as well as show the block representation that is produced by the system, when given this language as its input. Additionally, we show how the language author can use our system's configuration facilities, such as theme and pretty printing configurations, in order to alter the visual block representation. Lastly, we demonstrate the implementation of the example language's runtime environment, showing how the language author, or generally an experienced programmer, can use the system's infrastructure in order to provide executable visual code.

5.1 Grammar Overview

In this section we will discuss the grammar of an example language and show its visual block representation. The provided example language is a simple imperative general purpose visual programming language, primarily aimed at beginners. However, for the purposes of this thesis, we have chosen to present it in the form of a traditional grammar specification and keep abbreviations such as "stmt" instead of "statement" or "expr" instead of "expression". For its practical applications in learning, especially with the absence of a tutor or a teacher, we recommend using the full non-abbreviated words, as they are more easily comprehensible by beginners.



```
define if_stmt {
  IF
  expr(condition_expr)
  stmts(if_part)
}
```

```
define if_else_stmt {
  IF
  expr(condition_expr)
  stmts(if_part)
  ELSE
  stmts(else_part)
}
```

```
define while_stmt {
  WHILE
  expr(condition_expr)
  stmts(while_part)
}
```

```
define break_stmt {
  BREAK
}
```

```
define for_stmt {
  FOR
  expr(initialization_expr)
  expr(condition_expr)
  expr(step_expr)
  stmts(for_part)
}
```

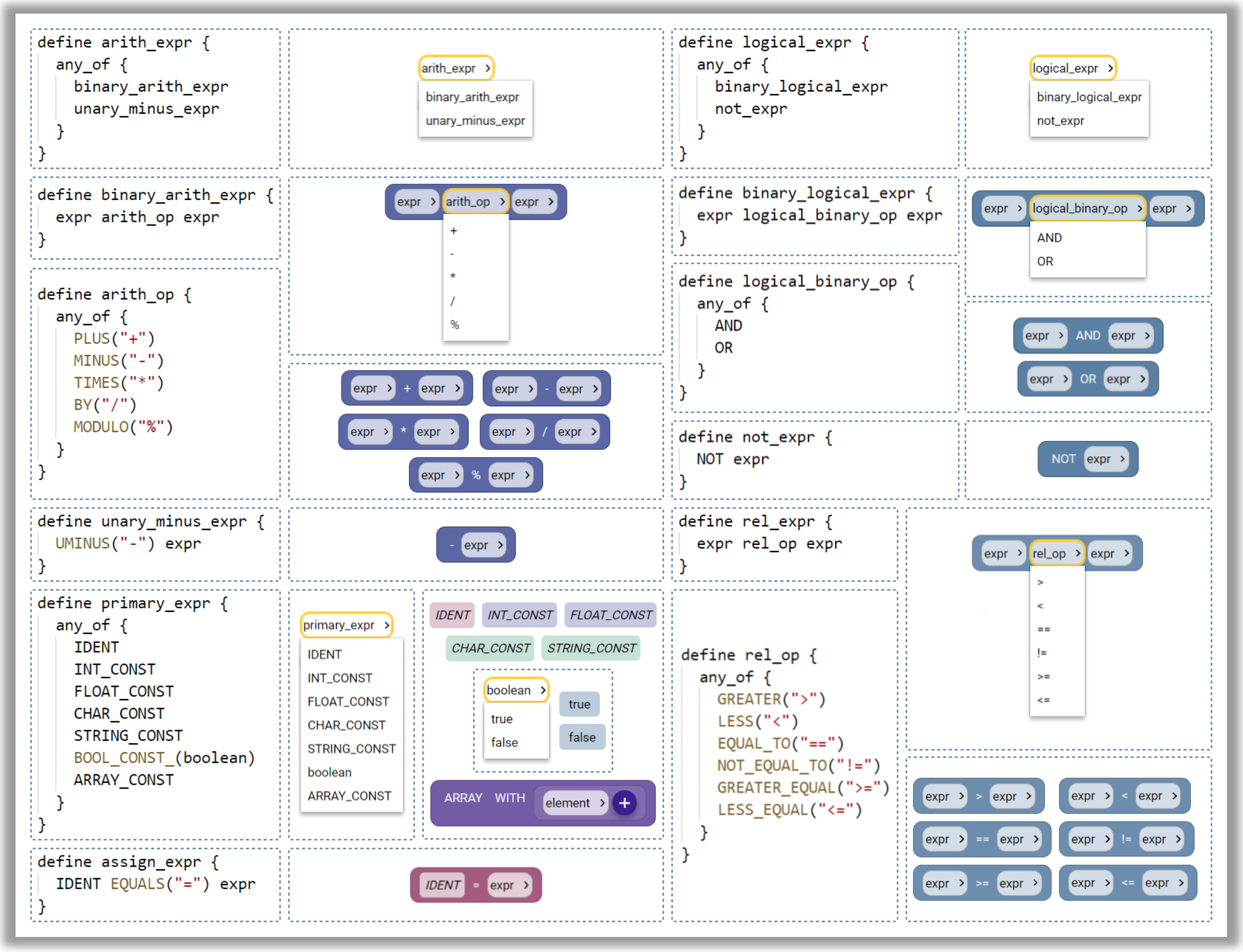
```
define continue_stmt {
  CONTINUE
}
```

```
define func_def {
  FUNCTION
  IDENT(NAME)
  OF
  ident_list
  stmts
}
```

```
define return_stmt {
  RETURN expr
}
```

***Figure 52**: All possible statement expansions, excluding "expression"*

Programs, in this example language, compose of statements which are executed, as normal, in a linear fashion. Each statement can be one of "if-statement", "if-else-statement", "while-statement", "for-statement", "expression-stmt", "function-definition", "break-statement", "continue-statement" and "return statement". For each one of these expansions, we have defined tooltips such as "Do something if a condition is true" for "if-statement" or "Continue to the next iteration of the current loop" for "continue-stmt". Note that statements such as "break", "continue" and "return" are permitted to be inserted in the context of any statement, as they are handled at runtime in order to produce valid JavaScript code. Additionally, we permit nested function definitions, which can use variables defined in outer scopes, as in JavaScript. This is usually not permitted by the visual programming languages used in popular visual programming environments, for simplicity reasons. Disallowing this for our example language, in a grammar level, is as easy as creating a new rule for the top-level statements which includes “function-definition” and excluding it from the “statement” rule.



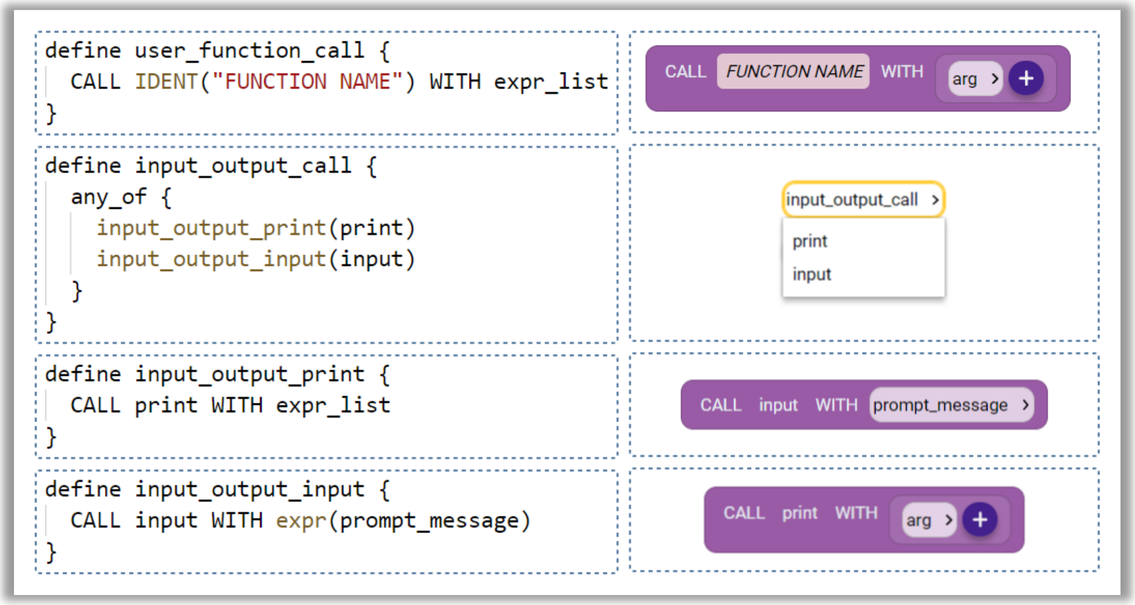
```
define arith_expr {
  any_of {
    binary_arith_expr
    unary_minus_expr
  }
}

define binary_arith_expr {
  expr arith_op expr
}

define arith_op {
  any_of {
    PLUS("+")
    MINUS("-")
    TIMES("*")
    BY("/")
    MODULO("%")
  }
}

define unary_minus_expr {
  UMINUS("-") expr
}

define primary_expr {
  any_of {
    IDENT
    INT_CONST
    FLOAT_CONST
    CHAR_CONST
    STRING_CONST
    BOOL_CONST_(boolean)
    ARRAY_CONST
  }
}

define assign_expr {
  IDENT EQUALS("=") expr
}

define logical_expr {
  any_of {
    binary_logical_expr
    not_expr
  }
}

define binary_logical_expr {
  expr logical_binary_op expr
}

define logical_binary_op {
  any_of {
    AND
    OR
  }
}

define not_expr {
  NOT expr
}

define rel_expr {
  expr rel_op expr
}

define rel_op {
  any_of {
    GREATER(">")
    LESS("<")
    EQUAL_TO("==")
    NOT_EQUAL_TO("!=")
    GREATER_EQUAL(">=")
    LESS_EQUAL("<=")
  }
}
```

Figure 53: *All possible expression expansions, excluding function calls*

Figure 52 depicts the definitions of every alternative statement expansion, other than “expression-statement”, which will be discussed later. As we discussed previously, parenthesized tokens in the grammar represent aliases for grammar symbols and are used for context-specific display of token names. For instance, in Figure 52, we use “condition_expr” as an alias for the “expr” grammar symbol in the definition of “if_stmt” and thus the system displays the alias at the corresponding block, instead of displaying “expr”.

