DENOTATIONAL SEMANTICS OF A MINIMAL STACK-BASED LANGUAGE WITH A SLANG SPECIFICATION SKETCH

William Steingartner [1], ***Valerie Novitzká*** [2]

[1,2]*Faculty of Electtrical Engineering and Informatics, Technical University of Košice*
Košice, Slovakia
[1]william.steingartner@tuke.sk, [2]valerie.novitzka@tuke.sk

Abstract. This paper presents a minimal stack-based domain-specific language together with its denotational semantics. The language consists of a small set of instructions manipulating a stack of natural numbers. Each program is interpreted as a function transforming program states, which allows for a mathematically precise description of computation. The semantic model is compositional: the meaning of a sequence of instructions is defined by function composition. This provides a clear and concise illustration of how simple programming constructs can be modeled using standard mathematical structures such as functions and sequences. In addition, the paper briefly outlines how the language can be specified in the SLANG system, demonstrating how formal semantic descriptions can support the development of interpreters. The proposed model is intentionally minimal and serves primarily as an illustrative example of mathematical modeling in computer science.

Keywords: **denotational semantics, domain-specific language, executable semantics, mathematical modeling, rapid prototyping, stack-based language**

1. Introduction

Mathematical modeling plays a fundamental role in theoretical computer science. It provides a precise and systematic framework for understanding, designing, and controlling systems, as well as for effective problem solving. In particular, formal semantics of programming languages is an area where mathematical models are used to precisely describe the behavior of programs. By employing well-defined mathematical structures such as sets, relations, and functions, programs can be interpreted as objects of formal analysis [7].

Such an approach enables rigorous reasoning about program properties, including determinism, correctness, and termination. Moreover, it provides a solid foundation for the design and implementation of programming languages and their compilers [1, 10].

One of the natural ways to model computation is to interpret a program as a transformation of a system state [4, 9]. In this paper, we focus on a simple yet expressive model based on stack manipulation. Stack-based languages are particularly suitable for this purpose due to their structural simplicity and direct correspondence between syntax and state transformation.

The aim of this paper is to define a minimal stack-based language and provide its denotational semantics. The meaning of a program is represented as a function transforming a stack. This compositional approach allows complex programs to be interpreted as compositions of simpler semantic functions. The model is intentionally minimalistic to emphasize the clarity of the underlying mathematical structure and to support a precise mathematical modeling of program meaning.

Stack-based languages offer a convenient setting for studying optimization techniques [2], as their simple instruction sets contrast with the complexity of efficient stack management; at the same time, they are widely used in practice, for example in virtual machines and in syntax-driven interpretation frameworks [3].

Finally, we briefly demonstrate how the proposed language can be described using the SLANG framework, which supports executable semantic specifications.

2. Syntax of the Language

The language consists of a finite set of instructions operating on a stack of natural numbers. A program is defined as a finite sequence of instructions. The syntax of the language is defined by the following grammar:

ins ::= `load` *n* | `add` | `sub` | `mul` | `dup` | `swap`
code ::= ε | *ins code*

where $n \in \mathbf{N}$. The symbol ε denotes the empty program, while the single space represents sequential composition of instructions. The (intuitive) meaning of the instructions is as follows:

- `load` *n* — pushes a natural number *n* onto the stack,
- `add` — removes the top two elements and pushes their sum,
- `mul` — removes the top two elements and pushes their product,
- `dup` — duplicates the top element of the stack,
- `swap` — swaps the top two elements of the stack.

Programs are constructed using sequential composition. Formally, a program can be defined inductively as empty program, a single instruction, or as a sequence of programs. For simplicity, we assume that programs are well-formed, i.e., operations such as `add` and `mul` are only applied when the stack contains a sufficient number of elements. The basic specification was inspired by a simple task of visualizing a computation and by exercises related to a basic automaton whose lan-

guage contained only four instructions: `add`, `sub`, `mul`, and `load` *a*. The original exercise is presented in [8].

