Retrofitting a Virtual Instrument DSL with Programming Abstractions

Mathias Vatter
JGU Mainz
Germany, Mainz

Sebastian Erdweg
JGU Mainz
Germany, Mainz

Abstract

KSP is an imperative DSL in music production that enables realistic modelling of musical instruments in real-time using Kontakt as a runtime environment. Once a niche topic for hobbyists, the field has since professionalized, with Kontakt becoming an industry standard. Its scripting language, however, has not evolved much, lacking modern functional and data abstractions while remaining closed-source. This paper proposes transformations that introduce modularity and basic abstraction principles to KSP. This entails functions with parameters and return values, recursive data types, and the implementation of lexical scope to replace the current global variable management. The transformations have been implemented in a preprocessing compiler framework–preceding the actual KSP interpreter–to an extend, that allows for the new syntax elements to be used in real-world KSP scripts.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; Preprocessors; • **Applied computing** → Sound and music computing.

Keywords: Virtual Instrument, KSP, Kontakt Script Processor, Language Abstraction

ACM Reference Format:
Mathias Vatter and Sebastian Erdweg. 2025. Retrofitting a Virtual Instrument DSL with Programming Abstractions. In *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '25), July 3–4, 2025, Bergen, Norway.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3742876.3742878

1 Introduction

By providing notations and constructs tailored to the requirements of a specific domain, domain-specific languages can significantly improve domain expert involvement in software development and widen accessibility by requiring less programming knowledge [12][26][1]. As Mernik et al. (2005) note, specialized notation increases expressiveness and productivity by reducing boilerplate code. However, this specialization sacrifices the flexibility of general-purpose programming languages (GPLs). DSLs often feature proprietary, non-standardized syntaxes which, coupled with issues like poor documentation and high development costs, can limit reusability and hinder long-term adoption. [12]

The Kontakt Script Processor (KSP) is a DSL primarily used in music production. Introduced around the same time as the aforementioned DSL literature, KSP is part of Kontakt—a software sampler for audio recordings (samples) [28][18][2]. Samplers are a type of virtual instrument often used as plugins for composition in digital audio workstations (DAWs) [24]. As a platform, Kontakt is widely used to create sample libraries that enable the realistic, real-time modelling of acoustic instruments, replicating their characteristics [25]. Released in 2004 with Kontakt 2 [19][20], KSP was advertised as an effect or composition tool [11], allowing access to and manipulation of Kontakt-specific parameters. Similar to many DSLs, official information on KSP's design is scarce beyond basic documentation and an active online community, from which much of this papers' information is sourced. KSP is an imperative, event-driven, strongly-typed language, with rudimentary control structures and data types. Its syntax shows influences from languages like Pascal/Ada, such as the use of the `:=` operator for assignments, or Perl with its use of the first character of variable names to denote their type. Critically, KSP lacks abstraction and modularity: it supports only global variables, primitive data types, statically declared arrays, and no function definitions with parameters or return values. These limitations, often justified by real-time audio demands [5], alongside its basic built-in editor (lacking syntax highlighting or autocomplete), suggest KSP was initially intended only for small scripts.

As is common for DSLs, the syntax has hardly changed since its introduction, even though music production and virtual instruments—along with Kontakt's usage—have evolved considerably. Virtual instruments now play a crucial role in film and video game soundtracks, Kontakt has established itself as an industry standard for composers and the development of sample libraries has become an entire business sector [6][4][7][18][27]. What once consisted of simple products playing back pre-recorded audio loops has

GPCE '25, July 3–4, 2025, Bergen, Norway

ACM ISBN 979-8-4007-1995-0/2025/07
https://doi.org/10.1145/3742876.3742878

evolved into sophisticated libraries with preset systems, comprehensive user interfaces and advanced automation and sequencing capabilities.

Increasing script complexity and industry professionalization has led to the hiring of more software developers. At the same time, the language's limitations might deter a lot of users. KSP seems to now reflect some of the disadvantages listed by Mernik et al, as syntactic peculiarities, limited generality, a changing domain and the high investment costs for adapting a DSL become noticeable. Meanwhile, KSP's closed-source nature prevents direct community improvements. Thus, introducing abtractions requires lowering transformations that work around the language's inherent limitations. We found ways around these and present three advanced abstractions for KSP:

- KSP's requirement that all variables be declared in the global scope increases complexity and limits modularity, as code blocks cannot be isolated and reused independently. We introduce lexically scoped variables, which we lower to global variables. However, for performance, it is important to limit the number of global variables used, hence we carefully support variable reuse (using a lifetime analysis) while taking thread-safety and callbacks into account (Section 3).
- We introduce user-defined functions with arguments and call-by-reference return variables into KSP. This requires a series of transformations to promote return parameters, hoist function calls to statements, and eventually inline function definitions (Section 4).
- KSP only supports primitive data types and statically declared arrays, which complicates modelling complex data structures. We introduce an interface for defining recursive Algebraic Data Types (ADTs) and lower them to KSP arrays. This necessitates complex transformations, including the introduction of memory management by reference counting to remain compatible with real-time applications (Section 5).

Overall, this paper shares our experience of introducing programming abstractions into a severely limited DSL. While it seemed infeasible at first, our experience shows that retrofitting a DSL with abstractions can be possible if one is willing to invest significant effort to find and implement analyses and transformations that work around the DSL's limitations. We believe that this experience can inspire others to seek programming abstractions for other DSLs, possibly applying similar techniques.

2 The KSP Syntax

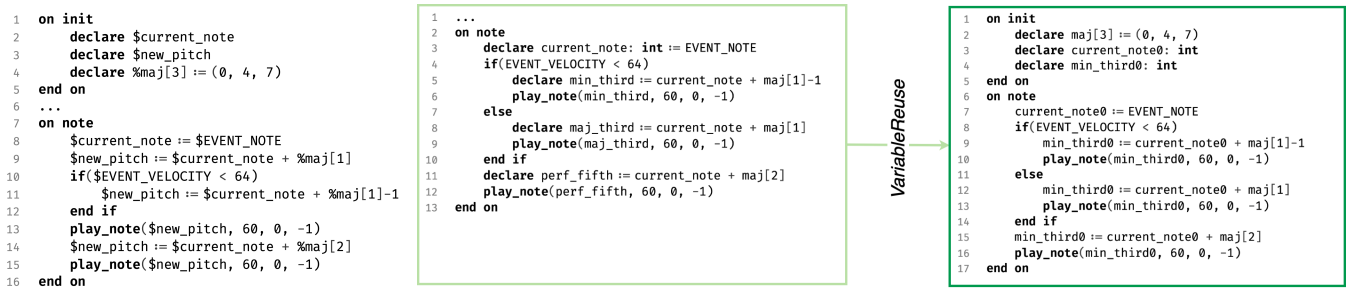
To get a grasp of its syntax and limitations, the basic concepts and constructs of the DSL will be illustrated with an example. KSP is an event-driven language where scripts consist of event handlers (callbacks), enclosed by `on <event-name>` and `end on`. In between, statements are executed sequentially. The callbacks form the highest level of the program and are triggered by specific events.

Listing 1a shows two callbacks: `on init` and `on note`. The `on init` callback is executed once on loading the script, while the `on note` callback is invoked on every note-on event. This particular example plays a triad in root position above every note the user plays on its MIDI controller [14]. If the input velocity is below `64`, a minor triad is played, otherwise a major triad. The *built-in* variable `$EVENT_NOTE` holds the MIDI note number that triggered the `on note` callback, `$EVENT_VELOCITY` its velocity. `$` is the type prefix for *integer* variables, `%` for arrays. `%maj` is a *user-defined* array containing the semitones of a major triad with its size determined by a compile-time constant. All *user-defined* variables must be declared in the `on init` callback (using `declare`) before they can be used. When a note event is triggered the `if` statement (line 10) decides on assigning a minor or a major third to the variable `$new_pitch` based on the velocity. The *built-in* command `play_note` (that can trigger additional note events) is then called with its `note` and `velocity` parameters. Here, the parameter `note` is represented by `$new_pitch` and calclulated by adding the root note to the interval from `%maj`, resulting in all samples corresponding to triad's MIDI notes being played simultaneously. In addition to *integers*, KSP supports *real* and *string* types, each with its own prefix. [15][11][19]

3 From Lexical to Global Scope

As variables in KSP are exclusively global and declared in the `on init` callback [17], they can be referenced in any subsequent callback. Declarations in other callbacks are not possible because KSP does not allow dynamic memory allocation. Instead, all variables are statically allocated and remain in memory for the entire runtime of the script.

