LaST: Language Study Tool

Enrique Molinari[1] **, Edgard Lindner**[1]

[1]Universidad de Belgrano
Federico Lacroze 1955 – +056 11 4511 4748
Buenos Aires, Argentina

{epmolina,elindner}@ub.edu.ar

Abstract. *An important challenge when studying the semantics*[2] *of programming languages is understanding the way syntactical constructions in high-level programming languages produce algorithms that are bound to the underlying virtual machine, and the way basic computer resources such as registers and memory are used and managed. This document describes an educational tool developed to assist teaching and to help students learn programming language semantics. This software tool, LaST (**La**nguage **S**tudy **T**ool) provides graphic feedback and interaction, allowing the study of the semantics of programming languages while visualizing their internal machine representation, and the actions triggered by certain high-level instructions. LaST includes four different high-level languages, and an assembly-like language, running in a user interface that allows source code and assembly code highlighting, step-by-step execution and graphical representation of computer resources. All these provide a rich learning environment that help the instructor explaining programming language related subjects, and allows the students to construct a more dynamic and integrated vision of them.*

Keywords: *computer education technology, computer science education, programming language semantics, teaching programming languages.*

1. Introduction

1.1. Motivation and Goals

Research studies of human intelligence and its development, show that the comprehension of abstract concepts is a difficult task. It is a really important challenge for an instructor to help create a concrete manifestation of these concepts.

Experience shows that new methods are needed for teaching subjects such as Computer Language Concepts, Programming Languages Paradigms, and Comparative Study of Programming Languages. As an example, in the medical field, technology supports the teaching process providing software which can simulate the behavior of the human body and its organs, facilitating the work of the instructor and allowing the students to construct a more dynamic and integrated vision.

Along these lines, the authors present a tool, LaST (Language Study Tool), which allows the study of programming languages concepts while visualizing the internal machine representation of each of them, and the machine's state transformations generated by high-level instructions.

[2]Throughout this document we refer as "semantics" to *dynamic semantics*, as it is described in [2], following the operational semantics approach: to describe the semantics of programming language constructs in terms of their effects in a virtual machine or abstract processor.

LaST design and implementation goal is twofold. First, to help students create a concrete visual manifestation of programming language concepts. Second, to assist instructors explaining programming language semantics and accommodate the different learning styles that are encountered in the classroom.

LaST is suitable for use in programming language concepts and/or programming language implementation courses.

1.2. Previous Work

The work presented here took as its starting point the proposal made by Ghezzi and Jazayeri in their book Programming Language Concepts [2], to use a basic abstract processor called Simplesem for the study of the semantics of programming languages. The purpose of this abstract processor is to describe the effects caused by a certain high-level syntactical construction in an underlying virtual machine, or in a real computer.

To better understand LaST, it is useful to mention the classification proposed by Ghezzi and Jazayeri to study the run-time structure of programming languages. This is a family of 5 language types called C1 to C5, which are based on simplified variants of the C programming language [4].

Based on this classification this document presents an open-source software application which translates high-level programs into Simplesem abstract code and graphically displays the processes which take place during their execution.

2. LaST: Language Study Tool

2.1. LaST in a Nutshell

LaST consists of a Simplesem language interpreter, four compilers, a source code editor, an integrated debugger and a component which displays graphically and dynamically the way conputer resources are used and managed. The four compilers correspond to the C2-C5 classification of programming languages provided by Ghezzi and Jazayeri. However, this number can be increased, since the application has been developed using an object-oriented approach, which allows LaST to be easily extended with new compilers. A new compiler to be added to LaST must implement a set of provided interfaces in order to access LaST components and features [6].

LaST was developed using the Java programming language [5]. The decision to use this particular language to implement the tool was based in Java's internet capabilities. As a Java/Swing application, LaST can be used not only as a stand-alone program but also as an Applet, which can be accessed through the internet, allowing instructors to embed it in their course's or personal web site, or inside web-enabled e-learning applications. LaST applet is available at [6].

