UDC 004.432 https://doi.org/10.15407/pp2020.01.074

Yevgeniya Sulema, Vladyslav Glinskii

SEMANTICS AND PRAGMATICS OF PROGRAMMING LANGUAGE ASAMPL

This paper presents semantics and practical implementation of the domain-specific programming language ASAMPL. This programming language has been developed to support the efficient processing of multimodal data processing, in particular, the processing of multimedia content which components are evidently defined in terms of time. The data processing concept employed in ASAMPL is based on the data structures, operations, and relations defined in the algebraic system of aggregates. The paper explains the compilation approach used for this programming language as well as it presents the test results and their discussion.
Key words: multimedia, multimodal data, programming language, compilation.

Introduction

Nowadays, such technologies as IoT, machine learning, big data, augmented and virtual reality are developing very fast and they are more and more in use for solving everyday tasks. For example, virtual assistants like Amazon Alexa or Apple Siri help users by employing natural language processing, voice recognition and computer vision along with other artificial intelligence techniques.

Multimedia technologies along with AR and VR allow us to make education more efficient [1–3]. Some examples are interactive classes and training programs for medical students, technicians, drivers, etc. These technologies enable modelling processes and visualize them, recreating required environment, improve skills while minimizing loss due to learner's mistake. They can be used to make monitoring and operating machines or even industrial complexes much simpler. As an example, it can greatly improve operation of the drone used in emergency situation by giving to operator visualization of the environment, identified objects or threats.

IoT solutions are used not only in industrial and agricultural complexes but also to create "smart homes" and "smart cities". Data gathered from different sensors can be used to make immediate decisions to control operation of connected equipment. At the same time this data is stored for analysis and prediction of the future trends in self-learning systems, visualization to human operators.

Rising amounts of diverse data push requirements to its storage, transmission and processing technologies further. So, nowadays we have real need to work with so-called multimodal data, i.e. data which has different origin, such as images, audio, environmental characteristics (humidity, temperature), etc. Usually such data is constantly changing in time. In some cases, the use of general-purpose programming languages for multimodal data processing tasks is not efficient enough when timewise processing is required. The understanding of this fact has leaded to the creation of a domain-specific programming language ASAMPL [4]. The purpose of the research presented in this paper is to make the semantics of ASAMPL more mature as well as to carry out the reference implementation of this language by developing a compiler and a standard library.

1. Concept and features of ASAMPL

The ASAMPL programming language derives its name from the "Algebraic System of Aggregates" and the "Mulsemedia data Processing Language" [4]. ASAMPL has been designed for simple and efficient multimodal data processing which is based on the following facts: (1) every element of such data is related to a certain point in time when it has been obtained (recorded, generated, measured, etc.); (2) multimodal data processing needs to take into account the modality of the data, its compatibility and the possible interrelation between data of different modalities. Therefore, the basic principle of organizing a program in this language is that the program code execution is governed by two entities: time and data modality.

© Y. Sulema, V. Glinskii, 2020

A key concept in ASAMPL is the concept of multi-image which is based on the mathematical apparatus of the Algebraic System of Aggregates (ASA) [5-7]. To represent the multi-image of a real-world object, two basic data structures defined in ASA, namely, tuples and aggregates, are employed in this programming language. The processing of tuples and aggregates is performed in accordance with the rules prescribed in ACA.

The main features of the ASAMPL language which specify the principles of the programming paradigm for multimodal data structures processing as the following:

1. Linking the data processing process to a timeline.
2. Synchronizing the data of different modalities.
3. Complex representation of multimodal data using tuples and aggregates.
4. Focusing on the simultaneous use of various multimodal data sources (data streams from remote sensors, media files from cloud storages, etc.).
5. Dynamic linking external libraries, renders (a renderer is a software tool for reproducing the data of a certain modality), handlers for encoding / decoding, data processing and reproduction for each modality, use of specific file formats and predefined devices.
6. Mathematical processing of data aggregates based on ACA.