In contrast, traditional lexical (static) scoping—introduced in ALGOL 60—allows a variable's visibility to be determined from the source code alone [22][13]. This facilitates a clear understanding on which variables belong to which section of the code. Using such a system in Listing 1a, would mean that the entire `on init` callback could be omitted, since all the variables declared there, are only used in the `on note` callback. Specifically, lexical scoping could be used to create a version like the one in Listing 1b (left) allowing for better readability and assignment of the variables without having to constantly switch to the `on init` callback. Above all, the now local variables could be given more expressive names, such as the descriptive names used here for the individual intervals. To achieve this, a transformation process has to be implemented that efficiently transfers the local variables into the global scope.



(a) A KSP Script with global scope.

(b) Example of a script with lexical scope before and after *Variable Reuse*.

Figure 1. Example of a KSP Script (left) and *Variable Reuse* Algorithm (right).

3.1 Establishing Lexical Scoping

Before this transformation, however, a verification of the lexical scope and its rules is needed. In our AST, a scope is represented by a *BlockNode* holding an ordered list of statements (e.g., declarations and control structures). Variables declared in the `on init` callback are global, variables in other callbacks and subprograms are local by default. A reference to an undeclared variable, or multiple declarations of the same variable within a single scope, triggers an error. However, *shadowing* is allowed. Variables can be declared with the same name as a variable in the outer scope without affecting the latter.

The approach follows standard compiler techniques [3]. A *stack frame* is used to store the variable declarations. However, instead of a LIFO stack, we use an array of maps where each map represents a scope that links variable names to their declarations. The *BuildLexicalScope* process traverses the AST using a visitor pattern [8]. It pushes a new map onto the stack structure when entering a block and pops it when leaving. Declarations are added to the current (top) map, except for those explicitly declared as *global*, which are added to the bottom map representing the global scope (`on init` callback). If a variable reference is encountered, each map is searched from the most local to the global scope to bind it to its declaration. If no declaration is found, an *UndeclaredVariableException* is raised; likewise, if a duplicate declaration is attempted in the same scope, a *RedeclaredVariableException* occurs. This yields an AST_{out} with verified lexical scope and variable references unambiguously linked to their declarations.

After this compiler pass, the references of `current_note` in Listing 1b (lines 5, 8, 11) are bound to their local declaration in line 3, while the reference to `min_third` in the nested scope of lines 5–6 is bound to its declaration in line 5.

3.2 Variable Reuse

A naive method to transform from lexical scope to KSP's global scope is to promote all local variables by declaring them in the `on init` callback. Although this makes the code compatible with KSP, it increases the number of static declarations and memory usage—an issue in real-time audio processing where all such variables reside in RAM. With this approach, Listing 1b (left) would have three more variables than the KSP version in Listing 1a.

To address this, the *Variable Reuse* algorithm exploits the fact that a local variable's lifetime ends when its scope is exited, making it available for reuse in subsequent callbacks. Once a scope is exited, its variables are considered *passive*, qualifying them for reuse, while global variables remain non-reusable to avoid conflicts. Since KSP always initializes freshly declared variables, all replaced declarations are converted into type-neutral assignments, while any pre-assigned values are retained.

The implementation centers on a map (*PassiveVariablesMap*), collecting *passive* variables, using a hash h computed from each variable's type and size (with the size factor applied only to arrays):

$$\text{computeHash}(variable) = \langle \text{Type}(variable), \text{Size}(variable) \rangle$$

Additionally, *PassiveVariablesUsedMap* tracks which *passive* variables have been used in the current block to prevent multiple reuses in the same scope, while a stack of maps (*PassiveVarsReplace*) maintains the binding between original variable names and their replacements. Global variables are stored in *GlobalDeclarations* to prevent reuse.

During AST traversal, when a block is visited, a new map is pushed onto the stack frame. As the statements within are processed, passive variables are identified and collected. Upon exiting the block, they are added to *PassiveVariablesMap* for reuse in subsequent scopes while any passive variables used in that block are released by removing the topmost *PassiveVarsReplace* map. If a declaration is *global*, it is deleted from the current AST position and added to *GlobalDeclarations*. For a local declaration, h is calculated to locate a matching, free *passive* variable. If a match is found in *PassiveVariablesMap* and not already in use, the declaration is replaced by an assignment and recorded in the topmost *PassiveVarsReplace* map. Otherwise, it is marked for future addition as *passive* once the block ends. Every visited variable reference is checked against the

Original

```
1  on note
2      declare current_note: int := EVENT_NOTE
3      if(EVENT_VELOCITY < 64)
4          play_offset(current_note, maj[1]-1)
5          print_offset(current_note, maj[1]-1)
6      else
7          play_offset(current_note, maj[1])
8      end if
9      play_offset(current_note, maj[2])
10 end on
11 function play_offset(root: int, add: int)
12     declare pitch: int := root + add
13     play_note(pitch, 60, 0, -1)
14 end function
15 function print_offset(root: int, add: int)
16     declare pitch: int := root + add
17     message("Played Note: ", pitch)
18 end function
```

After Parameter Promotion

```
1  on note
2      declare current_note: int := EVENT_NOTE
3      if(EVENT_VELOCITY < 64)
4          declare pitch0: int
5          play_offset(current_note, maj[1]-1, pitch0)
6          declare pitch1: int
7          print_offset(current_note, maj[1]-1, pitch1)
8      else
9          declare pitch0: int
10         play_offset(current_note, maj[1], pitch0)
11     end if
12         declare pitch0: int
13         play_offset(current_note, maj[2], pitch0)
14 end on
15 function play_offset(root: int, add: int, pitch0)
16     pitch0 := root + add
17     play_note(pitch0, 60, 0, -1)
18 end function
19 function print_offset(root: int, add: int, pitch1)
20     pitch1 := root + add
21     message("Played Note: ", pitch1)
22 end function
```

After Variable Reuse

```
1  on note
2      declare current_note: int := EVENT_NOTE
3      if(EVENT_VELOCITY < 64)
4          declare global pitch0: int
5          play_offset(current_note, maj[1]-1, pitch0)
6          pitch0 := 0
7          print_offset(current_note, maj[1]-1, pitch0)
8      else
9          pitch0 := 0
10         play_offset(current_note, maj[1], pitch0)
11     end if
12     pitch0 := 0
13     play_offset(current_note, maj[2], pitch0)
14 end on
15 function play_offset(root: int, add: int, pitch0)
16     pitch0 := root + add
17     play_note(pitch0, 60, 0, -1)
18 end function
19 function print_offset(root: int, add: int, pitch1)
20     pitch1 := root + add
21     message("Played Note: ", pitch1)
22 end function
```

Figure 2. *Left:* Example of a script with functions; *Middle:* Example of a script after *Parameter Promotion*; *Right:* Example of a script after *Parameter Promotion* and *Variable Reuse*

PassiveVarsReplace stack to make sure that its binding to its declaration remains intact. After the entire AST is traversed, all remaining declarations are promoted to the global scope. To ensure that all global variables have distinct identifiers, a global *Gensym* process generates unique names during promotion.

After applying *Variable Reuse*, Listing 1b shows the transformation on the right. All local declarations are converted into assignments, renamed and relocated into the `on init` callback. The local variable `maj_third` (line 8) is replaced by the variable `min_third`, which became passive after the `if` branch was completed. Likewise, the variable `perf_fifth` (line 11) was replaced by the passive variable `min_third`.

