Dear Prof. Donald Knuth,

We are sincerely thankful to you for your time and previous valuable detailed reply to our letter and gifting it to the IT history museum.

Thank you for your attention to the uncovered Address programming language capabilities.

According to your valuable comments, we add a more detailed description of existing practical implementations.

From personal memories, of Mr. Yurii Yushchenko:

"I remind myself how in 1980-s together with mother Kateryna Logvynivna we wrote in "Address programming language 1955" certain algorithms described in Volume 1. Chapter 2 – Information Structures of your great work "The Art of Computer Programming". It was very interesting to observe mother's thinking and work and a potential embedded in the Address programming language ".

In order to popularize unknown capacities of the Address programming language regarding representation of tree-like formats and their processing, a group of Ukrainian enthusiasts conducted a number of researches: they adapted the Address programming language (ADPL) syntax to the modern notation, realized language compiler and ran a number of cases demonstrating:

1) Indirect addressing of rank 2 (Pointers).
2) Properties and capabilities of the "'" operation (in modern terms: pointers dereferencing operation).
3) Indirect addressing of higher ranks, which is provided by multiple application of a "'" operation.
4) The availability of list chains and full Address programming language support for their processing.
5) We demonstrate the language capabilities using examples of programs for representation and processing of binary trees and complex hierarchical structures in Address programming language.
6) The capability of the Address programming language in writing programs using the OOP paradigm.
7) Other interesting and powerful capabilities of the Address programming language.

A. The adaptation of the Address programming language, its realization and examples.

1. The adapted syntax (Karyna Cherednyk)

First, she adapted the original Address programming language for enablement of composition and editing of initial program texts in modern text editors.

The url for the description of the adapted syntax and and the description of its C++ realization.: https://ekmair.ukma.edu.ua/items/edcb0f15-2627-4e6f-ba57-9b51cbf2ec1b.

2. C++ interpreter realization (Karyna Cherednyk)

Next step was the C++ language interpreter for the base part of the Address programming language. Examples of programs demonstrating Pointers operations are provided. As well a number of other simple programs are built and debugged.

URL for C++ code of realization.

https://github.com/karina-cherednyk/AddressProgrammingLanguage

2.1. The content of repository:

2.1.1. VM (https://github.com/karina-cherednyk/AddressProgrammingLanguage/blob/master/sources/vm.cpp)

2.1.2. Compiler (https://github.com/karina-cherednyk/AddressProgrammingLanguage/blob/master/sources/compiler.cpp)

2.1.3. Parser (https://github.com/karina-cherednyk/AddressProgrammingLanguage/blob/master/sources/parser.cpp)

2.1.4. Scanner (https://github.com/karina-cherednyk/AddressProgrammingLanguage/blob/master/sources/scanner.cpp)

2.1.5. Chunk (https://github.com/karina-cherednyk/AddressProgrammingLanguage/blob/master/sources/chunk.cpp)

2.1.6. Debugging tools (https://github.com/karina-cherednyk/AddressProgrammingLanguage/blob/master/sources/debug.cpp)

2.2. Examples of Pointers processing programs in the Address programming language original and adapted syntaxes.

C++	Original Address syntax	Adapted Address syntax
1. The simple loop program example, available in repository		

#**include** <iostream>
 	
int main()
{
  for (**int** pi=1; pi<=10;)
  { std::cout << pi << std::endl;
  pi = pi + 3;
  }
  return 0;
}
	Ц{1 (0) ***P***{'***π*** <= 10} => ***π***} l1, l2
	Печать '***π***
	'***π*** + 3 => ***π***
	l1
	l2 ...
		L{1 (0) PR{'pi <= 10} => pi} l1, l2
		print 'pi
		'pi + 3 => pi
		l1
		l2 ...

2. The example of program with nested loops in C++ and corresponding looping formulas in Address Programming Language (ADPL)		

#include <iostream>
 	
int main() {
 	
  for (int pi1=1, pi2 = 2; pi1<=3 && pi2 <= 4; pi1++, pi2 ++){
  std::cout << pi1 + pi2 << std::endl;
  }
 	
  for (int pi1=8, pi2 = 17; pi1<=10 && pi2 <= 24; pi1+= 2, pi2 += 2){
  std::cout << pi1 + pi2 << std::endl;
  }
  return 0;
}
	Ц{1(1)3, 8(2)10 ⇒ ***π***1, 2(1)4, 17(2)24 ⇒ ***π***2} l1, l2
	Печать '***π***1 + '***π***2
	l1
	l2 ...
		L{1(1)3, 8(2)10 => pi1, 2(1)4, 17(2)24 => pi2} l1, l2
		print 'pi1 + 'pi2
		l1
		l2 ...

