Animating Parsing Algorithms

Sami Khuri
(408)924-5081
khuri@mathcs.sjsu.edu

Yanti Sugono
(408)922-4069
sugono@pacbell.net

Department of Mathematics and Computer Science
San José State University
One Washington Square
San José, CA 95192-0103

Abstract

The paper describes a package that can be used to present the parsing algorithms. The package fully animates the top-down LL(1) and bottom-up SLR(1) parsing algorithms. By full animation we mean that the input string being parsed, the corresponding actions that take place in the stack, and the building of the parse tree are all simultaneously animated on the same screen, thus enabling the user to get a full appreciation of all the intricate details that occur during parsing. The package makes use of XTANGO and can be used in the beginning of the semester as a teaching tool. Later, the students could be asked to write their own animations of the compiling process.

1 Introduction

In this work, we show how to build and use a package that animates compiler algorithms. Through its use, students can get a full appreciation of the details associated with the compiling process. Later in the semester, they can be asked to develop their own package. While it is our belief that demonstrating an animation of the compiling algorithms will enhance the understanding of the compiling process, we are convinced that "active engagement on the part of students leads to higher motivation and better integration and retention of content" [2]. What better student engagement than to have them create their own animations?

In the next section, we briefly review the different parts of the compiler, focusing on the parser phase which is generally implemented by either top-down or bottom-up parsing algorithms. The labels "top-down" and "bottom-up" refer to the way the parsing tree is built. The third section treats animation tools in general, categorizes our package, and briefly compares our package to others whose descriptions have been reported in the literature. Section four starts with a brief introduction to XTANGO, and then explains our own package, including two examples and screen dumps to illustrate the power and efficiency of our package. We conclude the paper with some closing remarks and possible extensions of this work.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
SIGSCE 98 Atlanta GA USA
Copyright 1998 0-89791-994-7/98/ 2..$5.00

2 Compiling Algorithms

In this section, we make use of Louden's textbook [11] to give a brief description of the compiling process.

A compiler consists internally of a number of steps, or phases, that perform distinct logical operations. The phases of a compiler are: scanner, parser, semantic analyzer, source code optimizer, code generator, and target code optimizer. In this work, we are interested in the second phase: the parser. The parser receives the source code in the form of tokens from the scanner and performs syntax analysis, which determines the structure of the program. The parsing methods fall into two classes, top-down and bottom-up. A top-down parsing algorithm parses an input string of tokens by tracing out the steps in a leftmost derivation. Such an algorithm is called top-down because the implied traversal of the parse tree is a preorder traversal and, thus, occurs from the root to the leaves. The LL(1) parsing algorithm is one of the most popular top-down algorithms. Recall that in LL(1), the first L is for left-to-right scan of tokens, the second L is for leftmost derivation, and numeral "one" represents the fact that only one input token is scanned at a time. LL(1) parsing uses an explicit stack to perform a parse, rather than recursive calls as is the case with other top-down parsers such as the recursive-descent parser.

Bottom-up parsing algorithms are in general more powerful than top-down methods. Consequently, the constructions involved in these algorithms are more complex. The success of bottom-up LR parsers (also known as reduce-shift parsers) can be measured by the number of parser generators, such as YACC [7], that are commercially available. A bottom-up parser uses an explicit stack to perform a parse, similar to a top-down parser. The parsing stack will contain both tokens (obtained form the previous phase: the scanner) and nonterminals, and also some extra state information. The stack is empty at the beginning of a bottom-up parse and will contain the start symbol at the end of a successful parse. SLR(1) parser is one of the bottom-up parsers, where S is for simple, L is for left-to-right of tokens, R for rightmost derivation, and one, once again, means that the parser consider only one input token at a time. Both techniques are essentially look-up algorithms, where one has to build a parse table first. Then, the action to be performed always depends on the current input symbol and the character on top of the stack.

Traditional methods for explaining these concepts have included the use of static pictures, or the overlaying of transparencies on the overhead projector, or even overlaying of diagrams in textbooks [10].

Before explaining our package, we describe different animation tools and give a brief review of existing work in the field. We also explain how our package differs from others that tackle the same problem.