Finally, it is the intention of the authors to put LaST source code available in the internet under the Open Source license model [3]. This will allow the tool to be further developed and polished.

2.2. User Interface and Features

LaST's user interface has been carefully designed to provide a comfortable environment, in which the user can easily run and debug simple programs, while constantly displaying the contents of the memory and the effects each language construction produces in them.

To achieve this goal, the user interface has been divided in four main panels: the source code editor panel, the Simplesem code panel, the memory views panel and the output panel. All panels

can be individually hidden by the user to avoid displaying information which may be unnecessary in specific situations. Section 5 (Example section) of this document contains 4 screen shots of LaST's user interface.

LaST's main features are described next, along with their main purpose and utility for both students and instructors.

2.2.1. Source Code Editor

A program written in any of the languages supported by LaST can be edited directly in the editor or loaded from a text file. Basic operations such as *cut*, *paste* and *copy* are supported via a context menu.

Syntax Highlighting: Syntax highlighting has become a very common feature, present in almost every Integrated Developer Environment or programmer-oriented text editor. This technique allows programmers to understand and interpret code better and faster. LaST supports syntax highlighting in its source code editor, identifying tokens such as reserved words, operators, and constants. Custom-made compilers designed to work in LaST must specify which tokens must be treated as reserved words and which as operators, implementing a common interface.

2.2.2. Simplesem Code Panel

Simplesem code is either automatically generated by the compiler or loaded from a file if the user needs to use a Simplesem program directly without the corresponding high level code.

2.2.3. Memory Panel

Memory Segments: LaST allows the user to rapidly identify if a memory cell is being used as part of an activation record, heap data, static data or a register, using different background colors in each case.

Memory Size: When LaST starts, its memory segment contains 256 cells. While this can be enough for most example programs, there may be more complex cases, requiring more memory. For this reason, the memory size can be adjusted by the user.

Activation-Record Highlighting: For stack-based languages (such as the C3, C4 and C5 languages provided with LaST) it is useful to easily identify the activation record corresponding to the program unit which is being executed. In LaST's memory views, current activation records are highlighted using yellow background, allowing the user to rapidly identify the current ambient of reference containing the unit's local variables and control variables.

Variable Names: To easily identify which variable is stored in which cell, LaST displays the variable's high-level name together with its value in the memory panel, using the format "name = value". As in some situations this can lead to the misconception by the students that a variable name is in fact stored in memory along with its data, the instructor can dynamically disable this feature.

Registers and control variables are also identified by their name, between brackets, in order to avoid confusing them with regular, user-defined variables.

Static and Dynamic Chain Drawing: Static and dynamic chain are main concepts in computer language theory. Understanding them is not a straightforward task for the students, and it is

always easier for the instructor to represent them graphically so that they are more easily acquired by the class. LaST draws an arrow pointing from each static or dynamic pointer in a program unit's activation record, to the correspondent cell which it references. Both chains can be individually turned on or off.

A language may not implement one particular chain (for example, C3 doesn't use a static chain since it's not a block-structured language), for this reason custom compilers must inform LaST which chain must be drawn in the memory panel.

Registers: Since the Simplesem abstract machine does not include registers, memory cells are used to store important control values such as the base address of the current activation record, or the stack top; the first (bottom-most) cells in Simplesem memory are used as registers. The memory segment in LaST represent register cells using green background. To visually indicate which cells are pointed by these registers, the user can also ask LaST to draw an arrow from each register to the cell it references.

Custom compilers must inform LaST the number of cells the language uses as registers.

Split Memory: LaST memory panel can be split in two segments, each one providing a different view of the same data. Features such as variable names, static and dynamic chains, and user-defined pointers can be set individually on each of the two views so the user can have two different representations of the memory contents. This is particularly useful in languages with heap management: while the first view displays the stack contents, the second view can be scrolled to the portion of the memory which is being used for the heap.