In this example language, it is allowed to use a simple expression as a statement, similarly to many textual programming languages that usually require the addition of a semi-colon. In this context, “expression-statement” is simply an alias for “expression”. By allowing expressions as statements, and making expressions expand to assignments and function calls, we allow chaining assignments and using function calls as operands of arithmetic, relational and logical expressions as well as arguments to other function calls.

The expression grammar symbol can expand to arithmetic, logical, relational, primary, assignment and function-call expressions. Arithmetic expressions can be either binary, in the cases of addition, subtraction, multiplication, division and modulo or unary in the case of unary minus. Each of these operations has its own operand, which as in most traditional programming languages, are "+", "-", "*", "/" and "-" respectively. Similarly, logical expressions can either be binary, in the cases of logical "and" as well as logical "or", or unary in the case of logical "not". Relational expressions are always binary and can use ">", "<", "=", "!=", ">=" and "<=" as their operands. Primary expressions can be identifiers, floating or integer numbers, alpharithmetic strings or single characters, Booleans, or arrays. As we have previously discussed, blocks such as integers or identifiers require user-typed input, which is validated by the system. Figure 53 depicts all of the aforementioned expression expansions, excluding function calls, which will be depicted separately.



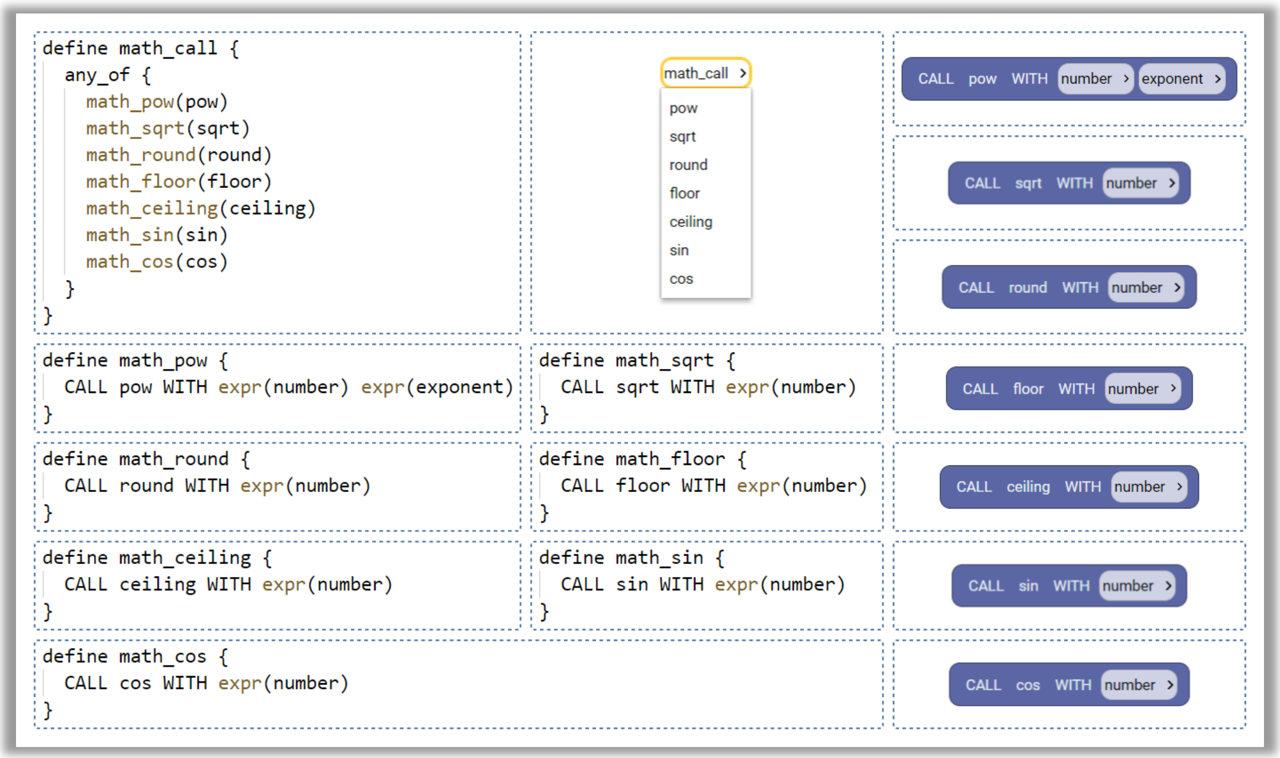
```
define user_function_call {
  CALL IDENT("FUNCTION NAME") WITH expr_list
}
```

```
define input_output_call {
  any_of {
    input_output_print(print)
    input_output_input(input)
  }
}
```

```
define input_output_print {
  CALL print WITH expr_list
}
```

```
define input_output_input {
  CALL input WITH expr(prompt_message)
}
```

Figure 54: User-defined function calls and input-output library functions



```
define math_call {
  any_of {
    math_pow(pow)
    math_sqrt(sqrt)
    math_round(round)
    math_floor(floor)
    math_ceiling(ceiling)
    math_sin(sin)
    math_cos(cos)
  }
}
```

```
define math_pow {
  CALL pow WITH expr(number) expr(exponent)
}
```

```
define math_sqrt {
  CALL sqrt WITH expr(number)
}
```

```
define math_round {
  CALL round WITH expr(number)
}
```

```
define math_floor {
  CALL floor WITH expr(number)
}
```

```
define math_ceiling {
  CALL ceiling WITH expr(number)
}
```

```
define math_sin {
  CALL sin WITH expr(number)
}
```

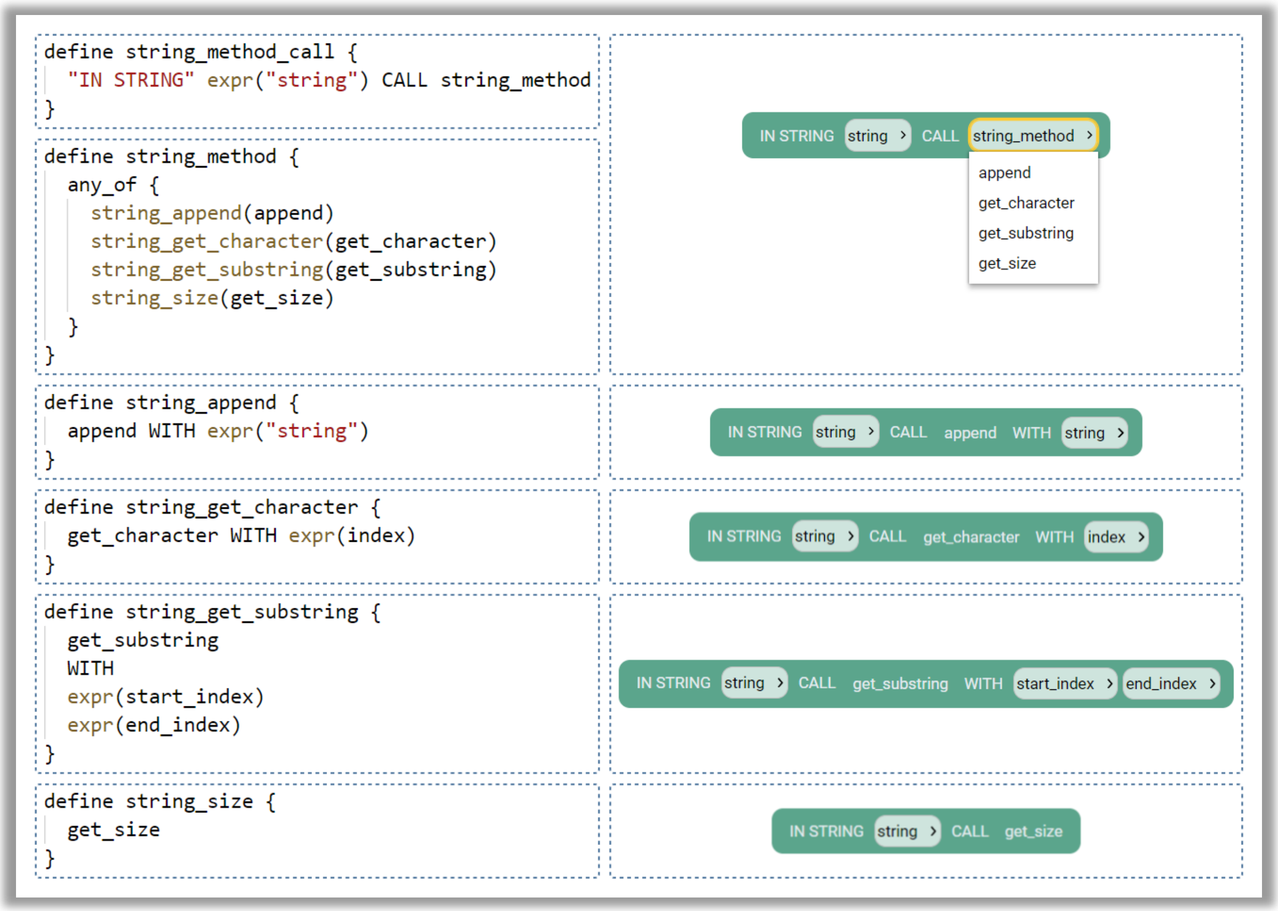
```
define math_cos {
  CALL cos WITH expr(number)
}
```