3.3 Parameter Promotion

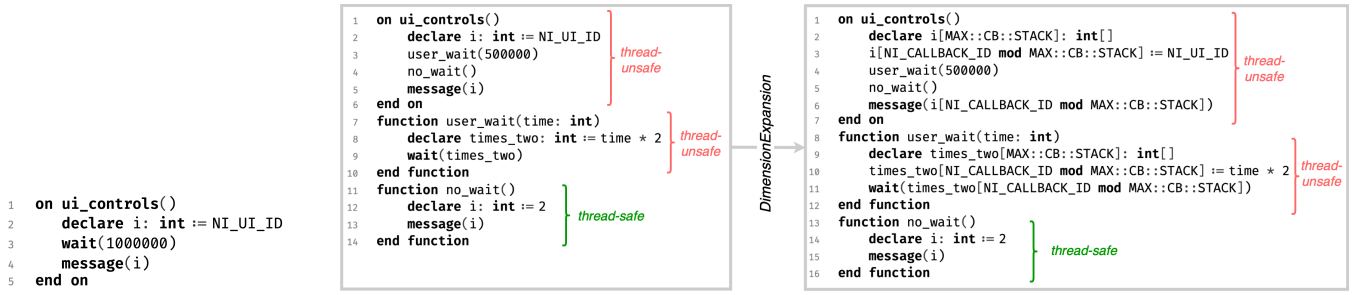
While *Variable Reuse* efficiently reuses *passive* variables, it does not address function-local declarations. Reusing such variables across different functions is infeasible due to varied invocation contexts (callbacks, nested subprograms), even if variables become passive upon function exit. A naive global declaration of all function variables unnecessarily increases the number of declarations. In figure 2 (left) the previous example is now abstracted with `play_offset` while `print_offset` is introduced, outputting debug information about the played offset. Naively promoting `play_offsets` local variable results in one global declaration, but by also applying this to the variable from `print_offset`, we would waste optimization potential.

Instead, the *Parameter Promotion* algorithm promotes function-local variables to parameters, moving their declarations to the call site. At the callback level, a function's local variables become assignments and are added to its formal parameter list. Thus, declarations only occur at callback level, enabling *Variable Reuse*. To mark their limited lifetime, they are placed in an artificial local scope with the call. For nested functions, *Parameter Promotion* causes their variables to be continuously promoted until they reach the callback level (assuming an implementation for parameterized functions as detailed in Section 4, since KSP lacks native support).

During AST traversal, function-local variables are collected in three maps: a map assigning each function its local variables (*localVarDeclarations*), a map storing the variables that have been "promoted" to the callback level (*declaresPerStmt*) and one storing function variables already declared as *global*. When a function call occurs, its definition is visited once, applying *Variable Reuse* to its local declarations, ensuring unique naming. These processed variables are stored in *localVarDeclarations* (with a pointer to the corresponding function) and converted into assignments. After function traversal, they become formal parameters, with references added as actual parameters at call sites. For nested calls this is repeated until the variables reach the callback level, where they are transferred to *declaresPerStmt* assigning them to the corresponding statement. Finally, all those variable declarations are placed in local scopes along with the statement itself, replacing the original call.

Applying this to Figure 2 initially creates more declarations than needed. `pitch0` is inserted above each invocation of `play_offset` and passed as an argument. Its original declaration is converted to an assignment with its *l_value* added to the header. Similarly, the variable in `print_offset` is promoted and renamed (`pitch1`), and inserted above its call. Note that there are two separate local scopes in the `if` branch. The benefit emerges in the subsequent *Variable Reuse* pass. `pitch0` (line 5) quickly becomes passive due to its artificial scope and is reused for `pitch1` (line 7), with similar reuse for declarations in lines 10 and 13, ultimately reducing the example's promoted variables to two.

Consequently, although *ParameterPromotion* requires positioning function-local variables above each call, their placement in artificial local scopes ensures their reuse. Any number of consecutive calls of `play_offset` would result in a single global variable declaration, meaning, that in the



(a) Asynchronous Callback with Local Variables

(b) Example of a script with asynchronous operations and functions; *Left*: Example after *MarkThreadSafe*; *Right*: Example after *DimensionExpansion*

Figure 3. Example of an asynchronous callback (left) and the Dimension Expansion Algorithm (right).

worst case this equals a naive promotion's variable count; in the best case, the promoted variables can be reused.

3.4 Dimension Expansion

While, so far, callbacks have been described and treated as sequential and self-contained, they can be interrupted by asynchronous KSP operations such as `wait`. Here, the program waits for a specified duration, during which another callback may be executed. [16] Variables that were previously set, could be reassigned in an intervening callback, leading to inconsistent variable states upon re-entering the original.

Consider Listing 3a. After *VariableReuse*, `i` would be declared globally. Here, `i` is set to the `UI_ID` of a control. Afterwards, the callback waits for one second before printing `i`. If another UI control is activated during this time, a new instnace of the callback is executed, global `i` is overwritten, and the resumed callback prints the new value rather than the original. To mitigate this, each callback and function is annotated with a "thread safety" property using the *MarkThreadSafe* algorithm. If during its DFS traversal any asynchronous operation is detected within a function call chain, all calling functions and the callback are marked as *thread-unsafe*. However, a *thread-safe* function invoked within a *thread-unsafe* callback stays *thread-safe* (Fig. 3b).

For *thread-unsafe* callbacks, *DimensionExpansion* ensures that local variables remain unique in each callback version. To bind a local scalar variable to a callback, it is transformed into an array where each index represents an execution of the callback. Similarly, local arrays gain an additional dimension. To determine the correct index the *built-in* variable `NI_CALLBACK_ID` (assigning a unique ID to each executed callback) is used. It is automatically incremented with each callback invocation. The algorithm applies $expandDimension(\text{variable}, \text{sizeExpr})$ to add a new dimension of size *sizeExpr* to a given variable. A one-dimensional variable v becomes $v[\text{sizeExpr}]$. If v has d dimensions, a new $(d+1)$-th dimension is added. Variable references are updated accordingly so that an expression like v or $v[i]$ are indexed with *cbIndex*, resulting in $v[\text{cbIndex}]$ or $v[i, \text{cbIndex}]$ where *cbIndex* is calculated as `NI_CALLBACK_ID mod MAX::CB::STACK`. `MAX::CB::STACK` is a global constant representing the maximum number of concurrent callbacks.

The local variable `i` in line 2 (Fig. 3b) is transformed into an array `i[MAX::CB::STACK]`, while its value assignment is moved to line 3. The index uses the current callback ID. The same transformation applies to `times_two` (line 9), with its references converted to array elements indexed by `NI_CALLBACK_ID`.

3.5 Putting It All Together

The transformation from lexical to global scope is achieved by sequentially applying the previously described algorithms to the AST. The *LexicalToGlobalScope* procedure therefore contains the following sequence:

$$BuildLexicalScope \longrightarrow \overset{MarkThreadSafe}{\underset{DimensionExpansion}{}} \longrightarrow ParameterPromotion \longrightarrow VariableReuse$$

It yields a transformed AST_{out}, in which all variable references are linked to their declarations, variable states are consistent, and local variables are efficiently transferred to the `on init` callback.

Due to the addition of inconsistent variable states, *VariableReuse* is extended with additional logic. After the *Dimension Expansion* process, *thread-unsafe* variables have their own value per global callback ID, which allows their reuse in other *thread-unsafe* callbacks with other *thread-unsafe* variables. To enable this, the *computeHash* function is modified to consider type, size and thread safety:

$$\text{computeHashNew}(v) = \langle \text{Type}(v), \text{Size}(v), \text{ThreadSafety}(v) \rangle$$

The *MarkThreadSafe* process supplies the necessary thread safety information, ensuring that *thread-unsafe* variables are only replaced by similarly marked ones, while *thread-safe* variables remain eligible for reuse even within *thread-unsafe* callbacks.