In our version, the only modification concerns the representation of the stack. We adopt a notation in which the top of the stack is always displayed on the left in the linear representation. This convention is consistent with approaches commonly used in the literature (see, for example [4]).

3. Semantic Domains

The semantic model is based on the notion of a stack. Let **N** denote the set of natural numbers (zero included). The set of all possible stack states is defined as a sequences of elements from **N**:

$\mathbf{State} = \mathbf{N}^*$.

The empty stack is denoted by ε. If n is a natural number and s is a stack, then the stack with n on top of s is written as $n :: s$. Thus, a stack is a finite sequence where the leftmost element represents the top of the stack.

4. Denotational Semantics

Denotational semantics (also known as mathematical semantics) defines the meaning of programs as mathematical objects [5], typically functions transforming program states. The meaning of a program is defined as a function that transforms a stack into another stack. We define the semantic function as:

$$[\![P]\!] : \mathbf{State} \to \mathbf{State}_\perp$$

to account for the possibility that the resulting state may be undefined. Here, $\mathbf{State}_\perp$ denotes the domain extended with an undefined element $\perp$. This function is defined compositionally, based on the structure of the program. Following is the definition of denotational semantics of the instructions:

$$[\![\texttt{push}\ n]\!]\ (s) = n :: s$$
$$[\![\texttt{add}]\!]\ (n :: m :: s) = (n + m) :: s$$
$$[\![\texttt{mul}]\!]\ (n :: m :: s) = (n \times m) :: s$$
$$[\![\texttt{dup}]\!]\ (n :: s) = n :: n :: s$$
$$[\![\texttt{swap}]\!]\ (n :: m :: s) = m :: n :: s$$

If the semantic function is not defined for a given input (e.g., due to insufficient stack elements), it yields the bottom value $\perp$.

For two programs *P1* and *P2*, their sequential composition is interpreted as function composition:

$$[\![P1\ P2]\!] = [\![P2]\!] \circ [\![P1]\!].$$

This means that program *P1* is executed first, followed by program *P2*. This compositional definition of program semantics immediately yields the following property.

Proposition 1.
The semantics of the language is deterministic, i.e., for every program *P* and every state *s*, the value $[\![P]\!](s)$ is uniquely defined.

Proof (sketch).
The result follows directly from the definition of the semantic function, since each instruction defines a deterministic transformation of the stack. The composition of deterministic functions is also deterministic.

5. Example

As an example, we consider the following program:

```
load 2 load 3 add load 4 mul
```

This program corresponds to the infix expression (2 + 3) * 4, which is expressed in reverse Polish notation (postfix) as 2 3 + 4 *. For the arithmetic fragment of the language, this notation can be obtained by omitting load and replacing arithmetic instructions such as `add` and `mul` with the corresponding operators + and *. This correspondence is only partial: stack-manipulation instructions such as `dup` and `swap` have no direct counterpart among standard arithmetic operators and are therefore interpreted as transformations of the stack rather than as components of an infix expression.

Figure 1 illustrates the evaluation of the program starting from the empty stack. The instruction `load 2` creates the stack ⟨2⟩, `load 3` yields ⟨3 | 2⟩, and `add` replaces the two top elements by their sum. After loading 4, the instruction `mul` multiplies the two remaining elements and produces the final stack ⟨20⟩. Thus, the program computes the value of the expression (2 + 3) * 4. We note that in Figure 1, square brackets are used to denote stack contents, while in the text we use angle brackets ⟨·⟩ to avoid confusion with bibliographic references.

6. SLANG Specification Sketch

The presented language can be naturally specified using the SLANG system [6], which supports formal definitions of programming languages and their semantics.

In SLANG, we define a syntactic domains for the language, and a semantic function mapping programs to state transformations. SLANG is a tiny but powerful tool for rapid prototyping and developing the interpreter of given language.