3. The example of a program in C++ with Pointers dereferencing and the same in ADPL using 2nd rank addressing and double application of the "‘" operation.		

#include <iostream>
 	
int main() {
 	
  int a = 1;
  int *b = &a;
  int c = 3;
 	
  std::cout << a + *b + c; /// 1 + 1 + 3 = 5
  return 0;
}
	'a = 1;'b = a;'c = 3
	Печать 'a + ''b + 'c
		'a = 1;'b = a;'c = 3
		print 'a + ''b + 'c [1 + 1 + 3 = 5]

3. Haskell realization (by Heorhii Sanchenko)

The Address programming language interpreter was realized in Haskell and examples were provided for treelike formats processing (complex hierarchical structures): list chains and binary trees. The realization of a well known Address language function - map is provided, which is widely used in the modern programming tools, including Functional and Logic programming.

We ask to pay your attention to the representation of trees with the help of a "minus - "‘" - operation". The Address programming language allows to represent trees by definition for every node, except the root, parent node. So in order to get the the set of of all sons, we may use the "minus - "‘" - operation".

3.1. URL to the description of Haskell realization: https://docs.google.com/document/d/1qimMgyXB5SFXAb-w2zblOCpCXQbSM2uz/edit?usp=sharing&ouid=115235351447626208409&rtpof=true&sd=true.

3.2. URL to the Haskell realization:: https://github.com/Jorge3129/address_lang_1/tree/main

3.2.1. URL to the structure description with references in README.md (https://github.com/Jorge3129/address_lang_1?tab=readme-ov-file#project-structure):

1) src (https://github.com/Jorge3129/address_lang_1/tree/main/src) - source code for ADPL (Address programming language) interpreter

A. Lib (https://github.com/Jorge3129/address_lang_1/blob/main/src/Lib.hs) - library entry point
B. Lexer (https://github.com/Jorge3129/address_lang_1/tree/main/src/Lexer) - lexical analyzer for ADPL
C. Parser (https://github.com/Jorge3129/address_lang_1/tree/main/src/Parser) - parser for ADPL
D. Value (https://github.com/Jorge3129/address_lang_1/tree/main/src/Value) - data types for representing the values in ADPL
E. ByteCode (https://github.com/Jorge3129/address_lang_1/tree/main/src/ByteCode) - data types for representing ADPL bytecode
F. Compiler (https://github.com/Jorge3129/address_lang_1/tree/main/src/Compiler) - core logic for compiling from source code to ADPL bytecode
G. Vm (https://github.com/Jorge3129/address_lang_1/tree/main/src/Vm) - virtual machine for ADPL bytecode
H. Utils (https://github.com/Jorge3129/address_lang_1/tree/main/src/Utils) - common utils

2) test (https://github.com/Jorge3129/address_lang_1/tree/main/test) - contains examples of programs in the original syntax and in ADPL
3) docs (https://github.com/Jorge3129/address_lang_1/tree/main/docs) - documentation

A. syntax (https://github.com/Jorge3129/address_lang_1/tree/main/docs/syntax.md) - documentation for the ADPL syntax adaptation
B. examples (https://github.com/Jorge3129/address_lang_1/tree/main/docs/examples.md) - comparison between programs written in C++, original Address Programming language syntax, and ADPL

3.2.2. The comparison of programs:

4) In repository: https://github.com/Jorge3129/address_lang_1/blob/main/docs/examples.md.
5) In google docs files: https://docs.google.com/document/d/1t64DKVFAuk5qB66_iir9WIsAPpsUdwveoqv_D0EF4-Y/edit?usp=sharing.