4 Implementation of Functions

Functions, or *subprograms*, are key abstraction mechanisms that structure and reuse code by encapsulating complex

```
on note
    declare pitch: int := EVENT_NOTE
    declare velo: int := EVENT_VELOCITY
    play_note(pitch, clamp(velo, 90), 0, -1)
end on
function clamp(value: int, max: int): int
    declare final: int := value
    if (value < 0)
        final := 0
    else if (value > max)
        final := max
    end if
    return final
end function
```

(a) Example of a function with parameters and return values.

```
function search_matrix(matrix: int[][],target: int)
    declare row, col: int
    while(row < num_elements(matrix, 1))
        while(col < num_elements(matrix, 2))
            if matrix[row,col] = target
                return (row * num_elements(matrix,1))+col
            end if
            inc(col)
        end while
        inc(row)
    end while
    return -1
end function
```

RewriteReturnNodes

```
function search_matrix(matrix: int[][], target: int)
    declare RETURN_FLAG: int
    while(RETURN_FLAG = 0)
        declare row, col: int
        while(row<num_elements(matrix,1) and RETURN_FLAG=0)
            while(col<num_elements(matrix,2) and RETURN_FLAG=0)
                if matrix[row,col] = target
                    return (row*num_elements(matrix,1)) +col
                    RETURN_FLAG := 1
                    continue
                end if
                inc(col)
            end while
            if(RETURN_FLAG = 1)
                continue
            end if
            inc(row)
        end while
        if(RETURN_FLAG = 1)
            continue
        end if
        return -1  // If the target is not found
        RETURN_FLAG := 1
        continue
    end while
end function
```

(b) The *Rewrite Return Notes* Algorithm; *Left*: A matrix search algorithm with multiple return statements. *Right*: The transformed example.

Figure 4. Combined figure showing the function example (left) and the return rewrite algorithm (right).

computations into clearly defined, parameterized units. An essential feature is their ability to receive input parameters and return computed results via return values [23]. In contrast, KSP's native function syntax is highly restrictive. They are defined globally by enclosing a sequence of statements between `function` and `end function` keywords, and invoked using `call <func_name>`. However, they lack support for function-local variables, parameters, return values, or early termination via `return` statements. Additionally, they cannot be invoked within expressions or passed as arguments and must be wrapped in standalone statements. They are prohibited in the `on init` callback and certain KSP operations are not permitted in native function bodies limiting their usability to only marginally improving code structuring. To overcome these limitations, we have devised several transformations.

4.1 Syntax and Representation in the AST

To represent the properties of subprograms in the surface language and the AST, we used a slightly modified syntax of native KSP functions, adding a parenthesized, type-annotated parameter list to the header. The function body may now contain arbitrary statements, and permit `return` nodes to terminate execution prematurely and yield a value. Moreover, functions can be invoked in an expression context.

Listing 4a illustrates a `clamp` function restricting an *integer* input to the range $[0,\ max]$. The local variable `final` is used to store the input value and gets returned. The function is used without the restrictions of native KSP functions by emplying it directly as an argument in the built-in `play_note` function.

The essential constructs are divided into two categories: the subprogram definition and the subprogram call [23]. Accordingly, two AST nodes are introduced: *FunctionDefNode* and *FunctionCallNode*. The former is defined in the global scope alongside callbacks and stored in the root node (*ProgramNode*), with a lookup table updated for quick retrieval by name and parameter count. Each definition contains a *FunctionHeaderNode* speficying its name, an ordered list of *formal parameters* and the return type. The function body is a *BlockNode*, containing a sequence of statements and optionally *ReturnNodes*. Each *FunctionCallNode* contains a *FunctionHeaderRefNode* with its name and a list of *actual parameters* (expressions). Due to this node being a reference, this design supports higher-order functions. Each function call binds to a specific function definition in the root node based on name and argument count.

4.2 Rewriting Return Statements

Before transforming functions into valid KSP code, we describe the implementation of `return` statements for value-returning and premature termination. In Listing 4a, the `return` statement marks the end of the function control flow and one might simply assign its value to a temporary variable. However, in nested conditionals this approach fails since control may continue past the intended exit point.

While a `break` statement would normally serve to interrupt execution by exiting a loop, KSP lacks `break` support. Instead, the *RewriteReturnNodes* algorithm repurposes `continue` for termination upon encountering a `return`. Therefore, it wraps the function body in a `while` loop controlled by a `RETURN_FLAG`. Upon encountering `return`, the algorithm inserts `RETURN_FLAG := 1` followed by `continue`, forcing the loop condition to be re-checked. As `continue` only affects the innermost loop, each nested loops' condition is extended to check `RETURN_FLAG = 0`, with post-loop checks to propagate exiting. Figure 4b illustrates this with a matrix search: Upon finding the target, the inner loop is exited, and an immediate check after each nested loop triggers a `continue` to bypass any further statements (e.g., `inc(row)`). Conversely, functions with a single, final `return` (e.g., `clamp`) require no such transformation.

Critically, every control path must include a `return`; otherwise, `RETURN_FLAG` remains `0`, trapping the function in an infinite loop and causing a KSP runtime error.

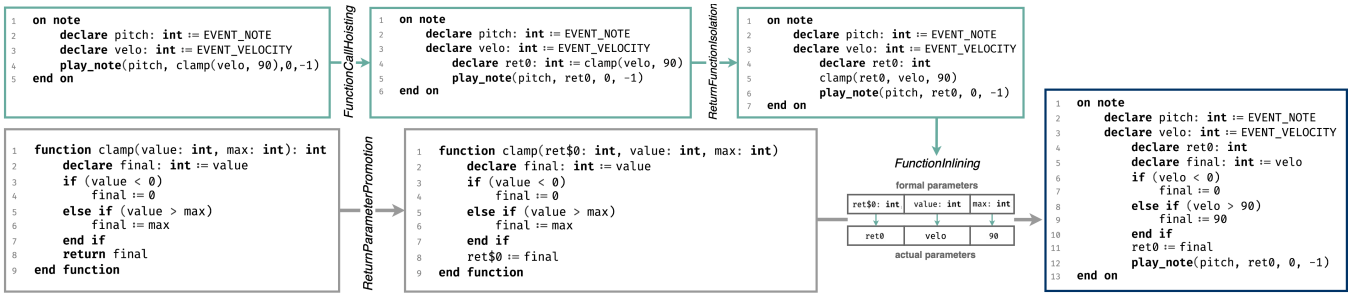


Figure 5. Function Isolation Transformations with the *clamp* function example.

4.3 Function Isolation

To support return statements and allow embedded calls in expressions, the functions are isolated by converting the latter into standalone statements. At the same time, the definitions have to be transformed accordingly. Figure 5 illustrates the necessary transformation steps for the `clamp` function.

Since isolated calls cannot directly yield return values, the `return` keyword is rendered ineffective. *Return Parameter Promotion* converts `return` statements into assignments to a temporary return variable, which is then promoted to a formal parameter, effectively replacing the conventional return mechanism with an argument-based value transmission. This pass only needs to apply to functions containing `return` statements; void return statements are removed. In figure 5 (lower left) the single *ReturnNode* is converted into an *AssignmentNode* where the left-hand side is a new variable `ret$0`. It obtains its type from the function's return type and works as a temporary return variable that gets assigned the respective value of every `return` statement. Its uniqueness is ensured by the `$` character, which is disallowed for user-defined variable names. `ret$0` is then inserted at the beginning of the function's parameter list.