***Figure 55**: Math library functions*

In this example language, function calls are divided into two categories: user-function calls and library-function calls. User-function calls can be invoked with an unlimited number of arguments, similarly to JavaScript. For library functions, arguments are embedded into their grammar definition and only allow the user to invoke them with the correct number of arguments, similarly to blocks in prevalent visual programming environments. Currently, library functions are divided into four categories: input-output, math, strings and arrays, but the design of the language grammar allows for easy additions of more categories, or expanding a category's existing collection of functions. Figure 54 depicts the grammar definitions for user-functions as well as the input-output library functions "print", which displays an alert box with a specified message, and "input" which displays a dialog box that prompts the user for input.

A math function call (Figure 55) can expand into seven different functions: "pow", which raises a number into an exponent, "sqrt", which computes a number's square root, "round" which rounds a number into the nearest integer, "floor", which computes the greatest integer that is less than or equal to the input number, "ceiling", which computes the least integer greater than or equal to the input number, “sin” and “cos”, which compute the sine and cosine of the input angle respectively. Note that, although it is not depicted in the provided figures, we accompany each function with a tooltip, which is shown when making the expansion, as well as when hovering over an already constructed function block.

As depicted in Figure 56, string library functions follow the concept of method calls, for which the string, to which the method is applied, is mentioned before the method’s name, in contrast to passing the string as the first or last argument of a function call. The currently available string method calls are “append”, “get_character”, “get_substring” and “get_size”. String “append” returns a new string equal to the string, concatenated with the method’s argument, “get_character” returns the character at the specified position of the string, “get_substring” returns a new string that starts at “start_index” (inclusive) and ends at “end_index” (non-inclusive), and “get_size” returns the number of characters in the string.



```
define string_method_call {
  "IN STRING" expr("string") CALL string_method
}
```

```
define string_method {
  any_of {
    string_append(append)
    string_get_character(get_character)
    string_get_substring(get_substring)
    string_size(get_size)
  }
}
```

```
define string_append {
  append WITH expr("string")
}
```

```
define string_get_character {
  get_character WITH expr(index)
}
```

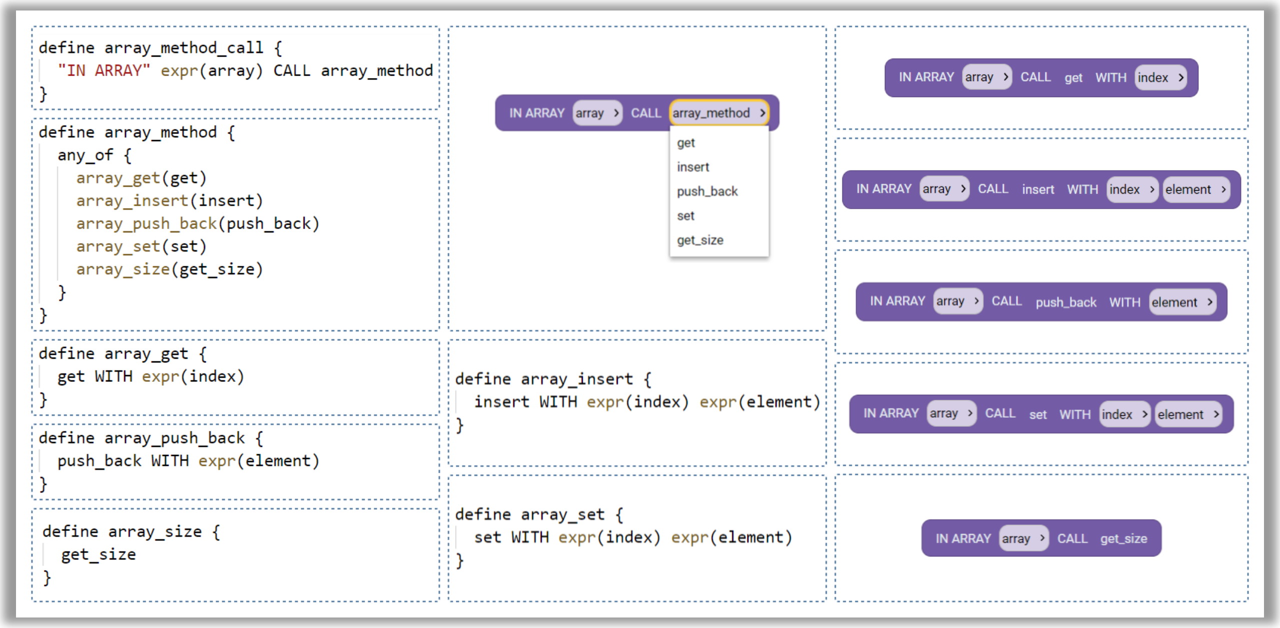
```
define string_get_substring {
  get_substring
  WITH
  expr(start_index)
  expr(end_index)
}
```

```
define string_size {
  get_size
}
```

IN STRING string CALL string_method
append
get_character
get_substring
get_size
IN STRING string CALL append WITH string
IN STRING string CALL get_character WITH index
IN STRING string CALL get_substring WITH start_index end_index
IN STRING string CALL get_size

***Figure 56**: String library functions*

Finally, the currently provided array methods (Figure 57) are “get”, “insert”, “push_back”, “set” and “get_size”. Array “get” returns the element at the given position of the array, “insert” inserts the given element at the given position and pushes the elements that follow it so that the array has one more element than before performing the insertion, “push_back” inserts the given element as the last element into the array and “get_size” returns the number of elements that are currently in the array.



```
define array_method_call {
  "IN ARRAY" expr(array) CALL array_method
}
```

```
define array_method {
  any_of {
    array_get(get)
    array_insert(insert)
    array_push_back(push_back)
    array_set(set)
    array_size(get_size)
  }
}
```

```
define array_get {
  get WITH expr(index)
}
```

```
define array_push_back {
  push_back WITH expr(element)
}
```

```
define array_size {
  get_size
}
```

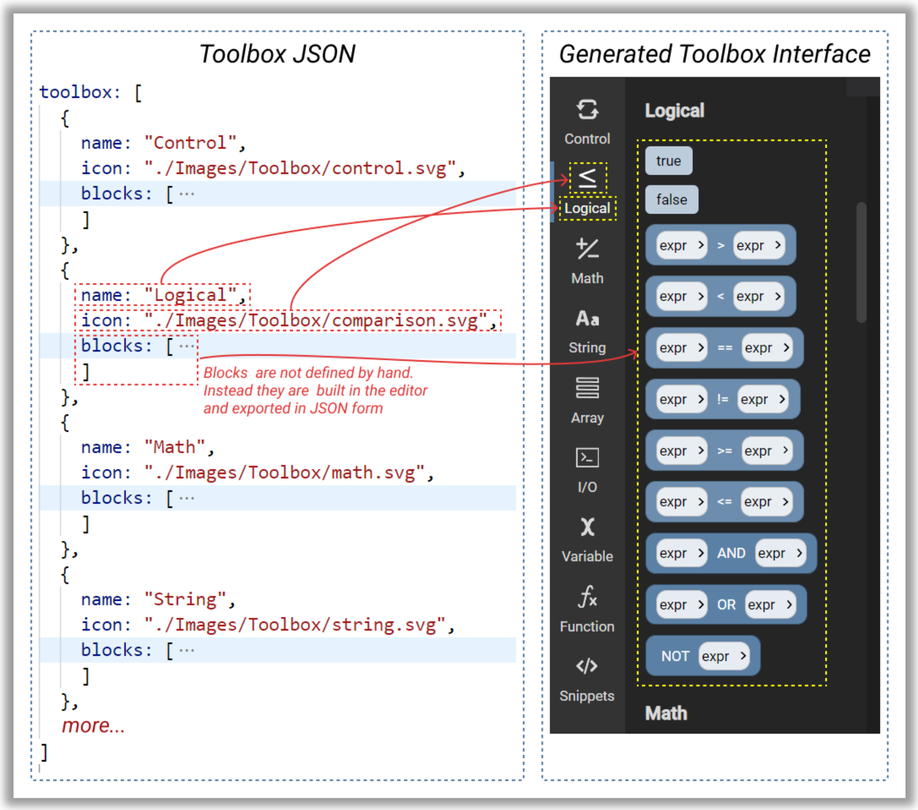
```
define array_insert {
  insert WITH expr(index) expr(element)
}
```

```
define array_set {
  set WITH expr(index) expr(element)
}
```

