UDC 004.432
DOI https://doi.org/10.32838/2663-5941/2020.1-1/24

Sulema Ye.S.
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"

Peschanskii V.Yu.
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"

TIMEWISE DATA PROCESSING WITH PROGRAMMING LANGUAGE ASAMPL

The domain-specific programming language ASAMPL is presented and its translation is discussed in the paper. This programming language enables data processing by using specific operators which enable data timewise processing, multimodal data synchronization, and aggregation. It gives flexibility in working with such data sources as remote sensors and cloud storages. The program code in programming language ASAMPL includes nine sections. The Libraries section allows a programmer to declare a list of imported libraries to be used in the program code. The Handlers section and the Renders section enable selection of handling and rendering tools from predefined libraries for their further use. The Sources section consists of a list of declarations for access to external resources. The Sets section enables declaration of data types. The Elements section is used for definition of single-value data. The Tuples section is used for definition of data tuples which are timewise ordered data values; a tuple is a specific data type of ASAMPL. The Aggregates section is used for definition of complex data structures called aggregates; this data type is a specific data type of ASAMPL. The Actions section includes any necessary operators which implement the logic of data processing in a program. The paper also presents the translator of a program code in ASAMPL and explains its components. The performance of the ASAMPL language compiler was evaluated based on two characteristics: run time and size of executable code. The comparison has shown that a program compiled by an ASAMPL translator is twice as short as a program in programming language C ++, which implements the same logic of data processing. The proposed programming language ASAMPL is aimed at the development of applied software dealing with multimodal data defined with respect to time. It can be used in a wide range of applications where timewise data processing is required.

Key words: *timewise data processing, programming language, program code translation.*

Problem statement. The task of timewise data processing is topical for many engineering applications, including medical engineering. It concerns the case when data is obtained from certain external source producing a sequence of values accompanied with time stamps. In this case, there is a need in timewise data processing, including data synchronization. In spite of that this task can be solved by using general-purpose languages, employing a domain-specific language can be a better option in some application cases.

Review of the literature. There is a wide range of programming languages developed and, thus, the number of research papers and books presenting different approaches of programming is also significant. Let us focus on some interesting researches related to the research presented in this paper. In [1], the author presents two views on real-time programming: based on use of general purpose languages and base on use of special purpose synchronous languages. The advantages and disadvantages of both approaches are highlighted and discussed. In [2], the authors survey the literature available on the topic of domain-specific languages and discuss terminology, risks and benefits, example domain-specific languages, design methodologies, and implementation techniques. The recent researches show that the interest to domain-specific languages is rising. Thus, there is a number of papers [3–9] which presents programming languages developed for a certain specific purpose. In particular, research [8] presents an approach of programmable programming languages. The general view on programming languages design is given in [10].

Task statement. The purpose of the research is the development of domain-specific language ASAMPL, which enables timewise data processing, as well as the development of a translator for this domain-specific language.

Presentation of the Main Research Material. The program language ASAMPL [11] is designed in order to create software tools for multimodal data processing which has to be carried out with respect to time scale represented by time stamps. These time stamps correspond to time moments when data of certain modality is obtained. The data can be received as a result of measuring a specific parameter characterizing an object of study. Let us present the language concept.

A program code in programming language ASAMPL consists of nine basic sections, the general scheme of which is as follows:

'Program', name, '{',
libraries section,
handlers section,
renderers section,
sources section,
sets section,
elements section,
tuples section,
aggregates section,
actions section,
'}';

The Libraries section allows a programmer to declare a list of imported libraries to be used in the program code. The Handlers section and the Renders section enable selection of handling and rendering tools from predefined libraries for their further use. The Sources section consists of a list of declarations for access to external resources. The Sets section enables declaration of data types. The Elements section is used for definition of single-value data. The Tuples section is used for definition of data tuples which are timewise ordered data values. A tuple is a specific data type of ASAMPL. Its nearest analogue in high-level programming language is an array or a structure. The Aggregates section is used for definition of complex data structures called aggregates. This data type is specific data type of ASAMPL and in fact it is a tuple of tuples. Finally, the Actions section includes any necessary operators which implement the logic of data processing in the program.