User Defined Pointers and Highlighted Cells: If a cell in Simplesem's memory contains an address, it is usually helpful to be able to rapidly visualize the cell that is pointed by it. Both the stack and heap memory views in LaST allow the user to set a pointer mark at any given cell in the memory; LaST will draw an arrow from it to the referenced cell. The user can also set a background color for any cell in both memory views. In both cases, the color used can be defined by the user.

2.2.4. Program Execution and Debugging

Execution Modes: The execution of a program in LaST can be done in two ways: completely or step-by-step. In the first case, a previously compiled program is executed without interruption from beginning to end, or until a breakpoint is reached. This is useful in case the results of the program execution (program output) are important, but not the execution process itself. Step-by-step execution, on the other hand, allows the user to run a program one instruction at a time, allowing the user to observe and analyze the processes and results produced by each syntactical construction in a program.

Stepping through high-level code is particularly useful in programming language courses, in which the semantics of each high-level construction is studied, regardless of their translation to machine-level language. However, LaST also allows a program to be run in step-by-step mode through the Simplesem instructions it was translated into. This can be useful as an introduction to compiler design courses, to analyze the particular translation of high-level constructs, or to test and debug a certain compiler designed to work in LaST.

The contents of the memory are dynamically updated while executing a program in step-by-step mode.

Breakpoints: Breakpoints can be set in both the high-level source code and Simplesem code.

When running a program, execution will stop at a breakpoint, the virtual machine will be paused, and the memory state will be locked. This way, the user can inspect the contents of the memory at a specific point during the execution. Execution can then be resumed and will continue until the program terminates, or a new breakpoint is reached.

2.2.5. Output Panel

This panel is used to display program output and compile or execution-time errors.

Error Tracking: When an error occurs during compilation or execution, a message is written in LaST's output panel, using the following format:

```
[Lexical|Syntax|Semantic|Run-time] error:[line number]:  [error message]
```

Lexical, Syntax and Semantic errors are produced at compile-time when attempting to compile the source code of a program, while Run-time errors appear at execution-time when a given instruction is being interpreted by the virtual machine. This distinction is done by LaST as a learning facility, to inform the user exactly during which step of the compiling process the error occurred.

For both compile-time and execution-time errors, the user can click in the message inside the output panel to ask LaST to highlight the source code line when a lexical, syntax or semantic error occurs or the simplesem instruction when a run-time error occurs. The highlighted line or instruction will then be painted using red background so it can be easily identified.

2.3. Included Languages

Four compilers were developed for LaST. Each compiler implements different features that help to learn and appreciate the semantics of programming languages. The complete definition of each of these languages can be found at [6].

Next, the most important features of each one are described, adding what the authors call **"Learning Features"**: the items each compiler offers to the learning process of programming language semantics.

All four languages provide conditional (if-then-else) and loop statements (for; while) and one primitive variable type (integer). They all support structured programming using program units (procedures, functions), and require a main program unit (program). The syntax of each language was carefully designed to ensure that the instructors and students can write programs easily.

2.3.1. C2 Compiler

C2C is a compiler for a static language. The total amount of memory that a program will use can be determined at compile time. Its main purpose is to help learning the semantics of languages with pure static memory management. Parameter passing can be by reference or value.

Learning Features

- Static memory management.
- Why recursion cannot be implemented in this model.
- Routines call.
- Overhead at run-time.
- Waste memory space.

- Parameter passing by value and reference.
- Scope vs lifetime.

2.3.2. C3 Compiler

C3C is a compiler for a stack based language. It supports direct and indirect recursion. Its main purpose is to help learning the semantics of stack-based languages and recursion. Parameter passing can be by reference, value, result and value-result.

Learning Features

- Implementation of stack based languages.
- Current and free registers (bp and sp registers).
- Dynamic chain, use and implementation.
- Recursion: concepts and implementation.
- Static and semi-static variables.
- Overhead at run-time.
- Code generated for a subprogram call.
- Parameter semantics: by reference, by value, by result, by value-result (concepts and implementation).
- Activation records.
- Scope vs lifetime.