IN ARRAY array CALL array_method
get
insert
push_back
set
get_size
IN ARRAY array CALL get WITH index
IN ARRAY array CALL insert WITH index element
IN ARRAY array CALL push_back WITH element
IN ARRAY array CALL set WITH index element
IN ARRAY array CALL get_size

***Figure 57**: Array library functions*

5.2 Configurations



```
toolbox: [
  {
    name: "Control",
    icon: "./Images/Toolbox/control.svg",
    blocks: [ ···
    ]
  },
  {
    name: "Logical",
    icon: "./Images/Toolbox/comparison.svg",
    blocks: [ ···
    ]
  },
  {
    name: "Math",
    icon: "./Images/Toolbox/math.svg",
    blocks: [ ···
    ]
  },
  {
    name: "String",
    icon: "./Images/Toolbox/string.svg",
    blocks: [ ···
    ]
  },
  more...
]
```

Blocks are not defined by hand. Instead they are built in the editor and exported in JSON form

Figure 58: *The example language's toolbox JSON and its corresponding user interface*

In this section we will review samples of the configuration files, used for setting up the editor toolbox, altering the default block theme, as well as pretty print and source-code conversion settings. For this example language, we have divided blocks into eight categories, related to their semantics. Particularly, the chosen categories are: control, logical, math, string, array, input/output, variable and function. Each of these categories is represented by its own toolbox subarea and is accompanied by its name and icon, which we defined in an appropriate configuration file (as depicted in Figure 58). Using the editor, we have exported blocks into their JSON form and have placed them into appropriate categories. The complete toolbox block collection can be seen in Figure 59. Additionally, we have created an initially empty "Snippets" category in which end-users can freely add any block sequence for easier access.

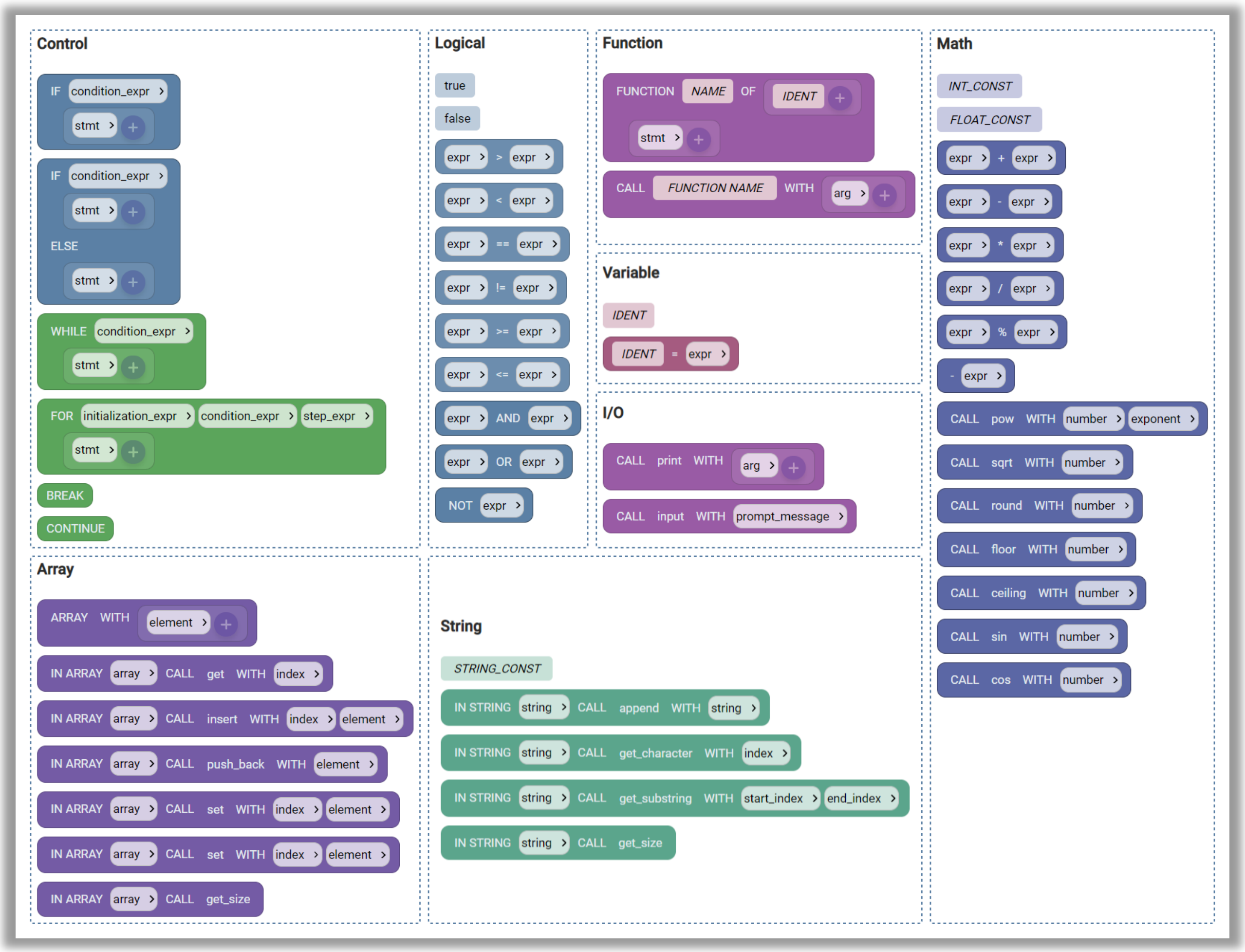
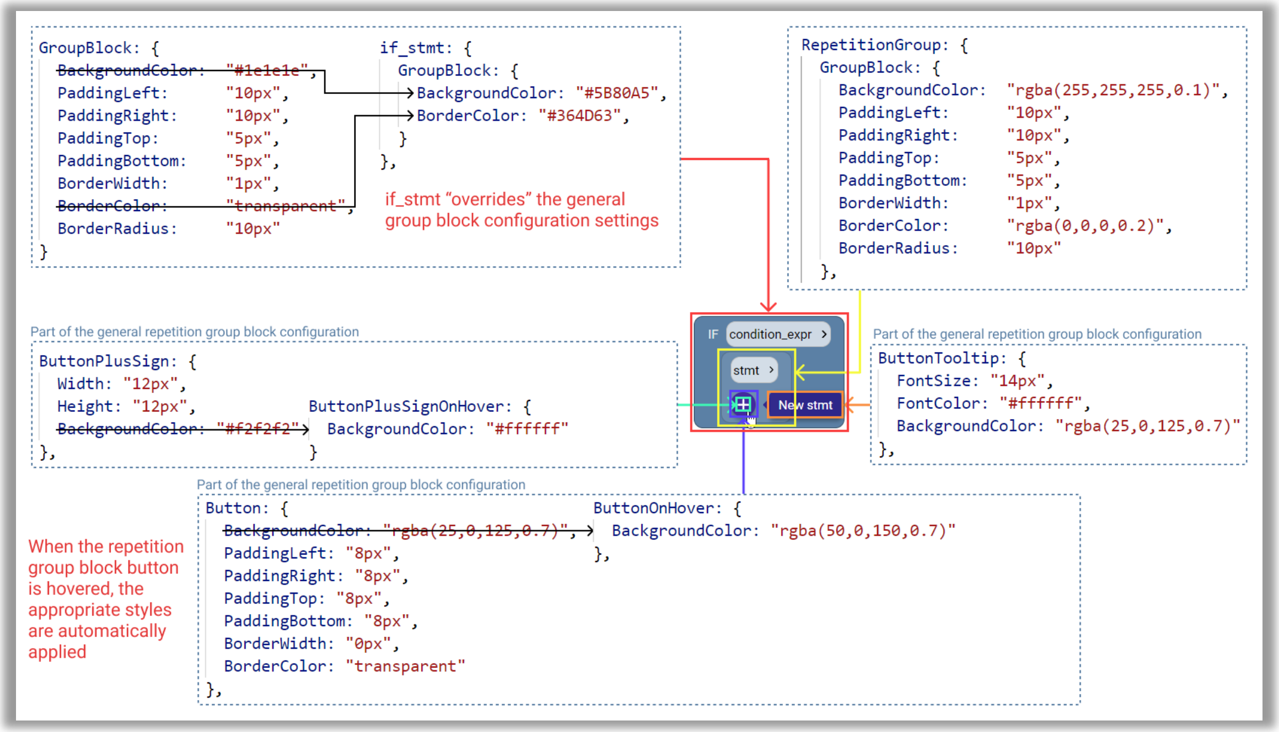


***Figure 59**: All the example language's available toolbox categories*

The visual representation of blocks, including their chosen colors, padding, inner element colors and more, is an important aspect for designing visual programming languages. For this example language, we have chosen the approach of a colorful block theme (Figure 59), similar to those of popular visual programming editors. With this approach blocks are color-coded based on their semantics in order to facilitate beginners in the learning process. Of course with the provided configuration facilities, the language authors can customize their blocks with access to different configuration properties and accomplish different visual results. For instance, Figure 37 depicts a visual code segment written in this example language, as it is viewed in three different themes.