While function definitions have been isolated, function calls with return values can appear as *r_values* in declarations or assignments, or as part of expressions (e.g., conditional expressions or function arguments). To consolidate the two distinctions into a single form, *FunctionCallHoisting* extracts embedded function calls from expressions and hoists them into standalone declarations. The transformation creates a temporary variable (with a unique name via *Gensym*) to store the return value. The original call site is then replaced by a reference to this variable. The semantics of the program remain largely unchanged, since the function execution site is the immediately preceding statement and the order of execution is preserved (from left to right to top to bottom) with the caveat of missing *short-circuit* evaluation. However, when hoisting from `while`-loop conditions, functions with side effects can return different results across iterations. To mitigate this without resorting to complex static analysis, *FunctionCallHoisting* reinserts the function call inside the loop. An assignment of the temporary variable and the function call is appended at the end of the loop body, ensuring its re-evaluation before each condition check.

Figure 5 (lower left) shows the `clamp` function call embedded as an argument. The algorithm creates a new unique variable `ret0` assigning it to the call. Afterwards, the original call site is replaced by a reference to `ret0`. To facilitate *Variable Reuse*, its short lifetime is marked by an artificial scope containing the entire statement and the declaration.

Finally, the *ReturnFunctionIsolation* process converts function calls in assignments or declarations into standalone statements so that the variables storing the return values are passed as parameters. For assignments, the *l_value* is passed as the first argument to the function call, effectively replacing the assignment. For declarations, the assignment is removed from the declaration and inserted as a separate call, with a reference to the declared variable as a parameter. This guarantees that every function call operates as an independent statement with its return value passed explicitly through a parameter.

This is shown in the upper right of figure 5. The call is removed from the declaration (line 4) and inserted below as an isolated call. Subsequently, the variable `ret0` is passed as the first parameter and the declaration of `ret0` loses its value assignment. With this, the process of isolating function calls is complete.

4.4 Function Inlining

After the preceding compiler passes, all function calls appear as isolated statements, and function definitions have been transformed such that they accept their return value as a parameter. The functions now resemble native KSP functions in that they no longer contain return statements and are invoked only as standalone statements. For some functions (without parameters, not called in `on init`) this is sufficient as they can now be treated as native KSP functions. However, for functions with parameters, the *FunctionInlining* process is required to replace the function calls with their corresponding function bodies and statically substitute formal parameters with their actual arguments.

Conventional parameter passing techniques include *pass-by-value* (copying the actual parameter value) and *pass-by-reference* (directly accessing the original parameter without copying). However, since KSP lacks parameter support, all parameter bindings and evaluations must occur at compile time. Thus, the substitution follows a *pass-by-name* approach—textually substituting the actual parameter for the formal one when it is used while preserving type information for compile-time type checking.

The *FunctionInlining* process accepts pairs of function definitions and calls returning a $Block_{out}$ where the call has been replaced by the function definition statements. During inlining, the algorithm stores the arguments of each function call, associating them with the formal parameters by their order. Subsequently, the definition is visited, copied, and, within this copy, every reference to a formal parameter is substituted with its argument. Figure 5 shows this process on the far right, where a mapping of the formal and actual parameters is created. The function body is copied and variables—such as the occurrence of `value` (replaced by `velo`), `max` in an `else if` condition, and finally the return parameter `ret$0`—are substituted. Finally, the block is inserted into the original code, replacing the function call.

Since function bodies may themselves contain other function calls, directly inlining every call could lead to redundant processing. For this reason, a separate compiler pass, *ASTFunctionInlining*, performs a DFS traversal of the AST ensuring that deeply nested function calls are inlined first, followed by the next higher ones. The process checks if a *FunctionCallNodes* has already been visited and if not, proceeds to visit its definition node. If it was visited, all necessary function calls in that AST branch have been already visited triggering its inlining. With this strategy, each function call is processed only once, keeping the overall time complexity linear to the AST size ($O(n)$).

Smaller functions can be inlined more efficiently, without the need to declare additional return variables, assignments or use function isolation transformations. For example, the recurring `clamp` function can also be represented in one line:

$$\text{clamp}(x, M) = \max(0, \min(x, M)) = \tfrac{1}{4}\Big((x + M - |x - M|) + \big|x + M - |x - M|\big|\Big)$$

This formula avoids the need to isolate the function. A more efficient method is to replace the function call directly where it is invoked, saving two lines of KSP code. We define *Expression-Only* functions as those whose body consists solely of a single *ReturnNode*. For these functions, inlining simply involves substituting the parameters and inserting the return expression at the call site. In our implementation, a dedicated process identifies all *Expression-Only* functions and replaces them with their return value by calling *FunctionInlining* prior to further compilation steps.

Note. This implementation currently mirrors native KSP by not supporting recursion. Under some circumstances, the function isolation pipeline already uses native KSP function syntax for parameterless functions, skipping the inlining step. Future work may integrate KSP's native "`call`" mechanism with a parameter stack for *pass-by-value* semantics, potentially reducing code bloat and compilation times.

5 Implementation of Recursive Data Types

KSP provides three primitive data types: *Integer*, *Real*, and *String*. However, this limited type set complicates the representation of more complex data models, as even multidimensional arrays must be flattened to one dimension to meet the requirements of the KSP interpreter. In contrast, most GPLs offer the ability to define custom data types (often called "records", "structures" or "structs") containing multiple fields of different types and group them under a common name. Unlike arrays, where elements are referenced by indices, record fields are addressed by named identifiers and can have different types [21].

The objective is to enable the definition of abstract data types with multiple fields, including the possibility for recursive references, while also supporting methods operating on the data, instance creation, and access chains. We explore the implementation of automatic memory management, enabling dynamic creation and deletion of instances—all while keeping the syntax close to existing KSP conventions incorporating proven concepts from established languages.

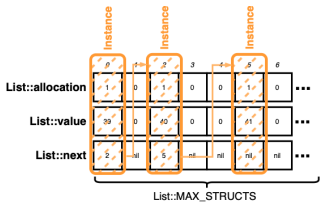
5.1 Syntax and Basic Concept

Records are defined between `struct` and `end struct` keywords, with field declarations performed using KSP's usual `declare` syntax. Fields may include variables, arrays, and multidimensional arrays as primitive or abstract data types. To treat records as first-class types, the type system is extended with *Object* types (representing structs) registered during parsing. Methods can be defined in a *Python*-like syntax, where each one must include `self` to refer to the current object, while the constructor is named `__init__` and is automatically invoked upon instance creation. Struct instances are referenced by a dedicated pointer variable of type *Object* and created by invoking the struct name as a function, with field values passed as arguments. If no constructor is defined, a default constructor is provided, which takes arguments for all fields in their declared order. Uninitialized pointer variables default to `nil`, which gets lowered to `-1`. Field and method access is performed using the dot operator.

Dynamic allocation of struct instances is simulated by implementing a "heap" using arrays, where each index acts as a memory address holding one struct instance. Figure 6 illustrates this using a recursive list with two fields: `value` representing the value of the current element, and `next`, a pointer of type *Object* referring to the next list element. The fields are transformed into the *heap* arrays of size

```
1  struct List
2      declare value: int
3      declare next: List
4
5      function __init__(self, value: int, next: List)
6          self.value := value
7          self.next := next
8      end function
9  end struct
```

(a) An example of a recursive struct definition.



(b) The simulated "heap" arrays.

```
1  declare this_list: List := List(42, nil)
2  declare that_list: List
3  that_list := List(39, List(40, List(41, this_list)))
4  message(that_list.next.value)
```

(c) Instantiating a struct and accessing its fields.

Figure 6. Implementation of Abstract Data Types on the example of a recursive list.

`List::MAX_STRUCTS`, where a `List` instance is composed of all elements at a specific index across the arrays. Each value in the `List::next` array refers to the index of the next instance. In figure 6b, the element at index `0` points to index 2 (holding the value `40`), while `List::next[2]` points to index 5, enabling recursive structures.

Allocation is handled by the constructor (Fig. 6c, right) locating free memory in the allocation array and storing the field values in the corresponding arrays. To do that, it searches for the first index with value `0` (using the KSP command `search`). If no free memory is found, `List::MAX_STRUCTS` instances already exist and an error message is displayed. Otherwise, the free index is assigned to `List::free_idx` and, after storing the field values, returned. Figure 6c shows an assignment where the pointer `this_list` is initialized with a `List` instance of value `42`. The constructor sets the pointer to the index where the instance is stored. Connected data structures can be formed by passing constructors as arguments to other constructors (line 3). Access to nested objects is achieved by transforming fields in between the dot operators into arrays and also nesting them.