Let us define the program structure and operators in programming language ASAMPL by using Extended Backus-Naur Form. Then the program in ASAMPL can be defined by the syntactic rule (1).

program = PROGRAM , *identifier* , "{" ,
LIBRARIES , "{" , { *identifier* , (IS | "=") , *link* , ";" } ["//" , *comment*] "}" , ,
HANDLERS , "{" , { *identifier* , (IS | "=") , *link* , ";" } ["//" , *comment*] "}" , ,
RENDERERS , "{" , { *identifier* , (IS | "=") , *link* , ";" } ["//" , *comment*] "}" ,
SOURCES , "{" , { *identifier* , (IS | "=") , *link* , ";" } ["//" , *comment*] "}" ,
SETS , "{" , { *identifier* , (IS | "=") , *type*, ";" } ["//" , *comment*] "}" , (1)
ELEMENTS , "{" , { *identifier* , (IS | "=") , (*set_type* | *value*) , ";" } ["//" , *comment*] "}" ,
TUPLES , "{" , { *identifier* , (IS | "=") , *set_type* , ";" } ["//" , *comment*] "}" ,
AGGREGATES "{" , { *identifier* , (IS | "=") , "[" , (*identifier* | *values_tuple* , ";" } ["//" , *comment*] "}" ,
ACTIONS , "{" , { *operator* , ";" } ["//" , *comment*] "}" , ["//" , *comment*] , "}" ;

The following operators are defined in programming language ASAMPL: TIMELINE, SEQUENCE, IF THEN, CASE OF, SUBSTITUTE FOR WHEN, DOWNLOAD FROM, UPLOAD TO, IS, RENDER WITH.

The timewise processing operator TIMELINE (TIMELINE AS) is a specific operator which allows a programmer to apply a certain action during a defined time period. Actions included into the operator body are carried out simultaneously. This operator can be considered as a specific type of a loop. There are three types of this operator:

− TIMELINE *time_value_1* : *step* : *time_value_2* {*a list of simultaneous actions*}

− TIMELINE AS *time_values_tuple* {*a list of simultaneous actions*}

− TIMELINE UNTIL *condition* {*a list of simultaneous actions*}.

This operator is defined by the syntactic rule (2).

timewise_processing_operator = TIMELINE , (*identifier*
| *time_value*) , ":", (*identifier* | *time_value*) , ":" ,
(*identifier* | *time_value*) , "{" , { *action* } , "}"
| TIMELINE , AS , *time_values_tuple* , "{" , { *action* } , "}" (2)
| TIMELINE , UNTIL , *logical_expression* , "{" ,
{ *action* } , "}" ;

The operator of sequential processing SEQUENCE {*a list of sequential actions*} is a specific operator which allows to unite data processing actions, which have to be carried our sequentially, in one compound action. This operator is defined by the syntactic rule (3).

sequential_processing_operator = SEQUENCE ,
"{" , { *action* , [";"]} , "}" ; (3)

The branch statement IF THEN (IF THEN ELSE) is a standard operator defined in many other languages. It is defined by the syntactic rule (4).

branch_statement = IF , *logical_expression* ,
THEN , "{" , { *action* } , "}" | IF , *logical_expression* , THEN , "{" , { *action* } , "}" , ELSE , "{" , { (4)
action } , "}" ;

The selection statement CASE OF (CASE OF ELSE) is also a standard operator available in many high-level programming languages. This operator is defined by the syntactic rule (5).

selection_statement = CASE , *identifier* , OF , "{"
, { (*identifier* | *value*) , ":" , *action* } , "}" | CASE
, *identifier* , OF , "{" , { (*identifier* | *value*) , ":" , (5)
action } , "}" , ELSE , "{" , (*identifier* | *value*) , ":" ,
action , "}" ;

The replacement operator SUBSTITUTE FOR WHEN is a specific operator which enables replacement of one data set by another one if a certain condition is true. For example, it can be used for replacement of high-resolution data by low-resolution data if the communication channel is limited. It is defined by the syntactic rule (6).

replacement_operator = SUBSTITUTE , *identifier* , FOR , *identifier* , WHEN *logical_expression* ; (6)

The downloading operator DOWNLOAD FROM (DOWNLOAD FROM WITH) is a specific operator which allows to download data from a certain data source such as remote device, cloud storage, local storage, etc. and to assign this data to a variable that can be of any type: an element, a tuple, and an aggregate. Data transformation form a specific data format is carried out with the use of a predefined handler. This operator is defined by the syntactic rule (7).

downloading_operator = DOWNLOAD , *identifier* , FROM , *identifier* [WITH , *identifier*] ; (7)

The uploading operator UPLOAD TO (UPLOAD TO WITH) is a specific operator which allows to upload data which is assigned to a certain variable of any type (an element, a tuple, an aggregate) to a defined resource, which can be either remote or local, as a file. Transformation of data from the variable type to a destination format is fulfilled by using a predefined handler. It is defined by the syntactic rule (8).