A program code is ASAMPL consists of blocks starting with the following keywords:

– ACTIONS, it defines data processing logic;

– AGGREGATES, it defines aggregates composed from defined tuples;

– ELEMENTS, it defines variables and their belonging to defined sets;

– HANDLERS, it defines built-in and external modules for data converting;

– LIBRARIES, it defines built-in and external libraries;

– PROGRAM, it starts a program code;

– RENDERERS, it defines built-in and external modules for data rendering;

– SETS, it defines sets which contain data elements;

– SOURCES, it defines data sources where multimodal data is stored, recorded, generated, etc.

– TUPLES, it defines tuples and their belonging to defined sets.

The program structure is as follows:

```
Program <name> {
  Libraries {
    <library1> is <path-
ToLibray1>;
    <library2> is <pathToLi-
brary2>;
    …… }
  Handlers {
    <handler1> is
<pathToHandler1>;
    <handler2> is
<pathToHandler2>;
    …… }
  Renderers {
    <renderer1> is <pathToRender-
er1>;
    <renderer2> is <pathToRender-
er2>;
    …… }
  Sources {
    <source1> is <pathToSource1>;
    <source2> is <pathToSource2>;
    …… }
  Sets {
    <set1> is <type1>;
    <set2> is <type2>;
    …… }
  Elements {
    <element1> is <set>;
    <element2> = <value>;
    …… }
  Tuples {
    <tuple1> = <set1>;
    <tuple2> = <set2>;
    …… }
  Aggregates {
    <aggregate1> = [<tuple1>, …,
<tupleM>];
    <aggregate2> = [<tupleK>, …,
<tupleN>];
    …… }
  Actions {
    <list of statements>   } }
```

The logic of multimodal data structures timewise processing is set by the operators which are based on the mathematical concept defined in ASA, in particular, logical

and ordering operations on aggregates and frequency and interval relations of discrete intervals as well as the concept of a multi-image [3–5].

To implement the multimodal data timewise processing logic, the ASAMPL specification includes the following operators: TIMELINE; SEQUENCE; SUBSTITUTE FOR WHEN; DOWNLOAD FROM WITH; UPLOAD TO WITH; RENDER WITH.

To define the semantics of the ASAMPL operators, the axiomatic approach is used in this research [8, 9]. An axiomatic specification is defined as the following derivation rule:

$$\frac{H_1, H_2, \ldots, H_n}{H},$$

where H is true if $\forall\ H_i$ is true, $i = [1..n]$.

TIMELINE operator enables timewise data processing by allowing a programmer to apply a certain action during a defined period of time. All actions included into the operator body are to be executed simultaneously. This operator has three options for the timeline defining.

The first option requires an evident indication of both the beginning and the end time moments of the timeline:

```
TIMELINE time1 : step : time2
{ list of actions to be carried
out simultaneously }
```

The semantics rule for the first option of TIMELINE operator is:

$$\frac{\{P \,\&\, (a \le x \le (a + \Delta a \cdot n))\}\, C\, \{Q\}}{\{P\}\ \text{TIMELINE}\ a : \Delta a : (a + \Delta a \cdot n)\ C\ \{Q\}}.$$

The second option allows a programmer to specify an arbitrary tuple of time values which correspond to specific time moments when the actions from the given list need to be carried out:

```
TIMELINE AS time_tuple { list of
actions to be carried out simul-
taneously }
```

The semantics rule for the second option of TIMELINE operator is:

$$\frac{\{P \,\&\, (x \in [a_1 .. a_n])\}\, C\, \{Q\}}{\{P\}\ \text{TIMELINE AS}\, [a_1 .. a_n]\ C\ \{Q\}}.$$

The third option of the TIMELINE operator enables using a condition which triggers the completion of the given actions fulfilment:

```
TIMELINE UNTIL condition
{ list of actions to be carried
out simultaneously }
```

The semantics rule for the third option of TIMELINE operator is:

$$\frac{\{P \,\&\, B\}\, C\, \{Q\}}{\{P\}\ \text{TIMELINE UNTIL}\ B\ C\ \{Q\}}.$$

SEQUENCE operator enables sequential processing of actions as one compound action:

```
SEQUENCE { list of actions to be
carried out sequentially }
```

The semantics rule for this operator is as follows:

$$\frac{\{P\}\, C_1\, \{Q\},\ \{Q\}\, C_2\, \{R\}}{\{P\}\ \text{SEQUENCE}\ C_1; C_2\ \{R\}}.$$

SUBSTITUTE FOR WHEN operator enables replacement of one data set by another one if a certain condition becomes true:

```
SUBSTITUTE name1 FOR name2
WHEN logical_expression.
```

This operator can be useful in the case when, for example, high-resolution data cannot be downloaded, uploaded, or processed because of insufficient data rates and, thus, they can be replaced with low-resolution data to allow an application to be executed even in such inappropriate conditions.