As previously stated, in the context of block customization, the language author has access to general settings per block category, as well as specific settings per grammar symbol name, according to their block category. For instance, an “if-statement” block in this example language is of the group block category and thus can be configured with the options offered by group blocks. In this context, an “if-statement” block acquires its style by combining the general group block configuration style with the “if-statement” specific configuration style, as depicted in Figure 60. In the same figure, we provide configuration settings for different block components. Note that the provided configuration settings are expressive and allow customizations such as color-change on hover of the repetition group button or on hover of selection block options. Despite that, accomplishing the example language theme does not require much work in terms of customization, as the general settings per block category mostly suffice. The specific settings per grammar symbol are usually used for background-colors and border-colors of specific elements (for instance an “if-statement” is blue and a “while-statement” is green). However, the visual difference between the condition placeholder of an “if-statement” and that of a “while-statement” is not accomplished by specifying different background colors but relies on the alpha value for the background color of the general selection block customization settings.



```
GroupBlock: {
  ~~BackgroundColor:  "#1e1e1e"~~,
  PaddingLeft:      "10px",
  PaddingRight:     "10px",
  PaddingTop:       "5px",
  PaddingBottom:    "5px",
  BorderWidth:      "1px",
  ~~BorderColor:      "transparent"~~,
  BorderRadius:     "10px"
}
```

```
if_stmt: {
  GroupBlock: {
    BackgroundColor: "#5B80A5",
    BorderColor: "#364D63",
  }
},
```

if_stmt “overrides” the general group block configuration settings

```
RepetitionGroup: {
  GroupBlock: {
    BackgroundColor:  "rgba(255,255,255,0.1)",
    PaddingLeft:      "10px",
    PaddingRight:     "10px",
    PaddingTop:       "5px",
    PaddingBottom:    "5px",
    BorderWidth:      "1px",
    BorderColor:      "rgba(0,0,0,0.2)",
    BorderRadius:     "10px"
  },
```

Part of the general repetition group block configuration

```
ButtonPlusSign: {
  Width: "12px",
  Height: "12px",              ButtonPlusSignOnHover: {
  ~~BackgroundColor: "#f2f2f2"~~    BackgroundColor: "#ffffff"
},                             }
```

Part of the general repetition group block configuration

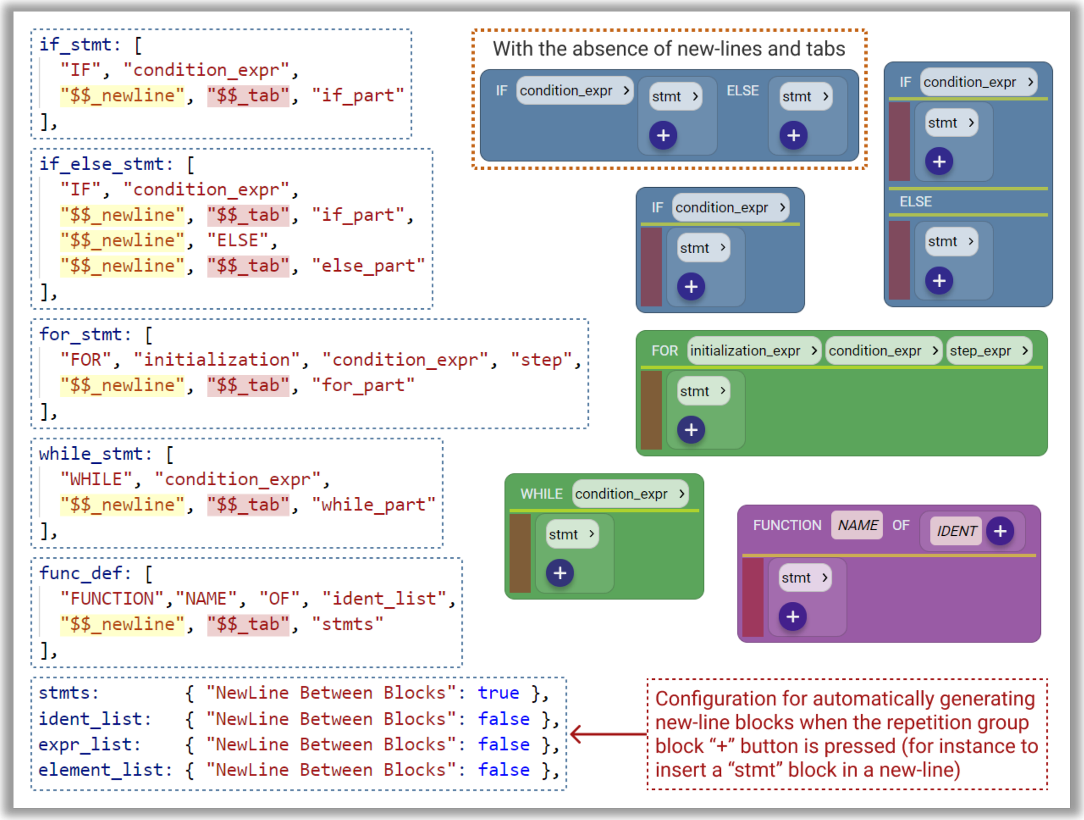
```
ButtonTooltip: {
  FontSize: "14px",
  FontColor: "#ffffff",
  BackgroundColor: "rgba(25,0,125,0.7)"
},
```

Part of the general repetition group block configuration

```
Button: {                                       ButtonOnHover: {
  ~~BackgroundColor: "rgba(25,0,125,0.7)"~~,       BackgroundColor: "rgba(50,0,150,0.7)"
  PaddingLeft: "8px",                           },
  PaddingRight: "8px",
  PaddingTop: "8px",
  PaddingBottom: "8px",
  BorderWidth: "0px",
  BorderColor: "transparent"
},
```

When the repetition group block button is hovered, the appropriate styles are automatically applied

***Figure 60**: Example theme configuration for “if_stmt”*



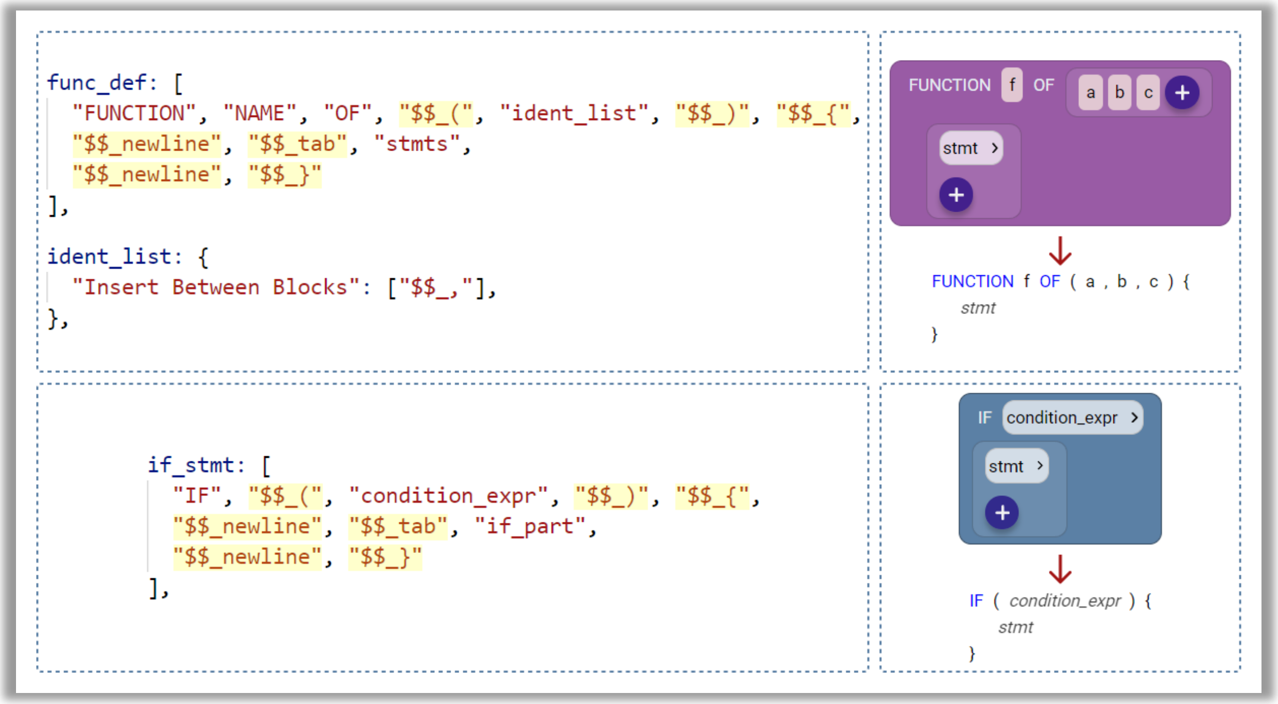
```
if_stmt: [
  "IF", "condition_expr",
  "$$_newline", "$$_tab", "if_part"
],
if_else_stmt: [
  "IF", "condition_expr",
  "$$_newline", "$$_tab", "if_part",
  "$$_newline", "ELSE",
  "$$_newline", "$$_tab", "else_part"
],
for_stmt: [
  "FOR", "initialization", "condition_expr", "step",
  "$$_newline", "$$_tab", "for_part"
],
while_stmt: [
  "WHILE", "condition_expr",
  "$$_newline", "$$_tab", "while_part"
],
func_def: [
  "FUNCTION","NAME", "OF", "ident_list",
  "$$_newline", "$$_tab", "stmts"
],
```

```
stmts:        { "NewLine Between Blocks": true },
ident_list:   { "NewLine Between Blocks": false },
expr_list:    { "NewLine Between Blocks": false },
element_list: { "NewLine Between Blocks": false },
```