3.2.3. Examples of programs:

<table>
<tr><th>C++</th><th>Original Address syntax</th><th>Adapted Address syntax</th></tr>
<tr><td colspan="3">1. This is the example of a program which calculates and displays to the screen the factorial value of numbers from 0 to 20.
In the original and Adapted language syntax examples we can see the main logic of the program described after the mark M, after the definition of the subprogram fact.
The M mark is used for the visual comparison with the structure of the same program in C++.
URL: https://github.com/Jorge3129/address_lang_1/tree/main/test/data/fact</td></tr>
<tr><td>#include <iostream>

unsigned long fact(unsigned long n) {
  if (n <= 1) return 1;
  auto prevFact = fact(n - 1);
  return n * prevFact;
}

int main() {
  for (size_t i = 0; i <= 20; i++) {
  auto res = fact(i);
  std::cout << res << std::endl;
  }

  return 0;
}</td><td>M

fact ... ∅ → n, ∅ → res
  Р { n <= 1 } 1 ⇒ res; Я ↓
  П fact { n - 1, prevFact }
  n × 'prevFact ⇒ res
  Я

M ...
  Ц { 0 (1) 20 ⇒ i } l1
  П fact { 'i, res }
  Печать 'res
  l1 ...</td><td>M

@fact ... Nil -> n, Nil -> res
  P { n <= 1 } 1 => res; Ret |
  Pg fact { n - 1, prevFact }
  n * 'prevFact => res
  Ret

@M ...
  L { 0 (1) 20 => i } l1
  Pg fact { 'i, res }
  print 'res
  @l1 ...</td></tr>
<tr><td colspan="3">2. The link to the example of a program for operations with a Stack structure.
The program is divided by blocks(subprograms) for the perception convenience.
URL: https://github.com/Jorge3129/address_lang_1/tree/main/test/data/stack</td></tr>
<tr><td colspan="3">2.1.This is an example of a subprogram stack_new for creation of a new instance of a Stack structure.
In a C++ example the memory allocation is done inside the function stack_new.
In the ADPL language case it is assumed that parameter res_addr contains the address of a free memory cell.</td></tr>
<tr><td>struct node {
  int value;
  node* next;
};
node** stack_new() {
  auto s = new node*;
  *s = nullptr;
  return s;
}</td><td>stack_new ... ∅ → res_addr
  0 ⇒ res_addr
  Я</td><td>@stack_new ... Nil -> res_addr
  0 => res_addr
  Ret</td></tr>
<tr><td colspan="3">2.2.This is an example of a stack_push subprogram for adding the value to the top of a stack.
When Original and Adapted syntax are compared we can see that Adapted syntax allows direct assignment of a value to the local variable for more convenience (it allows to avoid extra application of a “ ‘ ” operation).
Besides, when compared with a C++ example, we can see that in the Address programming language the structure fields are imitated by manual assignment of a value by shift (new_node->value is equal to new_addr + 1).</td></tr>
<tr><td>void stack_push(int val, node** head) {
  auto old_addr = *head;

  auto new_node = new node;

  *head = new_node;
  new_node->next = old_addr;
  new_node->value = val;
}</td><td>stack_push ... ∅ → val, ∅ → head
  'head ⇒ old_addr

  alloc 2 ⇒ new_addr

  'new_addr ⇒ head
  'old_addr ⇒ 'new_addr
  val ⇒ 'new_addr + 1
  Я</td><td>@stack_push ... Nil -> val, Nil -> head
  old_addr = 'head

  new_addr = alloc 2

  new_addr => head
  old_addr => new_addr
  val => new_addr + 1
  Ret</td></tr>
</table>

C++	Original Address syntax	Adapted Address syntax
2.3.This is an example of a **stack_pop** subprogram for extraction of values from the top of a stack. Here the predicate formula is used for checking stack on elements presence. For simplification purposes, in case of an empty stack the 0 value is returned.		

int stack_pop(node** head) {
  auto old_addr = *head;
  if (old_addr == nullptr) {
  return 0;
  }
  auto old_val = old_addr->value;
  auto next_addr = old_addr->next;
  *head = next_addr;
  delete old_addr;
  return old_val;
}
	stack_pop ... ***Ø*** → head, ***Ø*** → res_addr
	  'head ⇒ old_addr
	  P { old_addr = 0 } ***Я*** ↓
	
	  '('old_addr + 1) ⇒ old_val
	
	  ''old_addr ⇒ head
	  0 ⇒ 'old_addr
	  'old_val ⇒ res_addr
	  Я
		@stack_pop ... Nil -> head, Nil -> res_addr
		  old_addr = 'head
		  P { old_addr == 0 } Ret |
		
		  old_val = '(old_addr + 1)
		
		  next_addr = 'old_addr
		  next_addr => head
		  0 => old_addr
		  old_val => res_addr
		  Ret

2.4. This is the **stack_is_empty** subprogram example used for checking stack on present elements (emptiness)		

bool stack_is_empty(node** head) {
  return *head == nullptr;
}
	stack_is_empty ... ***Ø*** → head, ***Ø*** → res_addr
	  'head = 0 ⇒ res_addr
	  Я
		@stack_is_empty ... Nil -> head, Nil -> res_addr
		  'head == 0 => res_addr
		  Ret

2.5. This is an example of a program for demonstration of mentioned before subprograms for Stack operations. In the program we can see the looping formula used for gradually adding values from 1 to 5 to the Stack, and for gradual extraction of the stack elements and displaying results to the screen.		