3 Tools for Animation

Boroni et al. [5] divide the animation tools for studying computer science into three general categories: program animators, algorithm animators and concept animators. A program animator is a sophisticated, easy to use, interactive debugger. It is sophisticated since it includes unique features, information that enhances the user's understanding and appreciation of the program, such as time and space complexity of the program. An algorithm animator gives pictorial visualizations of an algorithm. The user sees an animation of the whole algorithm in execution. "Algorithm animation is concerned with visualizing the internal operations of a running program in such a way that the user gains some understanding of the workings of the algorithm" [12]. Examples include searching and sorting algorithms, where the animation shows objects being compared and moved around according to the specific algorithm. Concept animation, which consists in animating a specific concept, generally abstract, can be found in various fields of computer science. In automata theory, for instance, the acceptance and rejection of input strings to automata have been animated [1, 3].

Our package, Parser, falls under the second category, since it animates the whole compiling process. It animates a collection of concepts. In many other papers found in the literature that tackle compiling, the animating tools describe specific parts of the compiler. For example, LLparse and LRparse that first appeared in [4], and were later enhanced and extended to include FLAP [3], use animation to explain various concepts pertinent to the compiling process. They all fall under the third category: concept animator. In "Understanding the Bottom-Up SLR Parser" [8] too, the authors describe a package that gives a visual representation of how each phase of a compiler performs individually and how it interacts with the other phases, and falls under the third category. The mentioned tools are excellent and have been successfully used in teaching. It is our belief that the compiling technique, and more precisely, the building of the parse tree, whether top-down or bottom-up, would be easily understood if the process could be animated, where the students could see, on the same monitor, the string being parsed, the corresponding actions that take place in the stack, and the actual parse tree being built. The user thus sees the whole picture and not the animation of separate parts, as is the case with the above mentioned tools.

In the next section, we explain our package and demonstrate its power and efficiency by tackling two examples.

4 The Animation

4.1 XTANGO

For the animation, we make use of Stasko's XTANGO (X-window Transition-based Animation Generation) package [13]. It executes on UNIX workstations with bit-mapped color displays running the X11 windowing system and provides a platform for constructing algorithm animations. It is a well-known algorithm animation package that has been widely used. Hartley for instance, uses it to animate classical problems encountered in the study of interprocess communication in operating systems, such as the dining philosophers problem [6]. As a matter of fact, the XTANGO package has many sample animation programs, including fast Fourier transforms, searching and sorting, and combinatorial optimization problems, such as binpacking. To build an algorithm animation, one needs to create two files. The first one is for the program being animated, and the second one for the XTANGO animation control code. The first one, the program file, contains the program that the student generally implements in a compiler course. The XTANGO user simply has to learn a few control commands and place calls to functions at appropriate positions within the program that will be used to drive the animation. The animation control code file contains routines that describe how the animation is to appear. It is relatively easy to comprehend how both files bind to produce the animation. The package is user-friendly; after all, the "focus of the system is on ease-of-use" [13].

In the next section, we explain how our package Parser makes use of XTANGO to create the animation needed to visualize the whole parsing process.

4.2 Parser

Upon invoking Parser, an XTANGO window appears. At the console, a user is prompted to choose from the main menu. The three choices are the LL(1) parser, the SLR(1) parser, and exit the program. When the user chooses either one of the first two choices, she/he will be given four more choices: the first one is to load a parsing table file, the second one is to run the current grammar (in case we want to use the same grammar a second time), the third one is to demonstrate the specified parser, and the fourth is to go back to the main menu.

The parsing table is used as input data to the program. We provide seven parsing table files for the LL(1) parser and twelve parsing table files for the SLR(1) parser. The user can of course also write her/his own parsing table file(s). The specifications for writing a parsing table file for either the LL(1) or the SLR(1) parser are simple and are explained in the README file. Once a parsing table for a specific grammar is loaded, the user can input a string and determine whether the input string is accepted by the grammar or not. For further information about how the program works, the user can always choose the "demo" option in the menu for both parsers.

The animation basically manipulates four objects: a parsing stack, an input string buffer, an action box, and a parsing tree. As the program performs the parsing of an input string according to the current grammar, the animation mirrors these processes. An arrow pointer always points to the top of the parsing stack, while the other arrow points to the current character in the input string. The action box always displays each action taken by the specific parser, and a parsing tree grows as a parsing process is being performed. The parsing process can also be followed, at the same time, on the user console window. Thus, the user can have both windows opened and simultaneously see the animation in the XTANGO window and the derivations on the user console. In the next two examples, we give snapshots of both windows to illustrate the animation of the parsing process in action.