uploading_operator = UPLOAD , *identifier* , TO , *identifier* [WITH , *identifier*] ; (8)

The assignment operator IS is a standard operator available in other high-level programming languages. It is defined by the syntactic rule (9).

assignment_operator = *identifier* , (IS | "=") , *value* ; (9)

The rendering operator RENDER WITH is a specific operator which enables data reproduction by using a specific tool for this purpose. This operator is defined by the syntactic rule (10).

rendering_operator = RENDER , *identifier* , WITH , *identifier* ; (10)

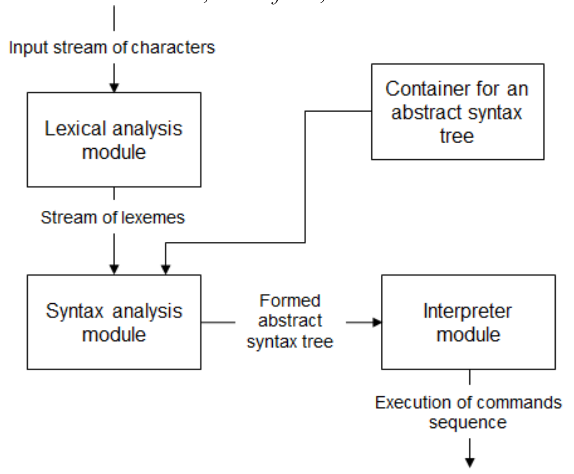
Input stream of characters
Lexical analysis module
Container for an abstract syntax tree
Stream of lexemes
Syntax analysis module
Formed abstract syntax tree
Interpreter module
Execution of commands sequence

Fig. 1. Generalized scheme of translator

In the ASAMPL translator, there are three main models, which are essential for processing and running of programming code: lexer, parser, and interpreter.

Lexer is a module of input characters sequence processing that performs lexical analysis of input sequence of symbols. The result of lexer execution is a sequence of lexeme, or tokens.

Lexical analysis executes in terms of formal sets of rules. In our case, the rules are determined by ASAMPL language grammar. It assigns a set of lexemes that can occur in the input sequence of characters. The result of this module is a sequence of lexemes prepared for their further processing in the next modules of translator.

Parser is a module that is aimed at comparison of linear sequence of formal language lexemes, i.e. ASAMPL lexemes, with its formal grammar. During the process of parsing, the input linear sequence of

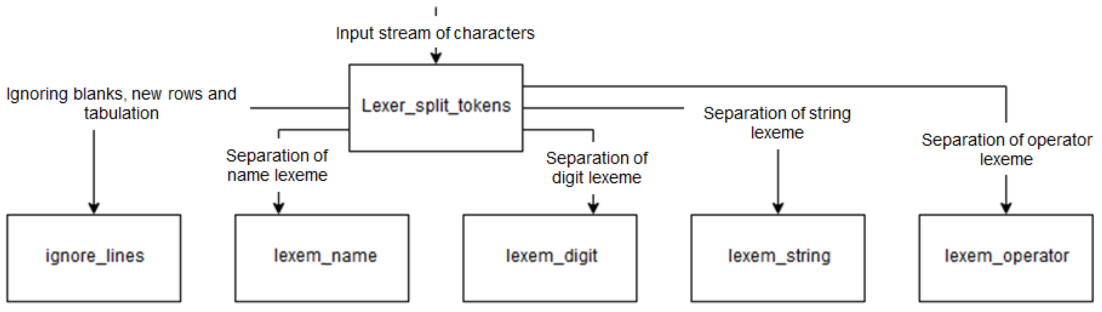
Input stream of characters
Ignoring blanks, new rows and tabulation
Lexer_split_tokens
Separation of string lexeme
Separation of operator lexeme
Separation of name lexeme
Separation of digit lexeme
ignore_lines
lexem_name
lexem_digit
lexem_string
lexem_operator

Fig. 2. Generalized scheme of lexical analysis module

characters is transformed into an abstract syntax tree that clearly illustrates the syntactic structure of the input sequence. It allows to switch to program processing of data in the translator.

Thus, the main task of syntax analyzer is parsing the rules implemented according to the recursive descent parser, i.e. by mutual calling of functions, when each function corresponds to one of grammar rules.

Rules that apply sequentially, i.e. from left to right, absorb lexemes obtained from lexical analyzer. After absorbing and processing lexemes forming of abstract syntax tree, which will be interpreted to an executable code in the next stage, occur.