#include <iostream>
 	
// this is a builtin function in ADPL
void printList(node** head) {
  std::cout << '[';
  for (auto cur = *head; cur != nullptr; cur = cur->next) {
  std::cout << cur->value;
  if (cur->next != nullptr) std::cout << ',';
  }
  std::cout << ']' << std::endl;
}
 	
int main() {
  auto s = stack_new();
  printList(s);
 	
  for (int i = 1; i <= 5; i++) {
  stack_push(i, s);
  }
  printList(s);
 	
  while (!stack_is_empty(s)) {
  auto top_val = stack_pop(s);
  std::cout << top_val << std::endl;
  printList(s);
  }
 	
  delete s;
  return 0;
}
	M
	
	M ...
	  П stack_new { s }
	  Печать s
	
	  Ц { 1(1)5 ⇒ i } l1
	  П stack_push { 'i, s }
	  l1 ...
	  Печать s
	
	  П stack_is_empty { s, s_em }
	  Ц { 0(1) P { 's_em ≠ 1 } ⇒ pi } l2
	  П stack_pop { s, top_val }
	  Печать 'top_val
	  Печать s
	  П stack_is_empty { s, s_em }
	  l2 ...
		M
		
		@M ...
		  Pg stack_new { s }
		  printList s
		
		  L { 1(1)5 => i } l1
		  Pg stack_push { 'i, s }
		  @l1 ...
		  printList s
		
		  Pg stack_is_empty { s, s_em }
		  L { 0(1) P { 's_em /= 1 } => pi } l2
		  Pg stack_pop { s, top_val }
		  print 'top_val
		  printList s
		  Pg stack_is_empty { s, s_em }
		  @l2 ...

C++	Original Address syntax	Adapted Address syntax
3. This is a program for operations with a list chain (a data structure similar to a linked lists) URL: https://github.com/Jorge3129/address_lang_1/tree/main/test/data/list_map		
3.1. This is an example of a subprogram **list_empty** for creation of a new instance of data structure list chain. The subprogram definition is similar to the definition of **stack_new** (2.1)		

struct node {
  int value;
  node* next;
};
 	
node** list_empty() {
  auto s = new node*;
  *s = nullptr;
  return s;
}
	list_empty ... ∅ → res_addr
	  0 ⇒ res_addr
	  ∎
		@list_empty ... Nil -> res_addr
		  0 => res_addr
		  Ret

3.2. This is an example of a subprogram **list_add** for adding elements into the end of a list chain. Here the predicate formula is used for emptiness check on the list, and the looping formula for finding the last node.		

void list_add(int val, node** head) {
  auto last_node = *head;
 	
  if (*head != nullptr) {
  for (auto pi = *head; pi != nullptr; pi = pi->next) {
  last_node = pi;
  }
  }
  auto new_node = new node;
  if (*head == nullptr) {
  *head = new_node;
  } else {
  last_node->next = new_node;
  }
  new_node->next = nullptr;
  new_node->value = val;
}
	list_add ... ∅ → val, ∅ → head
	  last_node = head
	
	  P { 'head ≠ 0 } ↓ go
	  Ц { 'head, '∅, P { 'π ≠ 0 } ⇒ π } l1
	  last_node = 'π;
	  l1 ...
	  go ...
	  alloc 2 ⇒ new_node;
	  'new_node ⇒ last_node
	  0 ⇒ 'new_node
	  val ⇒ 'new_node + 1
	  ∎
		@list_add ... Nil -> val, Nil -> head
		  last_node = head
		
		  P { 'head /= 0 } | go
		  L { 'head, 'Nil, P { 'pi /= 0 } => pi } l1
		  last_node = 'pi;
		  @l1 ...
		  @go ...
		  new_node = alloc 2;
		  new_node => last_node
		  ptr(0) => new_node
		  val => new_node + 1
		  Ret