At the AST level, two key transformations are applied: transforming abstract data type interfaces (represented by *StructNodes*) and access chains (represented by *AccessChainNodes*). A *StructNode* encapsulates the type name, an ordered list of *DeclarationNodes* for fields, and a list of *FunctionDefNodes* for methods, along with member and method tables for fast lookup. Each *AccessChainNode* abstracts a sequence of fields and methods delimited by the dot operator, storing the individual elements as an ordered list of *ReferenceNodes* or *FunctionCallNodes*.

5.2 *StructNode* Transformations

The transformation of *StructNodes* is achieved through a compilation pipeline as illustrated in Figure 7 using the *List* struct. In the first stage, *DesugarStruct* restructures the AST in a way that fields and their references can be correctly linked via *BuildLexicalScope*. The pass verifies that all methods are defined within the struct namespace, each method contains `self` as the first parameter and that only field references begin with "`self.`". It also checks for a user-defined constructor, automatically generates one if absent. To directly link fields and their references by common names, fields are prefixed (e.g., `List::`), just as the struct methods, to make them unique in the global function scope. Within the struct scope, potential ambiguities between access chains (e.g., `self.next.next.value`) and simple field references (e.g., `self.value`) are resolved via the member table. If a name appears in the table, the reference is maintained as a *ReferenceNode* and is prefixed `List::`; otherwise, it is converted into an *AccessChainNode*. The usage of `::` can prevent name clashing since the colon is not recognized by the tokenizer. The `self` parameter is removed from the constructor, aligning the number of parameters in constructor calls with their definitions.

Next, the *PreLoweringStruct* pass extends the transformed *StructNode* with compiler-specific variables. These additions include the `allocation` array, the `free_idx` variable, as well as data structures used for memory management (see Section 5.4). The transformed `List` definition in Figure 7 illustrates these modifications. If a struct contains no array fields, the maximum number of allocatable struct instances (`MAX_STRUCTS`) can be set directly. However, when arrays are present, their sizes must be taken into account, with the largest array's size serving as the divisor in the calculation. Consider a struct with two array fields: one with size n, the other with size $n-1$. By the end, all fields are extended by the heap dimension m necessitating that all dimensions of a field array combined do not exceed `MAX_STRUCTS` (M). To achieve this, *PreLoweringStruct* defines a *max* function to determine the largest field array and compute the value of m via integer division, calling *max* in a nested manner if more than two arrays exist. Here, this leads to $m = M/\max(n, n-1)$ allocatable instances.

The *LoweringStruct* algorithm eliminates *StructNodes* from the AST by converting struct fields and their references into higher-dimensional arrays, moving methods to the global function scope, and adjusting constructors for proper memory allocation. This is achieved by applying *DimensionExpansion* (Section 3.4), adding an extra *heap* dimension. For example, a scalar field A is transformed into $A[m]$, a one-dimensional array $B[S]$ becomes $B[m, S]$, and an n-dimensional field C becomes $C[m, S_1, \ldots, S_n]$, with m representing the allocatable instances. Field references

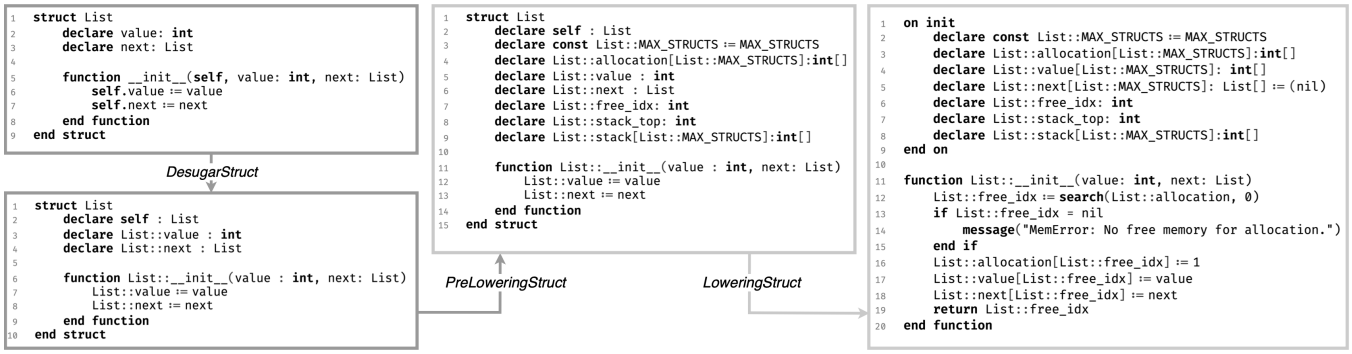


Figure 7. The transformation processes of a *StructNode* illustrated using the example of a *List* struct. Depicting the original struct definition, the struct after *DesugarStruct*, after *PreLoweringStruct* and after *LoweringStruct*.

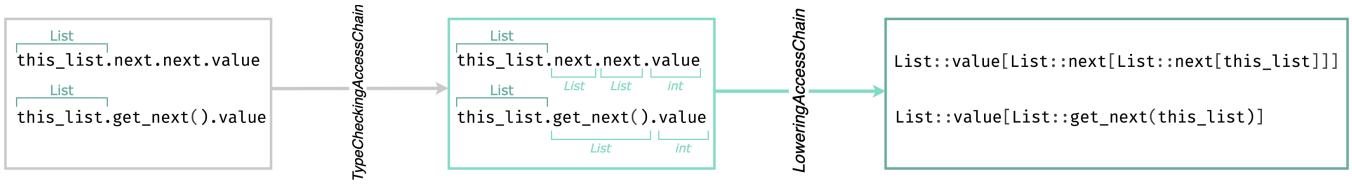


Figure 8. Transformation processes of an *AccessChainNode* illustrated using the example of a *List* struct.

outside the constructor are expanded by one dimension and indexed using the struct's `self` variable, those inside the constructor are indexed with `List::free_idx`. The constructor is transformed and the search function, the subsequent `if` statement, and the setting of the allocation array are added. Ultimately, all methods are moved into global scope and the remaining arrays and constants to the `on init` callback.

5.3 *AccessChainNode* Transformation

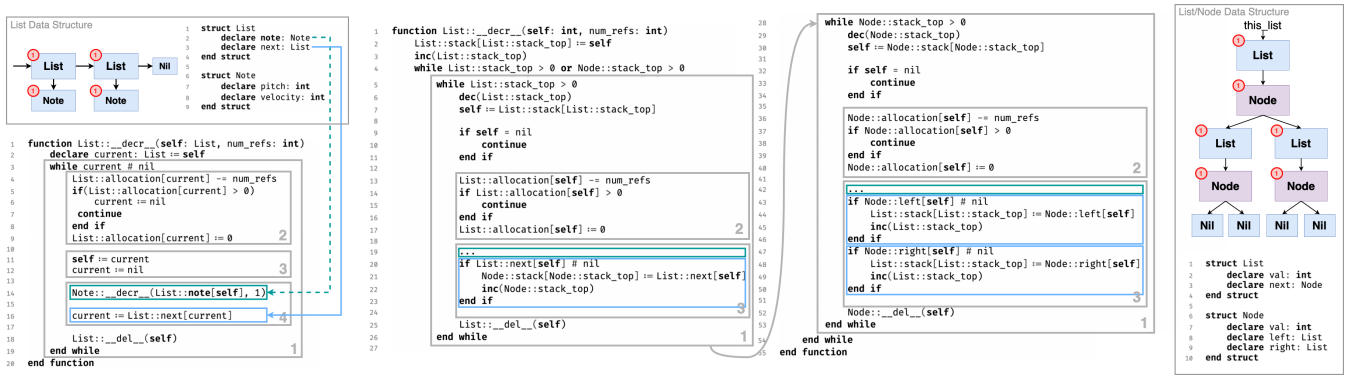
Since some legacy codebases might use the dot operator as part of a naming convention, access chains cannot be constructed during parsing. Instead, during *BuildLexicalScope*, variable references or function call nodes containing dot operators are inspected. If the full identifier is not declared as a variable, it is split into parts and converted into an *AccessChainNode*. Nested access chain nodes are recursively flattened into a single, linear structure for further processing. The following transformation steps are shown in figure 8.