2.3.3. C4 Compiler

C4C is a compiler for a block structure language. Its supports the ability to nest subprograms. Its main purpose is to help learning the semantics of block-structured languages. Parameter passing can be by reference, value, result and value-result.

Learning Features

- Dynamic chain, use and implementation.
- Static chain, use and implementation
- Differences between static and dynamic chains
- Local, non-local and global references.
- Activation records
- Static scope rules
- Scope vs lifetime
- Local routines vs global routines

2.3.4. C5 Compiler

C5C is a compiler that adds more dynamic behaviors to C4. Its main purpose is to help learning heap allocations. Parameter passing can be by reference, value, result and value-result.

This language supports additional features, such as:

- User-defined data types
  - Records
  - Static arrays

  - Dynamic arrays
- Pointer variables and pointer management with memory allocation operators
  - *new*
  - *valueat*( )
  - *addressof*( )

Learning Features

- Activation records whose size is known at execution time: dynamic arrays.
- Static, semi-static, dynamic and semi-dynamic variables.
- Heap management.
- Static vs dynamic arrays: behavior and implementation.
- Heap vs Stack.
- Insecurities of pointers: garbage and dangling references.
- Semantics and implementation of Records.
- Passing parameters by reference vs pointer copying (C-like).

2.3.5. Common Learning Features

- Compilation vs. Interpretation
- Differences in run-time structure.
- Code generated for each syntactical construction.
- Errors detected at compile and execution time.
- Performance comparison between two programs which do the same task written in different languages, by observing the way they manage system memory.
- For compiler design:
  - As mentioned, new compilers can be added to LaST. Instructors of courses such as Compiler Design, can then encourage their students to develop compilers that can be added to LaST. This expands LaST as a learning tool, as it can be used as a method to test code generated by the student's compilers, perform step-by-step executions, and graphically study the behavior of running program.
  - For testing the semantics of syntactical constructions in the research of new features in programming languages.

3. LaST Benefits

3.1. For Students

LaST helps students understand programming language concepts better, especially those related to the semantics of syntactical constructions. Graphical execution of sample programs with their corresponding translation to Simplesem code allows the students to see the real connection between a high-level language and the underlying machine, translating abstract concepts into concrete ones.

It encourages students to engage actively in their own learning, recreating the execution of a program discussed during a class, examining the execution of different programs during a lab class, doing exercises, or reviewing topics explained to them during class. It also encourages the students to write better, more optimized code.

3.2. For Instructors

LaST can be used to teach a considerable amount of concepts related to the study and implementation of programming languages. Its interface features facilitates the exposition and complex concepts can be easily illustrated, improving the actual teaching methods. As an example, suppose the instructor presents the students a program example, to explain the semantics of certain syntactical construction. Beginning the explanation the instructor conducts a ”blackboard execution” of the program, showing the changes and results which happen inside the computer with each instruction. To illustrate this changes the instructor draws a number of graphical representations of the computer resources which must be modified along with the program's execution. A popular option nowadays is to use slides representing the state of the memory and registers after each instruction is executed. This task goes on while the theoretical concepts are introduced.

Several problems arise if this method is used:

- The ”blackboard execution” usually turns out to be complex to keep up with, for both the instructor and the students.
- It is a highly error-prone method.
- The instructor may be forced to redo all slides to accommodate a new or improved example.
- The students who must acquire these new concepts must follow the execution of the program, observe and understand the changes produced in the computer resources, and at the same time understand the new concepts under study. This is a complex process, specially for students who have just been introduced to this field of study.

LaST addresses these problems because instructor and students must only focus on exposition and comprehension of the concepts being discussed.