The semantics rule for this operator is as the following:

$$\frac{\{P \,\&\, B\}\, C_1\, \{Q\},\ \{P \,\&\, \overline{B}\}\, C_2\, \{Q\},\ C_i \equiv (d := d_i)}{\{P\}\ d\ \text{SUBSTITUTE}\ d_2\ \text{FOR}\ d_1\ \text{WHEN}\ B\ \{Q\}}$$

where $i = [1, 2]$.

DOWNLOAD FROM WITH operator enables downloading data from a specified data source by using either a predefined handler, or a given one. In the first case, the operator has the following format:

```
DOWNLOAD data_name FROM source
```

Its semantic rule is as follows:

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\ \text{DOWNLOAD}\ d\ \mathit{FROM}\ d_1\ \{Q\}}.$$

In the second case, the handler identifier needs to be indicated directly:

```
DOWNLOAD data_name FROM source
WITH handler
```

Its semantic rule for this case is:

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\ \text{DOWNLOAD}\ d\ \mathit{FROM}\ d_1\ \text{WITH}\ z\ \{Q\}}.$$

UPLOAD TO WITH operator allows to upload any data (an element, a tuple, an aggregate) to a defined resource by using either a predefined handler, or a given one. In the first option, there is no need to indicate the handler because it is predefined:

```
UPLOAD data_name TO destination
```

The semantic rule for the first option is the following:

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\ \text{UPLOAD}\ d\ \mathit{TO}\ d_1\ \{Q\}}.$$

In the second option, the handler needs to be indicated evidently:

```
UPLOAD data_name TO destination
WITH handler
```

The semantic rule for the second option is the following:

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\ \text{UPLOAD}\ d\ \mathit{TO}\ d_1\ \text{WITH}\ z\ \{Q\}}.$$

RENDER WITH operator enables data reproduction. Each data modality is preprocessed and rendered by its specific render. The form of this operator is as follows:

```
RENDER data_name WITH render_name
```

The semantic rule for this operator is the following:

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\ \text{RENDER}\ d\ \text{WITH}\ z\ \{Q\}}.$$

Besides, ASAMPL employs a branch statement IF THEN; a selection statement CASE OF; an assignment statement IS.

Thus, the logic of the data processing is based on these basic actions. The method of ASAMPL code translation is presented in the next section.

2. Compilation approach

The compiler implementation language needs to be strongly typed high-level general-purpose programming language with simple and expressive syntax. Automatic memory management is desired but not mandatory. Besides, this language should be popular and mature enough because popular languages tend to have bigger community, comprehensive documentation, many ready-to-use program components, frameworks and libraries. The analysis of top programming languages [10, 11] has been resulted in selecting Python.

The parser and lexer compiler components can be generated from the formal language specification, thus, a generator with Python target language support was needed to be selected. Another important criteria included versatility, separation of output code from grammar and comprehensive documentation. For this research, ANTLR [12] has been selected. A viable alternative to it is PLY [13].

Multimedia is the third base component of the implementation, cross-platform and open-source projects with Python bindings were preferred. We have selected GStreamer [14] because its core does not depend on the type of processed data and it can be extended by plugins, which enable adding new formats, protocols or hardware support.

The architecture of the implemented two-pass compiler includes frontend and backend (Fig. 1).