With the absence of new-lines and tabs
IF
condition_expr
stmt
ELSE
stmt
FOR
initialization_expr
condition_expr
step_expr
WHILE
FUNCTION
NAME
OF
IDENT
Configuration for automatically generating new-line blocks when the repetition group block "+" button is pressed (for instance to insert a "stmt" block in a new-line)

***Figure 61**: The example language's pretty print configuration for*

This example language requires minimal customization for pretty printing. Particularly, there are five group blocks that need new-lines and tabs: "if-statement", "if-else-statement", "while-statement", "for-statement" and "function-definition". In order to define the placement of the required new-lines and tabs, the language author places the prefixed $$_newline and $$_tab keywords in an appropriate position inside the desired JSON element. Additionally, repetition group blocks require customization for determining whether the blocks generated by them will appear in a different line or not. For instance, clicking the "+" button to generate a new "statement" block automatically generates a new-line, while clicking the "+" button to generate a new function argument generates it in the same line. The whole pretty print configuration for this example language is depicted in Figure 61. Finally, for configuring source-text translation, we have used a similar configuration file to that of pretty print. Other than appropriately inserting new-lines and tabs, the source-text configuration infrastructure allows the insertion of parentheses, commas, braces and more. We have used this infrastructure to, for instance, parenthesize binary arithmetic expressions, wrap elements such as the inner statements of an "if-statement" in braces and separate function-arguments with commas. Figure 62 provides examples of configuration for conversion to source-text.



```
func_def: [
  "FUNCTION", "NAME", "OF", "$$_(", "ident_list", "$$_)", "$$_{",
  "$$_newline", "$$_tab", "stmts",
  "$$_newline", "$$_}"
],

ident_list: {
  "Insert Between Blocks": ["$$_,"],
},
```

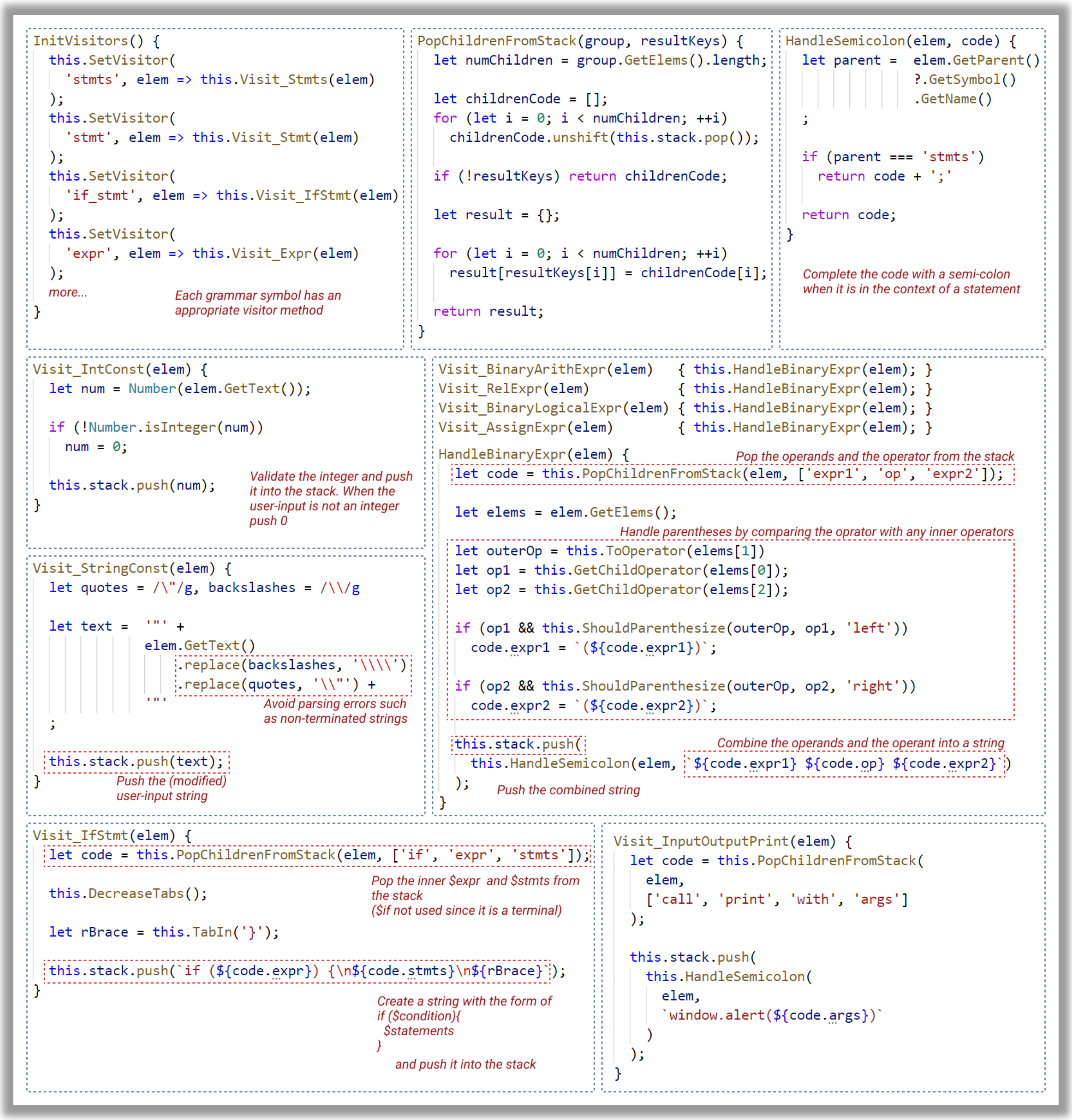
```
FUNCTION f OF ( a , b , c ) {
    stmt
}
```

```
if_stmt: [
  "IF", "$$_(", "condition_expr", "$$_)", "$$_{",
  "$$_newline", "$$_tab", "if_part",
  "$$_newline", "$$_}"
],
```

```
IF ( condition_expr ) {
    stmt
}
```

***Figure 62**: Configuration examples for conversion to source-text*

5.3 Translation and Execution



```
InitVisitors() {
  this.SetVisitor(
    'stmts', elem => this.Visit_Stmts(elem)
  );
  this.SetVisitor(
    'stmt', elem => this.Visit_Stmt(elem)
  );
  this.SetVisitor(
    'if_stmt', elem => this.Visit_IfStmt(elem)
  );
  this.SetVisitor(
    'expr', elem => this.Visit_Expr(elem)
  );
  more...
}
```

Each grammar symbol has an appropriate visitor method

```
PopChildrenFromStack(group, resultKeys) {
  let numChildren = group.GetElems().length;

  let childrenCode = [];
  for (let i = 0; i < numChildren; ++i)
    childrenCode.unshift(this.stack.pop());

  if (!resultKeys) return childrenCode;

  let result = {};

  for (let i = 0; i < numChildren; ++i)
    result[resultKeys[i]] = childrenCode[i];

  return result;
}
```

```
HandleSemicolon(elem, code) {
  let parent =  elem.GetParent()
                ?.GetSymbol()
                .GetName()
  ;

  if (parent === 'stmts')
    return code + ';'

  return code;
}
```

Complete the code with a semi-colon when it is in the context of a statement

```
Visit_IntConst(elem) {
  let num = Number(elem.GetText());

  if (!Number.isInteger(num))
    num = 0;

  this.stack.push(num);
}
```

Validate the integer and push it into the stack. When the user-input is not an integer push 0

```
Visit_BinaryArithExpr(elem)   { this.HandleBinaryExpr(elem); }
Visit_RelExpr(elem)           { this.HandleBinaryExpr(elem); }
Visit_BinaryLogicalExpr(elem) { this.HandleBinaryExpr(elem); }
Visit_AssignExpr(elem)        { this.HandleBinaryExpr(elem); }

HandleBinaryExpr(elem) {
  let code = this.PopChildrenFromStack(elem, ['expr1', 'op', 'expr2']);

  let elems = elem.GetElems();

  let outerOp = this.ToOperator(elems[1])
  let op1 = this.GetChildOperator(elems[0]);
  let op2 = this.GetChildOperator(elems[2]);

  if (op1 && this.ShouldParenthesize(outerOp, op1, 'left'))
    code.expr1 = `(${code.expr1})`;

  if (op2 && this.ShouldParenthesize(outerOp, op2, 'right'))
    code.expr2 = `(${code.expr2})`;

  this.stack.push(
    this.HandleSemicolon(elem, `${code.expr1} ${code.op} ${code.expr2}`)
  );
}
```