3.3. This is an example of a **list_map** subprogram for application of a certain transformative function to every element of a list chain. We use a looping formula for iteration over a list chain. A special notation [f] is added in the Adapted syntax of Address programming language in order to underline that f - is not a mark of a necessary subprogram, but it is a variable containing the address of this subprogram.		

node** list_map(int (*f)(int val), node** head) {
  auto r = list_empty();
 	
  if (*head == nullptr) return r;
 	
  for (auto pi = *head; pi != nullptr; pi = pi->next) {
  auto val = pi->value;
  auto new_val = (*f)(val);
  list_add(new_val, r);
  }
 	
  return r;
}
	list_map ... ∅ → f, ∅ → head, ∅ → r
	  П list_empty { r }
	
	  P { 'head = 0 } ∎ ↓
	
	  Ц { 'head, '∅, P { 'π ≠ 0 } ⇒ π } l1
	  val = '('π + 1)
	  П f { val, new_val }
	  П list_add { 'new_val, r }
	  l1 ...
	  ∎
		@list_map ... Nil -> f, Nil -> head, Nil -> r
		  Pg list_empty { r }
		
		  P { 'head == 0 } Ret |
		
		  L { 'head, 'Nil, P { 'pi /= 0 } => pi } l1
		  val = '('pi + 1)
		  Pg [f] { val, new_val }
		  Pg list_add { 'new_val, r }
		  @l1 ...
		  Ret

4. *Rust language realization (by Kyrylo Chornokozynskyi)*

An Interpreter and a Compiler of the Address programming language were realized in Rust, examples of processing of treelike formats are provided. The examples of "object-oriented" programs deserve special attention.

4.1. URL link to the description of the Rust language realization: https://drive.google.com/file/d/1wasJuVxo88ROQl_pSCrMStn5UCQeSbsk/view?usp=sharing.

4.2. URL of a Rust realization: https://github.com/kchernokozinsky/address-lang

- 4.2.1. Lexer
- 4.2.2. Parser
- 4.2.3. Compiler
- 4.2.4. Virtual Machine
- 4.2.5. Syntax
- 4.2.6. Examples
- 4.2.7. Syntax Highlighter

4.3. Examples of programs for comparison in C++, ADPL- Address programming language original and Adapted syntax

C++	Original ADPL syntax	Adapted ADPL syntax
1.An example of a program with double application of a " ‘ " operation (Pointer dereferencing) URL: https://github.com/kchernokozinsky/address-lang/tree/main/examples/basic_operations/deref		

#include <iostream>
 	
int main() {
int t = 10;
int* a = &t;
int** b = &a;
 	
std::cout << **b << std::endl;
 	
return 0;
}
	10 ⇒ a; a ⇒ b
	Печать "b
	
	or
	
	10 ⇒ a; a ⇒ b
	Печать ^{2}b
		10 => a; a => b
		Print{"b}
		
		or
		
		10 => a; a => b
		Print { D{b,2} }

C++	Original ADPL syntax	Adapted ADPL syntax
2. URL: https://github.com/kchernokozinsky/address-lang/tree/main/examples/tree		
2.1. This program demonstrates a creation of binary tree nodes and inserting new nodes into the tree. It shows a recursive approach to the inserting operation, where a new node is inserted depending on its value comparison with the current tree node. URL: ttps://github.com/kchernokozinsky/address-lang/tree/main/examples/tree/binary_tree/insert		

struct Node {
  int value;
  Node* left;
  Node* right;
 	
  Node(int val) : value(val), left(nullptr), right(nullptr) {}
};
 	
void insert(Node*& root, Node* node, Node*& result) {
  if (!root) {
  root = node;
  result = node;
  } else if (node->value < root->value) {
  if (!root->left) {
  root->left = node;
  result = node;
  } else {
  insert(root->left, node, result);
  }
  } else {
  if (!root->right) {
  root->right = node;
  result = node;
  } else {
  insert(root->right, node, result);
  }
  }
}
	insert ... ∅ ⇒ root; ∅ ⇒ node
	  P { 'root > 'node } ↓ gte
	  P { '(root + 1) = ∅ } node ⇒ root + 1 ↓ ***Π*** insert {'(root + 1), node}
	  ϑ
	  gte ...
	  P { '(root + 2) = ∅ } node ⇒ root + 2 ↓ ***Π*** insert {'(root + 2), node}
	  ϑ
	ϑ
	