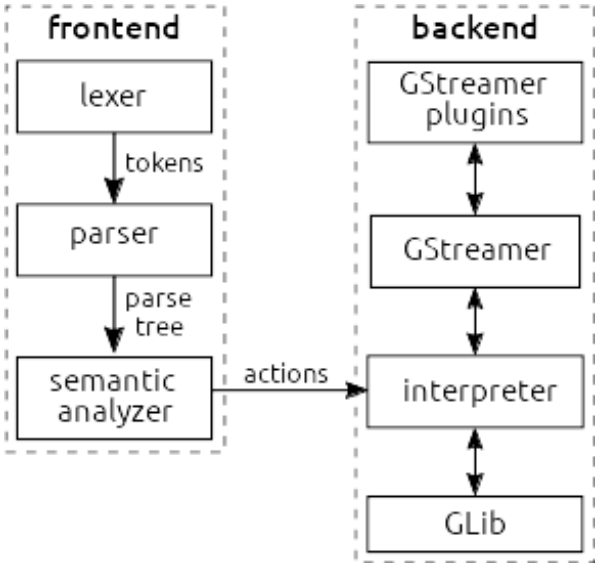

Fig. 1. Compiler architecture

2.1. Frontend. The frontend performs syntax and semantics verification (e.g., identifier availability, type checking). The lexer and parser are generated by ANTLR from the language context-free grammar description, it also generates the helper base classes for parse tree traversal: a listener and/or a visitor. Our semantic analyser implements the visitor design pattern to control the order of traversal.

All declarations for declarative sections are checked if a name they define has not been declared yet, and additional checks need to be performed for every section.

The parse tree traversal starts at the LIBRARIES section and the library importer component tries to import declared libraries as a regular Python package. The libraries which define custom types need to have *export_types* dict in their packages which maps type names to corresponding classes. If a library name has not been declared yet and its package was successfully imported, the library is registered in a corresponding lookup table and becomes available in anther program parts.

For declarations in the HANDLERS and RENDERERS sections, the visitor checks if the specified name has not declared yet, the library is imported, and it tries to import a specified element as a class in that library. If the class was imported successfully and inherits a required base class, it will be registered in the lookup table.

In SETS section, type import declarations are checked if a name has not been declared yet, the specified library was imported and has specified type, in that case the imported type is registered in the lookup table. Type alias are registered only for already present types if the name defined by an alias has not been used before.

The declarations in the ELEMENTS, TUPLES and AGGREGATES sections assign names to certain values in the lookup table when a specified type name or a referenced value name is defined.

The ACTIONS section declarations define program logic and data flows, they are converted to a hierarchy of action objects which represent specific discrete tasks.

The interface defined by the base action class includes methods *Start*, *Stop* and *Stop_Prepare*, *Finished* signal and *Depth* property (a depth within the hierarchy tree). The *Start* method returns False when the action finished immediately, otherwise it returns True. The *Stop_Prepare* method is used to notify an action before it is actually stopped with the *Stop* method call. The *Finished* signal is emitted only when the action finished neither during *Start* nor during *Stop* method calls.

Another important abstract base class is *CompositeAction*: a group of actions which can be treated as a single action and may define custom execution order. There are two concrete implementations: *ChinedAction* which executes child actions sequentially and *ParallelAction*, which executes child actions concurrently. When they are stopped, they stop child actions. The *ChainedAction* finishes execution only when all children have finished. The *ParallelAction* behaves in the same way when execution duration is not specified or when early finish is allowed, otherwise it continues execution even when all children have finished already and stops unfinished when the data processing time elapses.

To create a delay in the beginning of actions (e.g., in TIMELINE statements with non-zero start time), the *DelayAction* is used.

The behaviour which defines an update frequency and an elapsed time tracking is extracted into *ActionUpdateStrategy* subclasses and can be used in certain classes with help of the Dependency Injection design pattern. The base interface provides *update* method which receives a time difference for a current update step (for base class it updates an object age) and properties to get a current update interval duration, a next update delay, an overall duration, an age and an update loop finish status. All time values are stored with microsecond precision. There are two concrete implementations: *FixedStepUpdate* which has equal update intervals and *FramesTupleUpdate* which accepts a list of update intervals. The *DelayAction* objects instantiate and encapsulate *FixedStepUpdate* objects with an overall duration and an update interval duration equal to the specified delay. The *ParallelAction* can be initialized with any concrete implementation of *ActionUpdateStrategy* to define the action duration and the update frequency.