For example, a part of the definition of the semantic function in SLANG (generating the Java code) for instruction `load` is as follows:

```
function S ⟦Command⟧ : #Stack<Integer># -> #Stack<Integer># {
   equation S ⟦Add⟧ (s0) = s
       {
              a: #Integer# = #s0.pop();#;
              b: #Integer# = #s0.pop();#;
              #s0.push(a+b);#
              #s = (Stack<Integer>) s0.clone();#
       }
```

Similarly, other instructions can be defined using pattern-based rules. The compositional nature of denotational semantics aligns well with the capabilities of SLANG, enabling automatic generation of interpreters from formal specifications.

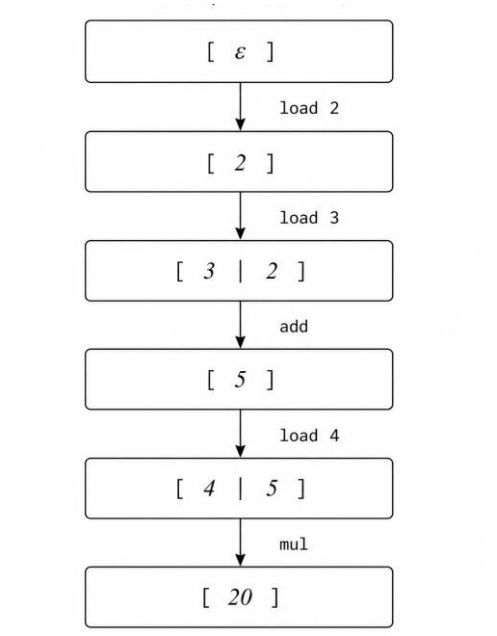

Fig. 1. Evaluation of the expression (2+3)*4 on the stack

7. Discussion and Conclusion

The paper presented a minimal stack-based language together with its denotational semantics. The model demonstrates how even very simple programming constructs can be described using precise mathematical tools. The compositional definition of semantics allows for straightforward reasoning about program behavior. Moreover, the representation of programs as functions over states provides a clear abstraction of computation. The proposed model can be extended in several directions. Additional instructions such as subtraction or conditional branching can be introduced. Furthermore, type systems or static analyses could be incorporated to ensure program correctness. Although the language is intentionally simple, it serves as a useful educational example and a foundation for more advanced models of computation. The presented model can be viewed as a simple instance of a state transition system, where programs correspond to state transformers.

Acknowledgement

This work was supported by KEGA project 028TUKE-4/2026 – Innovative approaches to teaching and researching formal methods in the context of modern software engineering, granted by the Cultural and Education Grant Agency of the Slovak Ministry of Education.

References

[1] Appel A. W., Palsberg J., Modern Compiler Implementation in Java, 2nd ed., Cambridge University Press, Cambridge 2002.

[2] Hernández Cerezo A., Superoptimization of Stack-Based Bytecode, PhD thesis, Universidad Complutense Madrid, 2025.

[3] Koopman P., Lubbers M., Strongly-Typed Multi-View Stack-Based Computations, Proc. PPDP 2023, pp. 1–12, doi:10.1145/3610612.3610623.

[4] H. R. Nielson, F. Nielson, *Semantics with Applications: An Appetizer*, Springer, 2007.

[5] D. A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon, 1986.

[6] W. Schreiner, W. Steingartner, *The SLANG Semantics-Based Language Generator – Tutorial and Reference Manual (Version 1.0)*, RISC Technical Report 23-13, 2023.

[7] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1993.

[8] University of Glasgow, Algorithms & Data Structures: MSc in Information Technology and MSc in Software Development – Examination Paper (2013), https://www.dcs.gla.ac.uk/~daw/teaching/ADS/Exams/paper.2013.pdf, accessed April 25, 2026.

[9] Radaković D., Herceg Đ., Towards a Completely Extensible Dynamic Geometry Software with Metadata, Computer Languages, Systems & Structures, vol. 52, pp. 1–20, 2018, doi:10.1016/j.cl.2017.11.001

[10] Mössenböck H., Compilerbau. Grundlagen und Anwendungen, dpunkt.verlag GmbH, 2024.