	node... ∅ ⇒ val
	  ∅ ⇒ val + 1
	  ∅ ⇒ val + 2
	ϑ
		insert ... null => root; null => node
		  P { 'root > 'node } | @gte
		  P { '(root + 1) == null } node => root + 1 | SP insert {'(root + 1), node}
		  return
		  gte ...
		  P { '(root + 2) == null } node => root + 2 | SP insert {'(root + 2), node}
		  return
		return
		
		node... null => val
		  null => val + 1
		  null => val + 2
		return

2.2. This program finds a node with a minimal value in a binary search tree starting from the pointed node. URL: https://github.com/kchernokozinsky/address-lang/tree/main/examples/tree/binary_tree/min		

void min(Node* node, Node*& leaf) {
  if (!node) {
  leaf = nullptr;
  return;
  }
 	
  Node* current = node;
  while (current && current->left) {
  current = current->left;
  }
  leaf = current;
}
	min ... ∅ ⇒ node; ∅ ⇒ leaf;
	  Π is_leaf {node, is_leaf}
	  P { 'is_leaf} ↓ not_leaf
	  node ⇒ leaf
	  ϑ
	  not_leaf...
	  Π has_one_son {node, son}
	  P { 'son = ∅ } two_sons↓
	  Π min {'son, leaf}
	  ϑ
		min ... null => node; null => leaf;
		  SP is_leaf {node, is_leaf}
		  P { 'is_leaf} | @not_leaf
		  node => leaf
		  return
		  not_leaf...
		  SP has_one_son {node, son}
		  P { 'son == null } @two_sons|
		  SP min {'son, leaf}
		  return

C++	Original ADPL syntax	Adapted ADPL syntax
2.3.This program demonstrates a node deletion function on the binary search tree. It searches recursively node with a value **val** and deletes it. URL:https://github.com/kchernokozinsky/address-lang/tree/main/examples/tree/binary_tree/remove		

void rem(Node* father, Node*& node, int val,
Node*& result) {
  if (!node) {
  std::cout << "Element not found" << std::endl;
  result = nullptr;
  return;
  }
 	
  if (node->value == val) {
  bool isLeaf;
  is_leaf(node, isLeaf);
 	
  if (isLeaf) {
  if (father) {
  if (father->left == node) {
  father->left = nullptr;
  } else {
  father->right = nullptr;
  }
  } else {
  node = nullptr;
  }
  delete node;
  result = nullptr;
  return;
  }
 	
  Node* one_son;
  has_one_son(node, one_son);
 	
  if (one_son) {
  if (father) {
  if (father->left == node) {
  father->left = one_son;
  } else {
  father->right = one_son;
  }
  } else {
  node = one_son;
  }
  delete node;
  result = one_son;
  return;
  }
 	
  Node* leaf;
  min(node->right, leaf);
 	
  node->value = leaf->value;
  rem(node, node->right, leaf->value, result);
  return;
  }
 	
  if (val < node->value) {
  if (!node->left) {
  std::cout << "Element not found" <<
std::endl;
  result = nullptr;
  return;
  }
  rem(node, node->left, val, result);
  } else {
  if (!node->right) {
  std::cout << "Element not found" <<
std::endl;
  result = nullptr;
	rem ... ∅ ⇒ father; ∅ ⇒ node; ∅ => val;
	  P { 'node = 'val } ↓ to_find
	  Π is_leaf {node, is_leaf}
	  P { 'is_leaf} ↓ case2
	  P { node = '(father + 1)} ∅ ⇒ father + 1 ↓ ∅ ⇒ father + 2
	  ϑ
	  case2 ...
	  Π has_one_son {node, son}
	  P { 'son = ∅ } case3 ↓
	  P { node = '(father + 1)} 'son ⇒ father + 1 ↓ 'son ⇒ father + 2
	  ϑ
	  case3 ...
	
	  Π min {node, leaf}
	
	  Π rem {father, node, 'leaf}
	
	  P { node = '(father + 1)} 'leaf ⇒ father + 1 ↓ 'leaf => father + 2
	  '(node + 1) ⇒ 'leaf + 1; '(node + 2) ⇒ 'leaf + 2
	  ϑ
		rem ... null => father; null => node; null => val;
		  P { 'node == 'val } | @to_find
		  SP is_leaf {node, is_leaf}
		  P { 'is_leaf} | @case2
		  P { node == '(father + 1)} null => father + 1 | null => father + 2
		  return
		  case2 ...
		  SP has_one_son {node, son}
		  P { 'son == null } @case3 |
		  P { node == '(father + 1)} 'son => father + 1 | 'son => father + 2
		  return
		  case3 ...
		