The *DownloadAction*, *UploadAction*, and *RenderAction* implement corresponding program statements DOWNLOAD FROM WITH, UPLOAD TO WITH, and RENDER WITH. They check compatibility of handlers and renderers with user-specified data stream objects, establish connections at the action start and disconnect them when stopped of finished execution. Additionally, *DownloadAction* and *UploadAction* create GStreamer pipeline element that handles input (or output) to a supplied source (or target) URL.

The action hierarchy is built by *TimelineBuilder* object and helper classes. Every TIMELINE statement block and root code block are treated as a separate time interval and are represented by separate *_Timeline* object within *TimelineBuilder*. These objects encapsulate checks for detecting duplicated data streams in lists, cases when the same data stream has different target or source declared for the same execution time, ensure compatibility between specified data streams, handler (or renderer) and action.

TimelineBuilder holds at least one *_Timeline* instance: for the root code block. Entering to a new TIMELINE statement block will create a new *_Timeline* instance with a parent object which becomes current while the previous one becomes its parent. Each *_Timeline* object has an initial mode set to "in parallel", but when the code block marked with SEQUENCE keyword is entered, it is switched to "in sequence". On exit from the top-level SEQUENCE block, all created actions within that block are put into wrapper *ChainedAction* (this step is skipped for single action) and are added to the actions list of this *_Timeline*. When TIMELINE statement block is exited, the current *_Timeline* instance is set to its parent. Empty TIMELINE blocks are ignored, but when time range is set and parent *_Timeline* has mode "in sequence", *DelayAction* is added instead to the parent. When time range is not specified for the current TIMELINE statement block and the parent has "in parallel" mode, all actions of the current *_Timeline* are pushed to the parent as they are. In all other cases they are put to *ParallelAction* which has an appropriate *ActionUpdateStrategy* set if needed. When non-zero start time is specified, *DelayAction* is created with the corresponding delay set. If parent has "in sequence" mode or start time is zero *ParallelAction* with *DelayAction* are pushed as is to the parent. Otherwise, they are wrapped with *ChainedAction* before they are added to the parent.

2.2. Backend. The backend provides runtime environment and runtime error handling, schedules execution of the action hierarchy received from the frontend stage. The action hierarchy is represented here by a single root *ParallelAction* object. The core component of the backend is the scheduler object *UpdateSource* which attach itself to GLib main context as an event source to receive updates.

Actions which need updates (e.g., *ParallelAction* and *DelayAction*) hold reference to the scheduler object and subscribe for updates when needed. The scheduler holds the priority queue of subscribed actions where priority determined by both the next update time in microseconds and the action depth within the hierarchy tree. This ensures that the parent action receives update before its children when they have the same update time. But for sibling actions update order (and, as a

result, the start order) is not guaranteed.

The scheduler update loop has three distinct stages (Fig. 2): *prepare*, *wait*, and *dispatch*.

The *prepare* stage determines the closest update time for subscribed actions and calculates a delay for the *wait* stage. On the first run, this stage starts a root action and the scheduler subscribes to the root action finished signal if it did not finish immediately, otherwise the *wait* stage is skipped.