Pop the operands and the operator from the stack

Handle parentheses by comparing the oprator with any inner operators

Combine the operands and the operant into a string

Push the combined string

```
Visit_StringConst(elem) {
  let quotes = /\"/g, backslashes = /\\/g

  let text =  '"' +
              elem.GetText()
                .replace(backslashes, '\\\\')
                .replace(quotes, '\\"') +
              '"'
  ;

  this.stack.push(text);
}
```

Avoid parsing errors such as non-terminated strings

Push the (modified) user-input string

```
Visit_IfStmt(elem) {
  let code = this.PopChildrenFromStack(elem, ['if', 'expr', 'stmts']);

  this.DecreaseTabs();

  let rBrace = this.TabIn('}');

  this.stack.push(`if (${code.expr}) {\n${code.stmts}\n${rBrace}`);
}
```

Pop the inner $expr and $stmts from the stack ($if not used since it is a terminal)

Create a string with the form of
if ($condition){
 $statements
}
and push it into the stack

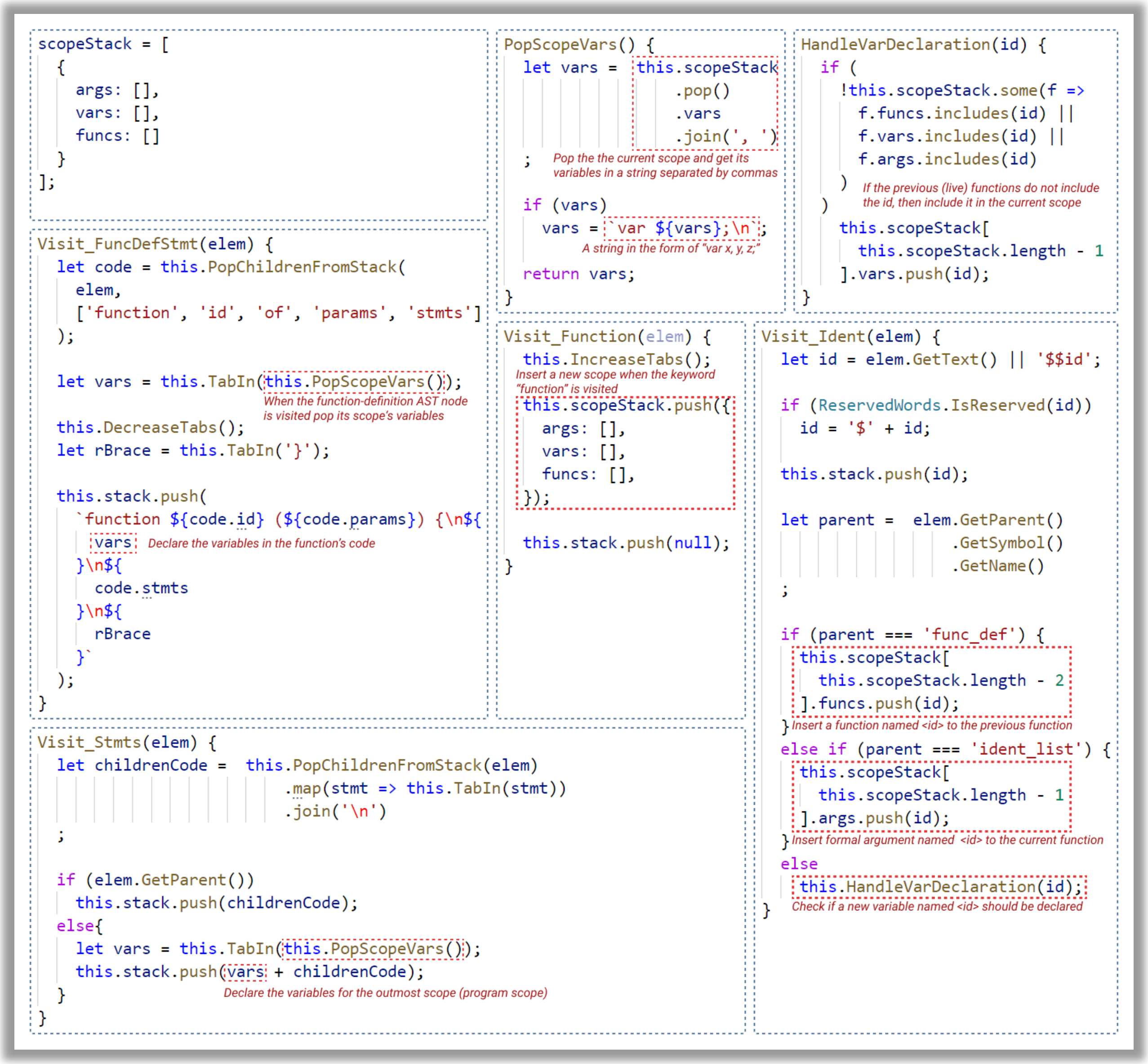
```
Visit_InputOutputPrint(elem) {
  let code = this.PopChildrenFromStack(
    elem,
    ['call', 'print', 'with', 'args']
  );

  this.stack.push(
    this.HandleSemicolon(
      elem,
      `window.alert(${code.args})`
    )
  );
}
```

Figure 63: *Examples from the runtime implementation of the example language*

In this section, we discuss the implementation for the example language's runtime environment. For the purposes of this thesis we do not provide the complete code-base of the implemented runtime environment and instead focus on key aspects for understanding its functionality. Execution is based on translating the visual code AST to JavaScript and executing it by using the JavaScript "eval" function.

In this context, we have used the previously described AstVisitor base class, and have authored a subclass in order to translate the visual code AST to JavaScript. The subclass uses a stack, to which AST nodes (blocks) push their converted JavaScript equivalent. Group blocks, which have more than one inner block, utilize the code segments that are pushed by their inner blocks, combine them appropriately and push the result into the stack. For instance, an "if-statement" group block that consists of an "if" simple block, a "condition" block and a "statements" block, is responsible for combining these and pushing a syntactically correct JavaScript "if-statement" by including parentheses before and after the condition, and braces before and after the combined statements. At the same time, we would like the code to be not only executable but also humanly viewable, in terms of indentation and parenthesization. To accomplish correct indentation, the implemented visitor subclass keeps track of the current count of tabs. Each AST node can increase, decrease or use this count to insert tabs where needed. To avoid unnecessary parentheses, AST nodes that use operators have to factor in their converted JavaScript operator precedence and associativity. For instance a binary expression of "(expr1) and (expr2)", for which expr1 is "x or y" and expr2 is "not z" is converted to "(x or y) and !z", as our implementation compares "and" to "or" as well as "not" to add parentheses only when needed. Figure 63 provides code and further explanations on our implementation.

An important aspect of the example language's semantics is variable definitions and declarations. The example language, as previously described, does not provide blocks for declaring variables but uses declaration by use, meaning that variables are automatically declared on their first use. At the same time, as previously stated, the example language allows nested function definitions. In this context, inner functions have access to variables defined in outer functions, similarly to JavaScript, but we do not allow shadowing outer scope variables, since there is no variable declaration keyword (shadowing outer scope variables is only possible by defining a function with the same name). For implementing such functionality, each translated JavaScript function defines the variables that were found directly in their statements, excluding variables that were references to ones of outer functions. With this implementation, a variable that was defined in an outer function is not shadowed in an inner function. In code, each function definition AST node is associated with its own scope, which tracks variables, functions and formal arguments. Every AST node that uses a variable identifier checks if this specific identifier is defined in an outer live scope. If the answer is positive, no modifications are made to the current scope, otherwise the identifier is added to it. When a function-definition node is visited, it declares its own scope's variables using the JavaScript "var" keyword. Figure 64 depicts code for all the required scope related operations.



```
scopeStack = [
  {
    args: [],
    vars: [],
    funcs: []
  }
];
```

```
Visit_FuncDefStmt(elem) {
  let code = this.PopChildrenFromStack(
    elem,
    ['function', 'id', 'of', 'params', 'stmts']
  );

  let vars = this.TabIn(this.PopScopeVars());
                        When the function-definition AST node
                        is visited pop its scope's variables
  this.DecreaseTabs();
  let rBrace = this.TabIn('}');

  this.stack.push(
    `function ${code.id} (${code.params}) {\n${
      vars  Declare the variables in the function's code
    }\n${
      code.stmts
    }\n${
      rBrace
    }`
  );
}
```

```
Visit_Stmts(elem) {
  let childrenCode =  this.PopChildrenFromStack(elem)
                          .map(stmt => this.TabIn(stmt))
                          .join('\n')
  ;

  if (elem.GetParent())
    this.stack.push(childrenCode);
  else{
    let vars = this.TabIn(this.PopScopeVars());
    this.stack.push(vars + childrenCode);
  }                Declare the variables for the outmost scope (program scope)
}
```

```
PopScopeVars() {
  let vars = this.scopeStack
                 .pop()
                 .vars
                 .join(', ')
  ;  Pop the the current scope and get its
     variables in a string separated by commas

  if (vars)
    vars = `var ${vars};\n`;
        A string in the form of "var x, y, z;"
  return vars;
}
```

```
Visit_Function(elem) {
  this.IncreaseTabs();
  Insert a new scope when the keyword
  "function" is visited
  this.scopeStack.push({
    args: [],
    vars: [],
    funcs: [],
  });

  this.stack.push(null);
}
```

```
HandleVarDeclaration(id) {
  if (
    !this.scopeStack.some(f =>
      f.funcs.includes(id) ||
      f.vars.includes(id) ||
      f.args.includes(id)
    )  If the previous (live) functions do not include
       the id, then include it in the current scope
  )
    this.scopeStack[
      this.scopeStack.length - 1
    ].vars.push(id);
}
```

```
Visit_Ident(elem) {
  let id = elem.GetText() || '$$id';

  if (ReservedWords.IsReserved(id))
    id = '$' + id;

  this.stack.push(id);

  let parent =  elem.GetParent()
                    .GetSymbol()
                    .GetName()
  ;

  if (parent === 'func_def') {
    this.scopeStack[
      this.scopeStack.length - 2
    ].funcs.push(id);
  } Insert a function named <id> to the previous function
  else if (parent === 'ident_list') {
    this.scopeStack[
      this.scopeStack.length - 1
    ].args.push(id);
  } Insert formal argument named  <id> to the current function
  else
    this.HandleVarDeclaration(id);
    Check if a new variable named <id> should be declared
}
```