Let us consider the algorithm of the module in details: Tree* parser_buid_tree(std::vector<Lexem>* lexem_sequence);

The input is a sequence of tokens obtained in the previous step during the processing of input data. This function fills in the fields of the Parser class, which contains links to the input list of tokens, an iterator for examining them one by one, and a string for storing errors that might be formed during conversion to an abstract syntax tree.

Parsing process is recursive, so in addition to creating a Parser class and processing a return value, another function is called in the body of the main module: static Tree * program(Parser * parser).

Let us refer to this function as a root one, as it calls the handlers for each of program blocks, in which the variables will be initialized for later use:

static Tree * libraries_section(Parser * parser);
static Tree * handlers_section(Parser * parser);
static Tree * renderers_section(Parser * parser);
static Tree * sources_section(Parser * parser);
static Tree * sets_section(Parser * parser);
static Tree * elements_section(Parser * parser);
static Tree * tuples_section(Parser * parser);
static Tree * aggregates_section(Parser * parser);
static Tree * actions_section(Parser * parser);

If no errors occurred during the execution of these functions, then the result is an abstract syntax tree that is ready for further processing, otherwise the tree will be incomplete and therefore not executable. In this case, the main function returns a NULL value and outputs an error code containing information about the line in which it occurred.

Let us consider a scheme of functioning of syntactic parsing of a sequence of tokens in details. There are two functions that are not the implementation of one of the rules of formal grammar ASAMPL:

static Tree * accept(Parser * parser, LexemType Lexem);

static Tree * expect(Parser * parser, LexemType Lexem);

The Accept function reads each subsequent item in a token sequence and compares its type with the type that was passed to it. If the types match, the function successfully reads the item.

The Expect function calls the Accept function, after which this function not only checks whether the next token belongs to a specific type, but also requires it for each subsequent token. If the check condition is not met, the function returns an error and outputs information on which line the error occurred.

static bool ebnf_sequence(Parser * parser, Tree * node_to_fill, GrammarRule rule)

The Sequence function reads cyclically the rule passed to it as an argument any number of times. This function is used to add any number of descendant nodes to those tree leaf nodes that are responsible for initializing variables or performing prescribed actions.

static Tree * ebnf_one_of(Parser * parser, Grammar-Rule rules[],
size_t length)

The one_of function tries to read at least one of the rules passed to it as an array. If possible, it returns the result of this rule as another branch of the abstract syntax tree, and the process completes. If an error occurred while executing a rule or none of the list of passed rules could be applied to the next token sequence, NULL is returned.

static Tree * ebnf_one_of_lexem(Parser * parser, Lex-emType types[], size_t length)

The one_of_lexem function attempts to read at least one of the tokens that were passed to it as array elements. If possible, it returns a node with a type of token processed as a new node of an abstract syntax tree, and the process completes. In case none of the list of transmitted tokens could be read, NULL is returned.

static Tree * ebnf_ap_main_rule(Parser * parser, GrammarRule next, GrammarRule ap)

The Ap Main Rule function, which is the result of left recursion, checks whether the following priority rule is applied, and if true, checks if the apostrophe rule is applied.

static Tree * ebnf_ap_recursive_rule(Parser * parser, LexemType types[], size_t typesLen, GrammarRule next, GrammarRule ap)

The Ap Recursive Rule function checks whether transmitted tokens are found, and if they are applied, whether the rule continues recursively.

The above functions are a list of rules that do not originate from the ASAMPL formal grammar rules, so they form all other rules that are directly derived from the formal ASAMPL grammar description.

Each function, which is a direct result of ASAMPL formal grammar description and is non-finite, contains calls to one of the basic functions, as well as calls to at least one of these functions. The finite functions do not contain calls to such functions, so they do not deepen the recursion and return a node leaf, which is the end of parsing for this branch.

The performance of the ASAMPL language compiler was evaluated based on two characteristics: run time (table 1) and size of executable code (table 2) to process the same ASAMPL sequence of actions and methods imported from the libraries.

The results for the ASAMPL compiler were compared with those obtained for the G ++ compiler (GNU C++) when parsing C ++ program equivalent. The comparison was made for two sets of ASAMPL language test code and its C ++ language equivalent.

The comparison was made for two different test versions of a programming code, with and without the use of additional arithmetic functions. The comparison tables above show that a program compiled by an ASAMPL translator is twice as short as the same C ++ program. Accordingly, more complex data processing programs requires much less memory using ASAMPL than programs written in other programming languages.