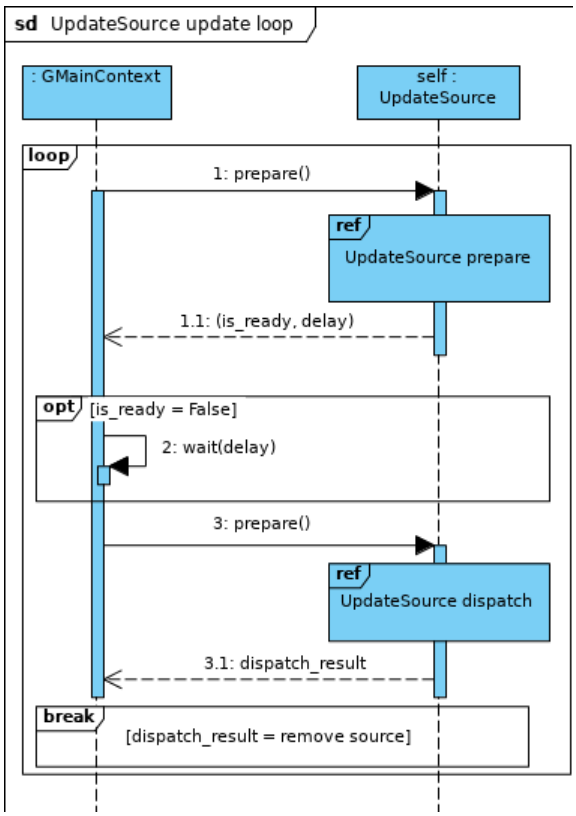

Fig. 2. Scheduler update loop

The *dispatch* stage pulls actions which reached their update time from a priority queue. The actions update method receives a defference between planned and actual update time and returns a Boolean value. An action which still needs updates is added back to the scheduler priority queue. At the end of this stage, *on_update_unsubscribed* handler is called for all unsubscribed actions in the order they received updates. If the root action is finished during or before the dispatch stage, the scheduler removes itself from GLib main context and the program finishes its execution.

2.3. Standard library. The standard library provides base classes and reference implementations of handlers, renderers and data streams. Handlers and renderers implement *BaseHandler* class, its interface define methods for attaching and detaching to data streams, compatibility checking, starting and stopping operation, signal *done* to notify subscribed objects. Renderers are always used as destination for data streams, while handlers can implement either source or destination. Depending on the implementation, they can be attached to single or multiple data streams at the same time (e.g., handlers for multiplexed data formats). The reference handler implementations are classes OGGIn and OGGOut which work with files in OGG container format and support the following data stream types: audio, video, and subtitles. Both handlers try to automatically find the required coder or decoder among GStreamer plugins when attached to uncompressed data streams.

Standard renderers are represented by *AudioOutput* and *VideoOutput* classes which implement playback of uncompressed audio and video. Data streams can have at most one source and multiple receivers (handlers and renderers). The base data stream class *BaseStream* provides an interface with the methods for attaching and detaching source and receiver objects, retrieving information about supported data types for its inputs and outputs, signals to notify when no outputs are present anymore or first output attached. The standard library provides implementation of uncompressed audio and video streams: *RawAudio* and *RawVideo*.

All these components have own GStreamer elements connected into pipelines. During runtime actions they are connected together to ensure data flow described by the program.

3. Results and discussion

In order to evaluate the efficiency of the implemented solution we compared it with traditional development tools, namely, C compiler from GNU Compiler Collection suite [15] and GStreamer framework. For this purpose, we created two sets of equivalent small- and medium-sized programs written in both C and ASAMPL. To evaluate performance of the implemented compiler, we

measured overall running time and peak memory consumption during program code parsing, analysis, and preparation program code to execution.

Table 1

Average program code size

Program code	Small size (lines of code)	Medium size (lines of code)
C	193	318
ASAMPL	43	95

For running time measurement, we used Perf tool [16], running each compiler for every program 1000 times. Averaged results for both program sets (see Table 2 and Fig. 3b) show that execution time of GCC C compiler is around 30 % lower.

Table 2

Average execution time

Program code	Small size (secs and stddev)	Medium size (secs and stddev)
GCC	0.220516528 (0.34%)	0.23745095 (0.34 %)
ASAMPL compiler	0.315360984 (0.53 %)	0.330313517 (0.29 %)
Difference	30 %	28 %

The results of peak memory consumption measurement with Valdgrind tool [17] (see Table 3 and Fig. 3c) show that on average it is 33% lower for ASAMPL compiler.

ASAMPL programs are visually simpler, concise, and expressive since they omit resource management, pipeline manipulation, synchronization and other tasks required to set up an environment, but not related to actual data processing task. As a result, the program size in lines-of-code (excluding comments and whitespace) for ASAMPL is at least 3 times smaller (see Table 1 and Fig. 3a).

Table 3

Peak memory consumption

Program code	Small size (Mbytes)	Medium size (Mbytes)
GCC	14.7	14.7
ASAMPL compiler	9.62	9.8
Difference	35 %	33 %

.