***Figure 64**: Scope handling in the runtime environment of the example language*

Chapter 6

Conclusion and Future Work

This thesis presents Code-Chips: a general-purpose syntax-directed visual programming editor that accepts the actual language grammar as its input. For this purpose, we have defined a simple but expressive meta language through which experienced programmers can author visual programming languages. Given a valid language grammar specification, the system generates language dependent configuration files, which can customize its user interface and blocks.

The system translates the input grammar symbols to interactive blocks which support syntax-directed editing operations and block-based visual programming operations. In contrast to traditional syntax-directed editors, end-users are not required to remember and recall commands. Instead, by interacting with selection blocks, end-users are given the ability to choose from all the available expansions in the given context.

In contrast to popular visual programming editors, the underlying language grammar is explicit and is directly communicated to the end-users. Additionally, the system empowers end-users with the ability to inspect the production chain of any formed program element, by viewing it in a block-based form.

Through the syntax-directed aspect of the system, end-users can benefit from a learning process based on language exploration. The system's visual programming aspect, including the block-based form for program elements, the toolbox with its ready-made language productions in the form of blocks, and the support for syntactic drag-and-drop insertions, allow for introducing beginners to programming. In combination, even experienced programmers can benefit from learning grammatically advanced or new language concepts, using a textual programming language within the system.

Although we believe the presented system can help beginners as well as more experienced programmers in the aforementioned manner, Code-Chips has yet to be evaluated and tested by real users in real scenarios. Particularly, it is our intent to conduct case studies through which we will collect valuable information on how the system is perceived in teaching scenarios, from the student's perspective, as well as from the educator's point of view. Additionally it would be interesting to measure how the system can solely, or in combination with traditional methods, benefit a developer in forming in-depth knowledge of syntactically complicated language concepts.

Another area in which the system can improve is the facilities for being integrated into application specific educational environments. Although Code-Chips offers functionality for embedding a programming language and customizing its produced blocks, this process requires using the meta language and altering configuration files. In the future, we plan on facilitating such tasks with graphical user interfaces. Particularly, the meta language can be hosted by the editor itself, and thus provide a means for defining languages for the system, within the system. Furthermore, the available customization properties can be the input of a user interface generator which will allow viewing and altering them, as well as provide immediate feedback.

Finally, it would be interesting to examine how the system could navigate from its current syntax-directed features, which are based on grammatical structure, to also support more context-aware features such as intelligent code completion for variable names, functions and methods.

Bibliography

[1] Khwaja, Amir Ali, and Joseph E. Urban. "Syntax-directed editing environments: Issues and features." Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice. 1993.

[2] Lang, Bernard. "On the usefulness of syntax directed editors." Advanced Programming Environments. Springer, Berlin, Heidelberg, 1987.

[3] Teitelbaum, Tim, and Thomas Reps. "The Cornell program synthesizer: a syntax-directed programming environment." Communications of the ACM 24.9 (1981): 563-573.

[4] Goldenson, Dennis R. "Learning to Program with Structure Editing: An Update and Some Replications." Proceedings of the National Educational Computing Conference. 1990.

[5] Myers, Brad A. "Taxonomies of visual programming and program visualization." Journal of Visual Languages & Computing 1.1 (1990): 97-123.

[6] Maloney, John, et al. "The scratch programming language and environment." ACM Transactions on Computing Education (TOCE) 10.4 (2010): 1-15.

[7] "With MIT App Inventor, Anyone Can Build Apps with Global Impact." *MIT App Inventor | Explore MIT App Inventor*, https://appinventor.mit.edu/. Accessed Dec. 2021.

[8] Hils, Daniel D. "Visual languages and computing survey: Data flow visual programming languages." Journal of Visual Languages & Computing 3.1 (1992): 69-101.

[9] Cox, Philip T., F. R. Giles, and Tomasz Pietrzykowski. "Prograph: a step towards liberating programming from textual conditioning." 1989 IEEE Workshop on Visual languages. IEEE Computer Society, 1989.

[10] "Unity Shader Graph: Build Your Shaders Visually with Unity." *Unity*, https://unity.com/shader-graph. Accessed December 2021.

[11] “Blueprint Visual Scripting.” *Unreal Engine Documentation*, https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/. Accessed December 2021.

[12] “What Is Max?” *What Is Max? | Cycling '74*, https://cycling74.com/products/max. Accessed December 2021.

[13] Scherer, Ronny, Fazilat Siddiq, and Bárbara Sánchez Viveros. "The cognitive benefits of learning computer programming: A meta-analysis of transfer effects." Journal of Educational Psychology 111.5 (2019): 764.

[14] Liao, Yuen-Kuang Cliff, and George W. Bright. "Effects of computer programming on cognitive outcomes: A meta-analysis." Journal of educational computing research 7.3 (1991): 251-268.

[15] Buitrago Flórez, Francisco, et al. "Changing a generation’s way of thinking: Teaching computational thinking through programming." Review of Educational Research 87.4 (2017): 834-860.

[16] Yaroslavski, Danny. "How does Lightbot teach programming." Retrieved January 29 (2014): 2016.

[17] Blockly Games, https://blockly.games/. Accessed December 2021.

[18] “Learn Computer Science. Change the World.” *Code.org*, https://code.org/. Accessed December 2021.

[19] “Coding for Kids, Kids Programming Classes and Games: Tynker.” *Tynker.com*, https://www.tynker.com/. Accessed December 2021.

[20] CodeCombat. “Coding Games to Learn Python and JavaScript.” *CodeCombat*, CodeCombat, https://codecombat.com/. Accessed December 2021.

[21] “Coding for Kids: Game-Based Programming.” *CodeMonkey*, https://www.codemonkey.com/. Accessed December 2021.

[22] Chomsky, Noam. Knowledge of language: Its nature, origin, and use. Greenwood Publishing Group, 1986.

[23] Piteira, Martinha, and Carlos Costa. "Learning computer programming: study of difficulties in learning programming." Proceedings of the 2013 International Conference on Information Systems and Design of Communication. 2013.

[24] Denny, Paul, et al. "Understanding the syntax barrier for novices." Proceedings of the 16th annual joint conference on Innovation and technology in computer science education. 2011.

[25] Zelkowits, Marvin V. "A small contribution to editing with a syntax directed editor." ACM Sigplan Notices 19.5 (1984): 1-6.

[26] Donzeau-Gouge, Veronique, et al. Programming environments based on structured editors: The MENTOR experience. INSTITUT NATIONAL DE RECHERCHE D'INFORMATIQUE ET D'AUTOMATIQUE ROCQUENCOURT (FRANCE), 1980.

[27] Fischer, Charles N., et al. "The POE language-based editor project." ACM SIGSOFT Software Engineering Notes 9.3 (1984): 21-29.

[28] Fraser, N. "Google Blockly-a visual programming editor" https://developers.google.com/blockly. Accessed December 2021.

[29] "Blockly Developer Tools", https://developers.google.com/blockly/guides/create-custom-blocks/blockly-developer-tools. Accessed December 2021.

[30] "Microsoft MakeCode Computer Science Education." Microsoft MakeCode, https://www.microsoft.com/en-us/makecode. Accessed December 2021.

[31] Bernat Romagosa, Michael Ball. "Welcome to Snap!" Snap! Build Your Own Blocks, https://snap.berkeley.edu/. Accessed December 2021.

[32] JQuery, https://jquery.com/. Accessed December 2021.

[33] Carter, Zach. "Jison." Jison / Documentation, https://gerhobbelt.github.io/jison/docs/. Accessed December 2021.

[34] Levine, John. Flex & Bison: Text Processing Tools. " O'Reilly Media, Inc.", 2009.