4. Comparing LaST Against Other Proposals

SDE (Simplesem Development Environment) [7], consists of a Java-based environment that allows students to code, debug and execute Simplesem programs. It executes Simplesem programs while displaying the memory changes made by their instructions. LaST and SDE both support step-by-step executions and breakpoints. SDE does not support high level programming languages while using LaST students can write programs in four different high level programming languages or in Simplesem. This allows students to begin writing programs in a high level programming language, and learning how high-level statements are translated into Simplesem instruction even before they start writing in a low-level, assembly-like language. Moreover, in LaST, step-by-step execution and breakpoints are supported not only in Simplesem instruction but in high-level statements as well.

SDE memory graphical feedback provides a single view of an array of memory cells that displays integer values, whereas LaST memory supports automatic coloring memory zones distinguishing among registers, activations records and heap allocations, drawing pointers, displays high-level variable names along with their integer value. All of these is included in a memory panel that can be divided in two views of the same data.

ICOMP (Illustrated COMPiler) [1] is an illustrated compiler for a block structured language. Its main purpose is to show the internal structure and behavior of a compiler to support compiler construction courses; LaST on the other hand can be applied mainly to programming language concepts and related courses. It could also be used as a complementary tool together with ICOMP in compiler courses, as a platform in which students can test their compilers.

Although ICOMP can display the run-time stack while executing a sample program, it does not distinguish between different memory zones nor lets the student interact with the program being executed. Moreover, sample programs in ICOMP can only be written in one single language (PL/0) allowing only one implementation model to be analyzed by the student, whereas LaST provides four different languages (with four different implementation models) and the ability to extend this number by creating new compilers.

5. Example

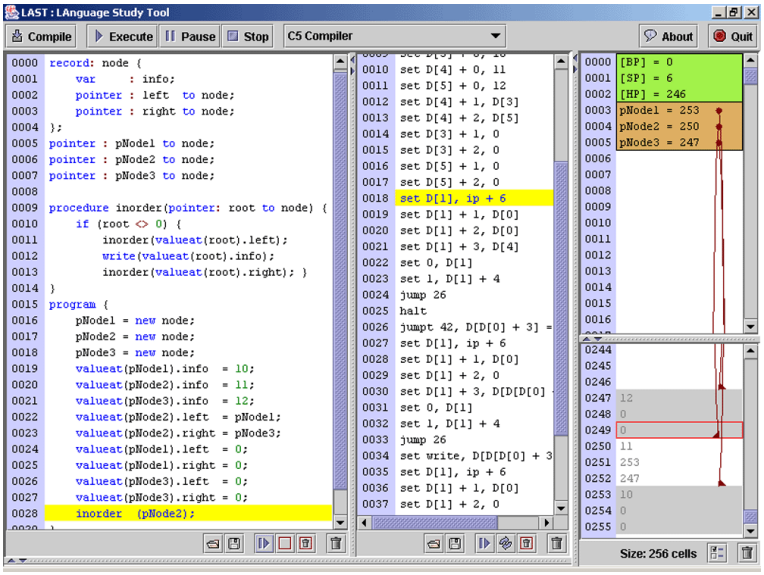

Figure 1: Just after the creation of the tree

This section presents a sample program to show how recursion and heap allocations can be appreciated with LaST. To illustrate these features, the sample program will be executed in step-by-step mode in order to observe the actions produced by each syntactical construction.

The program used constructs a binary tree, which consists of three nodes, and then prints out the in-order traversal of that tree. The execution starts at line 16, which is the first instruction of the main program (see Figure 1). The Source Code Editor, located on the left, contains the sample program. It was compiled using the C5 Compiler, and the result of the compilation process are the Simplesem instructions on the Simplesem Code Panel (middle panel). Now, the execution is paused at line 28 just before the call to the procedure *inorder*() and after the binary tree has been built. The Memory Panel on the right is split in two views. The top view shows the Base Pointer (cell 0), the Stack Pointer (cell 1) and the Heap Pointer (cell 2). Next is the static data with the global variables *pNode1*, *pNode2*, *pNode3* (cells 3, 4 and 5). On a color screen the base pointer, the stack pointer and the heap pointer are highlighted with green background and the static data with brown background.