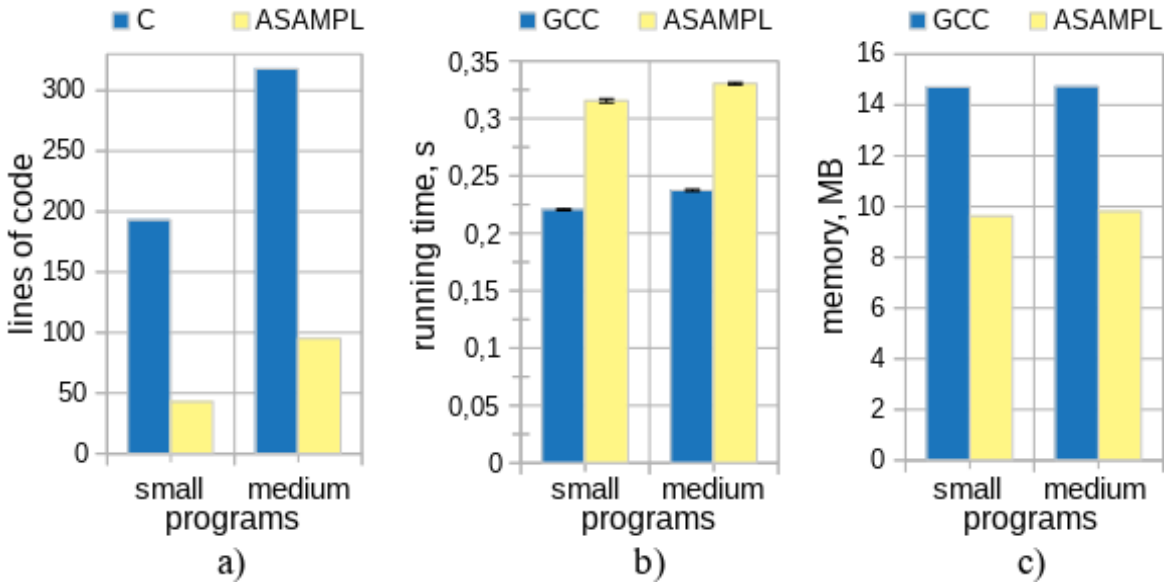

Fig. 3. Benchmarking results comparison charts

Conclusion

The programming language ASAMPL is a domain-specific language driven by data modality and time. It is developed for easy and effective processing of multimodal data defined with respect to time. To present such data, the programming language specification includes data structures called tuples and aggregates. ASAMPL offers a set of special-purpose operators, semantics of which is presented in the paper. To execute program code in ASAMPL, a complier has been developed. The paper presents both the architecture of this compiler and the compilation approach. The testing results show that ASAMPL code is simpler, concise, and expressive as well as memory consumption is lower for the developed ASAMPL compiler. Thus, ASAMPL can be considered as an appropriate option when the memory usage and the code simplicity are important issues.

References

1. Virtual Reality for Education. Available from: http://virtualrealityforeducation.com/ [Accessed 05/02/2020].
2. Antonov S., Antonova R., Spassov K. Multimedia Applications in Education. *Smart Technologies and Innovation for a Sustainable Future*. Springer. 2019. P. 263–271.
3. Weng C., Rathinasabapathi A., Weng A., Zagita C. Mixed Reality in Science Education as a Learning Support: A Revitalized Science Book. *Journal of Educational Computing Research*. 2018. 57(3). P. 777–807.
4. Sulema Y. ASAMPL: Programming Language for Mulsemedia Data Processing Based on Algebraic System of Aggregates. *Interactive Mobile Communication Technologies and Learning*. Springer. 2018. P. 431–442.
5. Dychka I., Sulema Ye. Logical Operations in Algebraic System of Aggregates for Multimodal Data Representation and Processing. *KPI Science News*. 2018. Vol. 6. P. 44–52.
6. Dychka I., Sulema Ye. Ordering Operations in Algebraic System of Aggregates for Multi-Image Data Processing. *KPI Science News*. 2019. Vol. 1. P. 15–23.
7. Sulema Ye., Kerre E. Multimodal Data Representation and Processing Based on Algebraic System of Aggregates, preprint. 2020. 37 p.
8. Milner R. Operational and Algebraic Semantics of Concurrent Processes. *Formal Models and Semantics*. 1990. P. 1203–1242.
9. Roşu G., Ştefănescu A. Towards a Unified Theory of Operational and Axiomatic Semantics. *Automata, Languages, and Programming*. Springer. 2012. P. 351–363.
10. TIOBE The Software Quality Company. Available from: https://www.tiobe.com/tiobe-index/ [Accessed 05/02/2020].
11. The Top Programming Languages 2019. IEEE Spectrum. Available from: https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019 [Accessed 05/02/2020].
12. ANTLR. Available from: https://www.antlr.org/ [Accessed 05/02/2020].
13. PLY (Python Lex-Yacc). Available from: https://www.dabeaz.com/ply/[Accessed 05/02/2020].
14. GStreamer. Available from: https://gstreamer.freedesktop.org/ [Accessed 05/02/2020].
15. GCC, the GNU Compiler Collection. Available from: https://gcc.gnu.org/ [Accessed 05/02/2020].
16. Perf Wiki. Available from: https://perf.wiki.kernel.org/index.php/Tutorial [Accessed 05/02/2020].
17. Valgrind's Tool Suite. Available from: https://valgrind.org/info/tools.html [Accessed 05/02/2020].