		  SP min {node, leaf}
		
		  SP rem {father, node, 'leaf}
		
		  P { node == '(father + 1)} 'leaf => father + 1 | 'leaf => father + 2
		  '(node + 1) => 'leaf + 1; '(node + 2) => 'leaf + 2
		  return

C++	Original ADPL syntax	Adapted ADPL syntax

return;
}
rem(node, node->right, val, result);
}
}
void is_leaf(Node* node, bool& result) {
result = !node->left && !node->right;
}
 	
void has_one_son(Node* node, Node*& one_son) {
if (!node->left && !node->right) {
one_son = nullptr;
} else if (!node->left) {
one_son = node->right;
} else if (!node->right) {
one_son = node->left;
} else {
one_son = nullptr;
}
}
	is_leaf ... ∅ ⇒ node; ∅ ⇒ is_leaf
	'(node + 1) = ∅ and '(node + 2) = ∅ ⇒ is_leaf
	∎
	
	has_one_son ... ∅ => node; ∅ => one_son
	P { '(node + 1) ≠ ∅ and '(node + 2) ≠ ∅} ∎ ↓
	P { '(node + 1) = ∅ and '(node + 2) = ∅} ∎ ↓
	P { '(node + 1) = ∅ } '(node + 2) ⇒ one_son ↓
	'(node + 1) ⇒ one_son
	∎
		is_leaf ... null => node; null => is_leaf
		'(node + 1) == null and '(node + 2) == null => is_leaf
		return
		
		has_one_son ... null => node; null => one_son
		P { '(node + 1) != null and '(node + 2) != null} return |
		P { '(node + 1) == null and '(node + 2) == null} return |
		P { '(node + 1) == null } '(node + 2) => one_son | '(node + 1) => one_son
		return

C++	Original ADPL syntax	Adapted ADPL syntax
3. This program demonstrates the structure Person with fields and functions for operations with these objects. URL: https://github.com/kchernokozinsky/address-lang/tree/main/examples/object		

#include <iostream>
#include <string>
struct Person {
  std::string* name;
  int* age;
  Person* married_on;
 	
  Person(const std::string& name_val, int age_val)
  : name(new std::string(name_val)), age(new int(age_val)), married_on(nullptr) {}
  ~Person() {
  delete name;
  delete age;
  }
};
void say_hi(Person** person) {
  if (!*person || !(*person)->name) {
  std::cout << "Hello, I have no name :(" << std::endl;
  } else {
  std::cout << "Hello, my name is " << *(*person)->name << std::endl;
  }
}
void say_age(Person** person) {
  if (!*person || !(*person)->age) {
  std::cout << "I don't know how old I am :(" << std::endl;
  } else {
  std::cout << "I'm " << *(*person)->age << " years old" << std::endl;
  }
}
void say_who_you_married_on(Person** person) {
  if (!*person || !(*person)->married_on || !(*person)->married_on->name) {
  std::cout << "I'm not married yet" << std::endl;
  } else {
  std::cout << "I'm married to " << *(*person)->married_on->name << std::endl;
  }
}
void marry(Person** person1, Person** person2) {
  if (person1 && *person1 && person2 && *person2) {
  (*person1)->married_on = *person2;
  (*person2)->married_on = *person1;
  }
}
int main() {
  Person* alice = new Person("Alice", 23);
  Person* bob = new Person("Bob", 25);
 	
  say_hi(&alice);
  say_age(&alice);
 	
  marry(&alice, &bob);
  say_who_you_married_on(&alice);
  delete alice;
  delete bob;
 	
  return 0;
}
	M
	
	new ... **Ø** ⇒ name; **Ø** ⇒ age; **Ø** ⇒ married_on; **Ø** ⇒ person;
	  'name ⇒ p
	  'age ⇒ p + 1
	  'married_on ⇒ p + 2;
	  p ⇒ person
	ᗱ
	
	destruct ... **Ø** ⇒ person;
	  Ø ⇒ person
	  Ø ⇒ person + 1
	  Ø ⇒ person + 2;
	ᗱ
	