Table 1

Average running time in milliseconds

	ASAMPL	G++
Test set 1	1106	1063
Test set 2	1521	1427

Table 2

The size of executable code for processing video files in rows

	ASAMPL	G++
Test set 1	4	9
Test set 2	6	14

At the same time, the timing characteristics are quite close to the average program running time. Improvement of time characteristics is possible through optimization of the translator code.

Conclusions. The proposed programming language ASAMPL is aimed at the development of applied software dealing with multimodal data defined with respect to time. It can be used in a wide range of applications where timewise data processing is required. The specific feature of programming language ASAMPL is its orientation on both multimodal data structures processing and work with external devices and data storages. The translation and execution of program code developed in programming language ASAMPL can be fulfilled with the proposed translator.

References:

1. Berry G. Real time programming: special purpose or general purpose languages. INRIA, 1989.
2. Van Deursen A., Klint P., Visser J. Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices, 200. Vol. 35, No. 6. PP. 1–11.
3. Gaunt A.L. et al. TerpreT: A Probabilistic Programming Language for Program Induction. Cornell University, 2016.
4. Bingham E. et al. Pyro: deep universal probabilistic programming. The Journal of Machine Learning Research, 2019. Vol. 20, No. 1.
5. Carpenter B. et al. Stan: A probabilistic programming language. Journal of Statistical Software, 2017. Vol. 76. No. 1.
6. Hong Ge, Kai Xu, Zoubin Ghahramani. Turing: a language for exible probabilistic inference. *AISTATS*, 2018.
7. Coblenz M. Obsidian: A Safer Blockchain Programming Language. *IEEE/ACM 39th International Conference on Software Engineering Companion*, 2017.
8. Felleisen M. et al. A programmable programming language. Communications of the ACM, 2018. Vol. 61, No. 3.
9. Andersen, L., Chang, S., Felleisen, M. Super 8 languages for making movies. *ACM SIGPLAN International Conference on Functional Programming*, 2017. PP. 1–29.
10. Parr T. Language Implementation Patterns. USA, 2010. 389 p.
11. Sulema Ye. ASAMPL: Programming Language for Mulsemedia Data Processing Based on Algebraic System of Aggregates. Advances in Intelligent Systems and Computing, Springer, 2018. Vol. 725. PP. 431–442.

Сулема Є.С., Песчанський В.Ю. ЧАСОВА ОБРОБКА ДАНИХ З ВИКОРИСТАННЯМ МОВИ ПРОГРАМУВАННЯ ASAMPL

У статті представлено спеціальну мову програмування ASAMPL та розглянуто спосіб її компіляції. Ця мова програмування дозволяє оброблювати дані, використовуючи спеціальні оператори, що дають можливість оброблювати часові дані, синхронізувати мультимодальні дані та здійснювати їх агрегацію. Це надає гнучкості у роботі з такими джерелами даних, як дистанційні сенсори та *хмарні сховища. Програмний код мовою ASAMPL включає дев'ять блоків. Блок бібліотек дозволяє програмісту оголосити список імпортованих бібліотек, які будуть використані в програмному коді. Блок оброблювачів та блок рендерингу дозволяють обрати інструменти для обробки та рендерингу з попередньо визначених бібліотек для подальшого їх використання. Блок джерел містить перелік декларацій доступу до зовнішніх ресурсів. Блок множин дозволяє оголошувати типи даних. Блок елементів використовується для визначення даних, що представлені одним значенням. Блок кортежів використовується для визначення кортежів даних, які впорядковані за часом; кортеж є специфічним типом даних ASAMPL. Блок агрегатів застосовується для визначення складних структур даних, які називаються агрегатами; цей тип даних є специфічним типом даних ASAMPL. У блок дій входять всі необхідні оператори, які реалізують логіку обробки даних у цій програмі. У статті також пропонується структура транслятора програмного коду, написаного мовою ASAMPL, а також пояснюються її складові частини. Ефективність компілятора мови ASAMPL оцінювалась на основі двох характеристик: час виконання та розмір виконуваного коду. Порівняння показало, що програма, скомпільована за допомогою транслятора ASAMPL, вдвічі коротша за програму, яка написана мовою програмування C++, та реалізує ту саму логіку обробки даних. Запропонована мова програмування ASAMPL призначена для розроблення прикладного програмного забезпечення для обробки мультимодальних даних, визначених у часі. Її можна використовувати в широкому колі задач, де потрібна обробка даних у часі.*

Ключові слова: *часова обробка даних, мова програмування, трансляція програмного коду.*