Література

1. Virtual Reality for Education. Available from: http://virtualrealityforeducation.com/ [Accessed 05/02/2020].
2. Antonov S., Antonova R., Spassov K. Multimedia Applications in Education. Smart Technologies and Innovation for a Sustainable Future. Springer. 2019. P. 263–271.
3. Weng C., Rathinasabapathi A., Weng A., Zagita C. Mixed Reality in Science Education as a Learning Support: A Revitalized Science Book. Journal of Educational Computing Research. 2018. 57(3). P. 777–807.
4. Sulema Y. ASAMPL: Programming Language for Mulsemedia Data Processing Based on Algebraic System of Aggregates. Interactive Mobile Communication Technologies and Learning. Springer. 2018. P. 431–442.
5. Dychka I., Sulema Ye. Logical Operations in Algebraic System of Aggregates for Multi-

modal Data Representation and Processing. KPI Science News. 2018. Vol. 6. P. 44–52.
6. Dychka I., Sulema Ye. Ordering Operations in Algebraic System of Aggregates for Multi-Image Data Processing. KPI Science News. 2019. Vol. 1. P. 15–23.
7. Sulema Ye., Kerre E. Multimodal Data Representation and Processing Based on Algebraic System of Aggregates, preprint. 2020. 37 p.
8. Milner R. Operational and Algebraic Semantics of Concurrent Processes. Formal Models and Semantics. 1990. P. 1203–1242.
9. Roşu G., Ştefănescu A. Towards a Unified Theory of Operational and Axiomatic Semantics. Automata, Languages, and Programming. Springer. 2012. P. 351–363.
10. TIOBE The Software Quality Company. Available from: https://www.tiobe.com/tiobe-index/ [Accessed 05/02/2020].
11. The Top Programming Languages 2019. IEEE Spectrum. Available from: https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019 [Accessed 05/02/2020].
12. ANTLR. Available from: https://www.antlr.org/ [Accessed 05/02/2020].
13. PLY (Python Lex-Yacc). Available from: https://www.dabeaz.com/ply/[Accessed 05/02/2020].
14. GStreamer. Available from: https://gstreamer.freedesktop.org/ [Accessed 05/02/2020].
15. GCC, the GNU Compiler Collection. Available from: https://gcc.gnu.org/ [Accessed 05/02/2020].
16. Perf Wiki. Available from: https://perf.wiki.kernel.org/index.php/Tutorial [Accessed 05/02/2020].
17. Valgrind's Tool Suite. Available from: https://valgrind.org/info/tools.html [Accessed 05/02/2020].

Received 06.02.2020

About the authors:

Yevgeniya Sulema,
candidate of tech. sciences (PhD),
associate professor at Computer Systems Software Department.
The number of publications in Ukrainian and foreign journals is over 130.
Hirsh index is 4.
https://orcid.org/0000-0001-7871-9806,

Vladyslav Glinskii,
Master student at Computer Systems Software Department.
https://orcid.org/0000-0003-1360-7218.

Affiliation:

Igor Sikorsky Kyiv Polytechnic Institute,
03056, Kyiv,
pr. Peremogy, 37, build. 15.
Tel.: +38 044 204 99 44.

Email: sulema@pzks.fpm.kpi.ua