Assuming all variable declarations and function parameters are annotated, type verification uses the declaration pointers gathered during *BuildLexicalScope*. Note that only the first member of an access chain carries a declaration pointer and therefore type information. *TypeCheckingAccessChain* returns an $AccessChainNode_{out}$ with complete type details by iteratively taking the type of the preceding member, getting the member table of the object, and inferring the type of the next member from it. If a member is missing or if a primitive type was used, an *UndeclaredFieldException* is thrown. For function call members, the struct's method table is searched and the method's return type used for further inference. The algorithm terminates once the end of the access chain is reached.

The final lowering phase converts an $AccessChainNode_{in}$ into a nested structure of arrays or function calls. First, all pointer and array references are extended by one dimension and prefixed with the type of their predecessor. Consider figure 8 where `this_list` is the first element. It remains unchanged since it refers to a pointer variable in the same scope. The next member (`next`) is prefixed with the type of `this_list` becoming `List::next[null]`. This process continues along the chain. Subsequently, the nesting is performed by an additional iteration, where the previous member always serves as the index or argument (for method calls) for the next member. For instance, the access chain $A.B.C$ is transformed into $C[B[A]]$. If a field is already an array, it is similarly extended by one dimension, with the previous node inserted as an additional index, so that $A.B[42].C$ becomes $C[B[A, 42]]$.

5.4 Reference Counting

To enable automatic memory management, we implement a *reference counting* strategy that tracks and deallocates unreachable objects in real time. Traditional garbage collection mechanisms like *Mark-Sweep* are unsuitable for our use case due to their *stop-the-world* nature [9][10]. Instead, continuous deallocation distributes memory management costs throughout the runtime [10]. We track reference counts via the existing allocation arrays. When an object is created, its allocation index is set to 1. Any subsequent references to



(a) Linear Direct Recursion using *List* as an Example **(b)** Non-Linear Indirect Recursion using *List* and *Node* as an Example, *Left*: Decrease Method of *List*, *Right*: List ADT Example and directed graph of a *List* Instance.

Figure 9. Examples of different Decrease Methods for recursive data types

that object increment the count by `1`, removing a reference decrements it. Once a count hits `0`, the object is deallocated; its directly linked objects then undergo the same decrement, potentially triggering a cascading deallocation. Reference counting methods are generated for each struct. The `__incr__` method checks that the object's index is not `nil` before incrementing its count, while `__decr__` decrements it and calls the `__del__` method to reset the object's fields when the count drops to `0`. Because KSP does not support recursive functions, recursive deallocation must be implemented iteratively. We distinguish among three recursion types: (1) *Linear Direct Recursion* applies to structures with at most one recursive field and can be handled with a simple `while` loop. It is suitable for non-recursive structs as well. (2) *Non-linear Direct Recursion* involves multiple recursive fields pointing to the same type and requires a stack; and (3) *Non-linear Indirect Recursion* involves recursive fields of different types and requires stacks for all involved structs along with nested loops.

To determine the recursion type, a cycle detection algorithm traverses all structs reachable from a given *StructNode* via DFS, gathering all reached recursive data types into a set. An empty set indicates a non-recursive object, a singleton set implies linear direct or non-linear direct recursion, and a set with multiple elements indicates non-linear indirect recursion.

Figure 9 illustrates exemplary *Decrease* Methods for two of the recursion types, with their basic architecture annotated by numbered blocks.

Consider the *Linear Direct Recursion* case using the `List` struct, where each element holds a non-recursive `Note` object (with `pitch` and `velocity`), so no additional data structure (e.g. stacks) is needed when iteratively traversing (Fig. 9a). Block 1 shows the main `while` loop iterating until `current` becomes `nil`–with `current` holding the value of the next field during a cascade of deallocations. Block 2 decrements the reference count of the current object; if the count remains above zero, `current` is set to `nil` and the loop iteration is terminated immediately. In block 3, the current object is temporarily stored in `self` before `current` is reset, ensuring that the `Delete` method can later be applied to `self` even if `current` changes. Block 4 processes the struct's fields: non-recursive fields (such as `note`) are deallocated via their own *Decrease* methods, while recursive fields update `current` to continue the traversal. The explicit assignment of `current` to `nil` guarantees termination if no new value is assigned, and the subsequent call to `Delete` resets all fields to prevent infinite loops in cyclic structures.

For *Non-linear Indirect Recursion*, where multiple data structures form a cycle, the compiler generates a more complex approach. Each participating struct is assigned its own stack, and a central loop coordinates these stacks by iterating as long as at least one is non-empty. For instance, as illustrated in Figure 9b, a `List` struct may reference a `Node` via its `next` field, while the `Node` in turn references a `List`. The two inner loops (Block 1) iterate over their respective stacks, decrementing reference counts (Block 2) and pushing non-`nil` recursive fields onto the stacks (Block 3). Afterwards, the `Delete` method is called to reset the current instance. Iteration continues until the respective stack is empty. The central loop, which combines the stack status, ensures that both stacks are processed alternately. This mutual handoff guarantees that all objects in the cycle are traversed and deallocated without leaving any unresolved references. With the *Non-linear Direct Recursion* case, just one stack needs to be iterated over.

To determine when to invoke reference counting methods, we introduce the *PointerScope* algorithm. It inserts *IncrRefNode* and *DecrRefNode* nodes into the *AST*. Specifically, when a pointer variable (of type *Object*) is assigned a new reference via an *AssignmentNode* or *DeclarationNode*, the reference count is increased—unless the *r_value* is `nil` or

a constructor (which already sets up the allocation). Before reassigning a pointer variable, the count is decreased (with a `nil` check). When a local pointer variable's scope ends, a *Decrease* call is appended, by a stack-frame tracking pointer variables, similar to the *BuildLexicalScope* approach.

However, temporary objects (created within access chains or as function call arguments) are not stored in pointer variables. Although their constructors set the reference count to `1`, no corresponding *Decrease* is applied under the current implementation. After *FunctionCallHoisting*, these temporary objects are assigned to variables, but since the lowering phase already converted all *Object* types to *Integer* types, the *PointerScope* adjustments no longer apply. For this reason, temporary constructors are marked previously by checking if the constructor appears under the aforementioned conditions. Immediately after hoisting, the *TemporaryPointerScope* algorithm traverses the *AST* to insert *Decrease* calls for the marked objects operating similarly to *PointerScope* by maintaining a stack of variables.

5.5 ADTs, Asynchronous Operations & Cyclic References

As previously shown, KSP variable consistency is not guaranteed when asynchronous operations trigger additional callbacks during waiting periods. This was resolved for local variables via *Dimension Expansion*. Functions that do not perform asynchronous operations are inherently *threadsafe*, meaning that local variables and all automatically generated reference counting methods remain consistent.

Even though constructors can be auto-generated, user-defined constructors may contain asynchronous operations, which can lead to errors when returning memory addresses. For example, if a `wait` command in a constructor triggers a callback before it completes, the global `free_idx` may be overwritten, causing loss of the first object's memory address. Thus, constructors must not include asynchronous operations; for any such usage, we raise an error.

Global pointer variables present a similar risk. Since they are not subject to *Dimension Expansion*, assigning a global pointer within an asynchronous callback can lead to incorrect reference counts. For instance, if a global pointer is assigned after a `wait` command, the preceding *Decrease* method might operate on a new object rather than the original, leaving its reference count unchanged and preventing deallocation. Prohibiting asynchronous assignments to global pointers is challenging without sacrificing flexibility; therefore, we issue a warning in this case recommending using local pointers instead.