	say_hi ... **Ø** ⇒ person;
	  P {'person = **Ø**} **Печать** "Hello, I have no name :(" ↓ **Печать** "Hello, my name is " + 'person
	ᗱ
	
	say_age ... **Ø** ⇒ person;
	  P {'(person + 1) = **Ø**} **Печать** "I don`t how old I am :(" ↓ **Печать** "I`m " + '(person + 1) + " years old"
	ᗱ
	
	say_who_you_married_on ... **Ø** ⇒ person;
	  P { '('person + 2) = **Ø**} **Печать** "I`m not married yet" ↓ **Печать** "I`m married on " + ''('person + 2)
	ᗱ
	
	marry ... **Ø** ⇒ person1; **Ø** ⇒ person2;
	  'person2 ⇒ 'person1 + 2
	  'person1 ⇒ 'person2 + 2
	ᗱ
	
	M...
	"Alice" ⇒ name1
	23 ⇒ age1
	Ø ⇒ married_on1
	
	"Bob" ⇒ name2
	25 ⇒ age2
	Ø ⇒ married_on2
	
	П new {name1,age1, married_on1, alice}
	П new {name2,age2, married_on2, bob}
	
	П say_hi {'alice}
	П say_age {'alice}
		@M
		
		new ... null => name; null => age; null => married_on; null => person;
		  'name => p
		  'age => p + 1
		  'married_on => p + 2;
		  p => person
		return
		
		destruct ... null => person;
		  null => person
		  null => person + 1
		  null => person + 2;
		return
		
		say_hi ... null => person;
		  P {'person == null} Print {"Hello, I have no name :("} | Print {"Hello, my name is ", 'person}
		return
		
		say_age ... null => person;
		  P {'(person + 1) == null} Print {"I don`t how old I am :("} | Print {"I`m ", '(person + 1), " years old"}
		return
		
		say_who_you_married_on ... null => person;
		  p = 'person + 2
		  P { 'p == null} Print {"I`m not married yet"} | Print {"I`m married on ", 'p}
		return
		
		marry ... null => person1; null => person2;
		  'person2 => 'person1 + 2
		  'person1 => 'person2 + 2
		return
		
		M...
		"Alice" => name1
		23 => age1
		null => married_on1
		
		"Bob" => name2
		25 => age2
		null => married_on2
		
		SP new {name1,age1, married_on1, alice}
		SP new {name2,age2, married_on2, bob}
		
		SP say_hi {'alice}
		SP say_age {'alice}

C++	Original ADPL syntax	Adapted ADPL syntax
	П marry {alice, bob} **П** say_who_you_married_on {alice} **П** destruct {alice} **П** destruct {bob}	SP marry {alice, bob} SP say_who_you_married_on {alice} SP destruct {alice} SP destruct {bob}

B. The Address programming language was realized for popularization purposes and to give an opportunity to interested enthusiasts to write,run and debug their own programs in this language.

We will be grateful for your analysis and your valuable feedback regarding provided examples and programs in the repositories.

Thank you for your support in discovering this important historical achievement and saving its legacy.

C. A list of monographs with published examples of programs in the Address programming language that use 2nd-rank addressing (Pointers) and list chains

1. Гнєденко Б.В., Элементы программирования / Гнєденко Б.В., Королюк В.С., Ющенко Е.Л.; // М. : – ГИФМЛ, 1961. – 348 с., https://files.infoua.net/yushchenko/Elementy-programmirovaniya_BGnedenko-VKoroljuk-EYushchenko_1961.pdf .

2. Глушков В.М., Вычислительная машина "Киев". Математическое описание / В.М. Глушков, Е.Л. Ющенко. // К. : – Гостехиздат УССР, 1962. – 183 с. : ил., https://files.infoua.net/yushchenko/Vychislitelnaya-mashyna-Kiev_VHlushkov_EYushchenko_1962.pdf .

3. Ющенко Е.Л., Адресное программирование // К. : – Гос. издательство технической литературы, УРСР, 1963. – 287 с., https://files.infoua.net/yushchenko/Adresnoe-programmirovanie_EYushchenko_1963.pdf .

With best regards,

Docent of National University Kyiv Mohyla Academy (NAUKMA)

_____ Yurii Yushchenko

Master of NAUKMA graduate _____ Karyna Cherednyk

Bachelor of Engineering _____ Heorhii Sanchenko

NAUKMA graduate _____ Kyrylo Chornokozynskyi

Master of Computer Engineering _____ Oleksandr Bezrukavyi