The bottom view of the Memory Panel shows the three nodes of the tree. The heap starts with the node with the value 10, then from cell 252 to 250 the node with value 11 and finally from cell 249 to 247 the node with the value 12. The cell 249 is highlighted with a square because was the last cell modified. The arrows represents the pointers from the variables in the static area to the nodes in the heap. To simplify the heap view, arrows between nodes are not displayed.

When the procedure *inorder*() is called, its activation record is created and pushed into the stack. Figure 2 shows the LaST user interface just after *inorder*() is invoked. The activation record

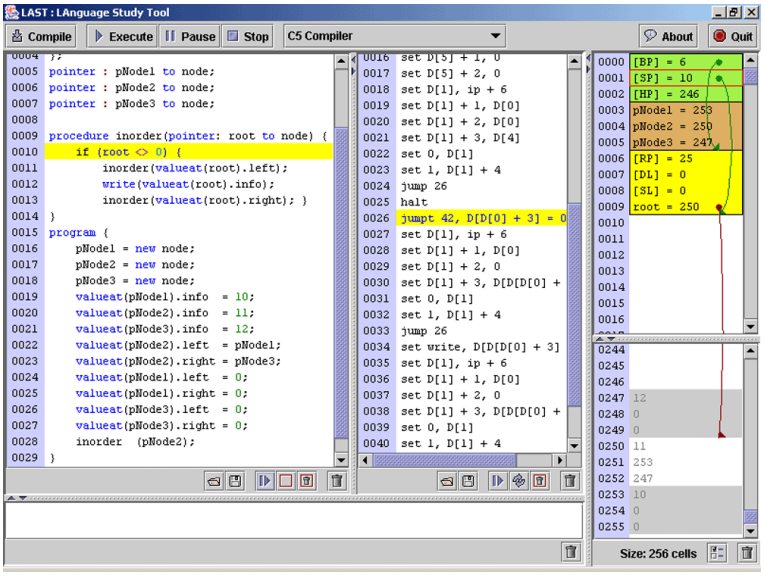

Figure 2: The first call to the *inorder*() procedure

of *inorder*() contains the Return Pointer (cell 6), Dynamic Link (cell 7), Static Link (cell 8) and the parameter (passed by value) *root* (cell 9). The parameter *root* points to the node with the value 11 (see the arrow in Figure 2). The arrows of the Base Pointer and Stack Pointer are also displayed. Again, in a color screen the current activation record is highlighted in yellow background.

Note that the next Simplesem instruction to be executed is on line 26 (middle panel), the conditional jump, which correspond to the *if* statement at line 10 in the high level code (left panel). This is the first sentence of the *inorder*() subprogram and next to be to execute.