Manual memory management in our system addresses cyclic reference issues that pure reference counting cannot resolve. Rather than implementing a hybrid garbage collector—which would add unwanted overhead—we support manual deallocation using the reserved word `delete`. When a `delete` statement is encountered, it is transformed into a call to the object's *Decrease* method (with a `nil` check and its full reference count as argument), and the pointer is removed from the *PointerScope* stack frame to prevent duplicate deallocation.

6 Conclusion and Future Work

This paper presented transformations that extend the Kontakt Script Processor with higher-level abstractions. Key contributions include preserving variable state, restructuring variable scoping by reusing local variables, a function transformation pipeline allowing for parameterized functions with return values and an abstraction mechanism for modeling recursive data structures with reference counting. These solutions have been integrated into a preprocessing compiler framework that produces efficient KSP code.

Our experience confirms that even a severely restricted DSL can be augmented with modern programming abstractions by tailoring transformations to its specific idiosyncrasies and investing significant engineering effort.

While formal user studies are pending, initial feedback from KSP developers using early versions of the compiler has been positive, particularly emphasizing the enhanced codebase maintainability. Local variables, conveniently declarable at their point of use, reportedly improved code organization and readability. *Variable Reuse* further encouraged their liberal adoption by mitigating concerns about generating an excessive number of global declarations. The ability to declare variables locally within each scope resolved prior issues, such as conflicting iterator variables (e.g., `i`) in nested loops or functions and worrying about inadvertent variable name interference across scopes, thereby facilitating management of large projects. Parameterized functions were also lauded for their contribution to modularity, with their direct usability in expressions simplifying code and boosting reusability.

Nonetheless, limitations remain. Future work will focus on optimizing memory usage and reducing code overhead through refined static analyses (for lifetime estimation and heap dimensioning) and closer integration of native KSP functions via a parameter stack. Further investigation will target more flexible memory management to minimize manual deallocation and explore partial recursion support to simplify data structure traversal (currently only possible iteratively). Finally, performance studies and broader community testing are essential to evaluate the practical impact of these abstractions, their real-world adoption, and the overall interest in deviating from standard KSP scripting practices.

References

[1] Saleem Abdeen. 2023. The Future of Programming: Embracing Polyglotism and Domain-Specific Languages. Medium article. https://medium.com/@saleemabdeen17498/the-future-of-programming-embracing-polyglotism-and-domain-specific-

languages-402ddbdf6258 Accessed: 2025-01-24.
[2] Jim Aikin. 2003. Sample Solutions. *Electronic Musician* 19, 13 (December 2003), 74–77.
[3] Andrew W. Appel and Maia Ginsburg. 1998. *Modern Compiler Implementation in C*. Cambridge University Press, Cambridge, UK, Chapter 6.1 Stack Frames, 127.
[4] VI Control Forum Comments by EvilDragon. 2023. KSP Discussion and Kontakt Market Reach. Online forum post. https://vi-control.net/community/threads/ksp-is-hard.86988/post-5052230 Accessed: 2025-01-24.
[5] VI Control Forum Comments by EvilDragon. 2025. KSP Run in Realtime Audio Thread. Online forum post. https://vi-control.net/community/threads/advantages-of-new-komplete-ui-over-traditional-ksp-scripting.155936/page-3#post-5627959 Accessed: 2025-01-24.
[6] Josh Davies. 2020. Examining the Role of Orchestral Sample Libraries in Modern Production. https://happymag.tv/examining-the-role-of-orchestral-sample-libraries-in-modern-production/ Accessed: 2025-01-26.
[7] VI Control Forum Discussion. 2022. Discussion about Kontakt being the Industry Standard. Online forum post. https://vi-control.net/community/threads/kontakt-is-the-protools-of-the-composing-industry.125240/ Accessed: 2025-01-24.
[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. Visitor. In *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 331–344.
[9] Richard Jones, Antony Hosking, and Eliot Moss. 2012. *Mark-sweep garbage collection* (1st ed.). Chapman and Hall/CRC, Boca Raton, FL, USA, Chapter 2, 17–30.
[10] Richard Jones, Antony Hosking, and Eliot Moss. 2012. *Reference counting* (1st ed.). Chapman and Hall/CRC, Boca Raton, FL, USA, Chapter 5, 57–74.
[11] Nicki Marinic, Josef Natterer, and Wolfgang Schneider. 2006. *The Kontakt Scripting Language*. Native Instruments Software Synthesis GmbH. Manual.
[12] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *Comput. Surveys* 37, 4 (2005), 316–344. https://doi.org/10.1145/1118890.1118892
[13] John C. Mitchell. 2002. *Concepts in Programming Languages*. Cambridge University Press, Cambridge, UK, Chapter 7, 162–203.
[14] Meinard Müller. 2021. MIDI Representations. In *Fundamentals of Music Processing: Using Python and Jupyter Notebooks* (second edition ed.). Springer, Germany, 13–15. https://doi.org/10.1007/978-3-030-69808-9
[15] Native Instruments. 2024. *KSP Reference Manual*. Native Instruments. https://www.native-instruments.com/ni-tech-manuals/ksp-manual/en/ Accessed: 2024-12-30.
[16] Native Instruments. 2024. *KSP Reference Manual - Time-related Commands*. Native Instruments. https://www.native-instruments.com/ni-tech-manuals/ksp-manual/en/time-related-commands Accessed: 2024-12-30.
[17] Native Instruments. 2024. *KSP Reference Manual - Variables*. Native Instruments. https://www.native-instruments.com/ni-tech-manuals/ksp-manual/en/variables Accessed: 2024-12-30.
[18] Native Instruments. 2024. What is Kontakt? https://blog.native-instruments.com/what-is-kontakt/
[19] Mike Novy. 2010. *KSP Scripting 1: Understanding and Developing NI Kontakt Scripts*. CW Music, Germany. Comprehensive guide on Kontakt Script Processor (KSP) scripting.
[20] Len Sasso. 2008. Scripting in Kontakt 3. *Electronic Musician* 24, 2 (February 2008), 42–46.
[21] Robert W. Sebesta. 2012. Records. In *Concepts of Programming Languages* (10th ed.). Pearson, Boston, MA, 276–279.
[22] Robert W. Sebesta. 2012. Scope. In *Concepts of Programming Languages* (10th ed.). Pearson, Boston, MA, 218–229.
[23] Robert W. Sebesta. 2012. Subprograms. In *Concepts of Programming Languages* (10th ed.). Pearson, Boston, MA, 388–421.
[24] Soundfly. 2023. A Brief Introduction to Samplers. https://flypaper.soundfly.com/produce/a-brief-introduction-to-samplers/ Accessed: 2025-01-25.
[25] UJAM. 2023. Comprehensive Guide to Virtual Instruments. https://www.ujam.com/tutorials/comprehensive-guide-to-virtual-instruments/?srsltid=AfmBOopGqzV5-3C13FS5IQDMd3HRypdGXf6hYW4x8gHUnzcpwmbbIqNf Accessed: 2025-01-25.
[26] Ingo Weisemöller. 2012. *Generierung domänenspezifischer Transformationssprachen [Generation of Domain-Specific Transformation Languages]*. Dissertation. RWTH Aachen University, Aachen, Germany. http://www.se-rwth.de/publications/Generierung-domaenenspezifischer-Transformationssprachen.pdf
[27] G. Yelton. 2018. Battle of the Super Samplers: Three major sampling platforms duke it out for software supremacy. *Electronic Musician* 34 (2018), 16–20.
[28] YummyBeats. 2019. KSP Scripting (NI Kontakt) :: BASICS :: What's KSP - is it easy to learn? https://blog.yummybeats.com/ksp-kontakt-scripting/ksp-scripting-ni-kontakt-basics-ksp-is-it-easy-to-learn/