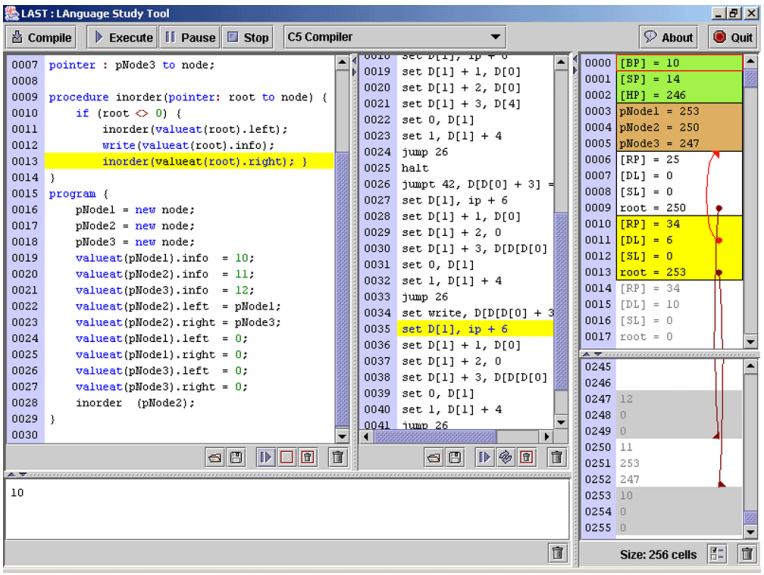

Figure 3: After two recursive calls

Figure 3 shows the LaST user interface after two recursive calls to *inorder*().The first recursive call creates a second activation record (cells 10 to 13) and the variable *root*, allocated there, points to the node with value 10. Then, the activation record of the second recursive call is created (cells 14 to 17) with the variable *root* pointing to 0 (*null* value). After that, the conditional statement in line 10

is evaluated again. This time the comparison fails, and the second recursive call to *inorder*() finishes, freeing the memory allocated for its activation record (cells 14 to 17). The execution continues with the *write* statement (line 12) which prints the value 10 to the Output Panel. Figure 3 also display the arrow from cell 11 to cell 6 which represent the dynamic link.

Figure 4 shows the LaST user interface after the program has finished executing. The value of the nodes has been printed in the Output Panel, and all activation records have been released from the stack.

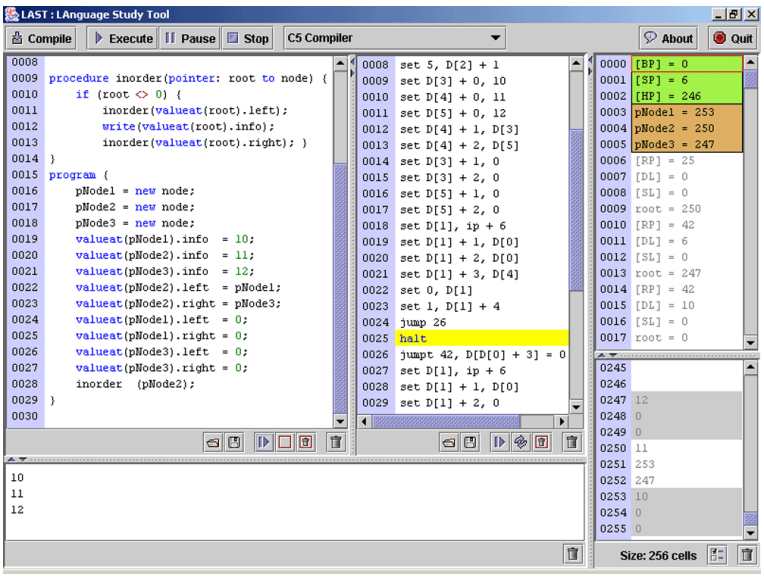

Figure 4: The execution finished

6. Conclusions and Future Work

LaST is a powerful tool which helps improve the learning process using it to assist teaching and to encourage students to engage actively in their own learning.

LaST can be a great auxiliary for instructors in programming languages and related courses. It's easy to use and comprehend, and extensible. This important feature allows LaST to be used to show the semantics of any syntactical construction, such as those related with object oriented languages, providing the necessary compiler has been developed for LaST.

Work with LaST as a learning resource in programming language courses will continue. It will be extended incorporating different programming languages (along with their respective compilers) to enrich it in order to help teaching new syntactical constructions semantics.

References

[1] K. Andrews, R. Henry, and W. Yamamoto. Design and implementation of the uw illustrated compiler. In *Sigplan '88 Conf. Programming Language Design and Implementation*, 1988.

[2] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 1997.

[3] O. S. Initiative. Open source initiative osi. http://www.opensource.org, 2003.

[4] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[5] S. Microsystems. Java technology. http://java.sun.com, 2003.

[6] E. Molinari and E. Lindner. Last: Language study tool. http://sitios.ub.edu.ar/last, 2003.
[7] A. Winzer. A simplesem interpreter in java. Master's thesis, Technical University of Vienna, 1997.