Example 1
Consider the following LL(1) grammar:
$A \rightarrow (A)A|Z$, where Z denotes the empty string.
The string: ()((())) is parsed by the LL(1) compiler for that grammar. A snapshot of the compiling process in execution can be seen in the XTANGO window shown in Figure 1. The compiling process started with

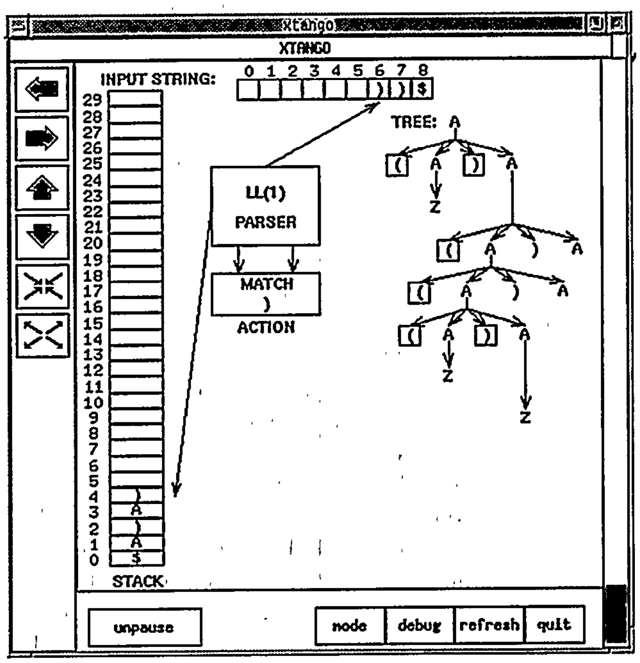
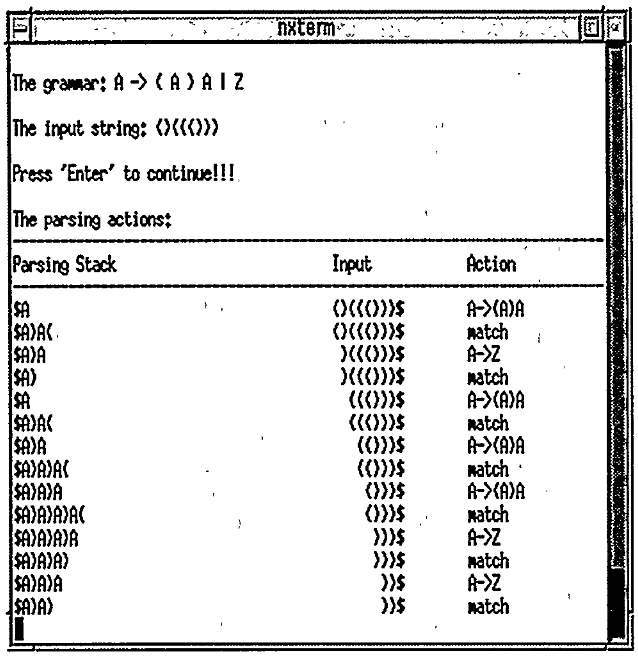


Figure 1: Snapshot of parsing process for Example 1.

the start symbol "A" at the bottom of the stack ($ is the end of stack marker), with the string ()((())) in positions 0 to 7 ($ in the 8th position is the end of string marker) in the "input string" section of the window, and with an empty tree. Figure 1 gives a snapshot of the parsing process after the first six characters of the input string, i.e., ()((((), have been processed. Consequently, only)) is shown in the "input string" part in Figure 1. In the tree, the first six characters have already been processed, and labeled as leaves, as the squares around them indicate. The parser is about to recognize the sixth character,")", and that is why we have "Match" in the action box. There is a match between the current input symbol and the top of stack character. Incidently, the other possible actions of the parser are generate, reject, and accept. In the next step (not shown here), the top of stack character is popped, the 6th character will vanish from the input string, ")" in the level before the last in the tree will be put in a square, and the two pointers (to the stack and to the input string) will move accordingly. Figure 2 shows all the derivations that occurred from the beginning of the compilation process to exactly the same point as the snapshot of the compiling process depicted in Figure 1.

In the next example, we show snapshots of the XTANGO and user windows of the parsing process when a string is processed by a bottom-up compiler.

Example 2
Consider the following SLR(1) grammar:



```
The grammar: A -> ( A ) A | Z

The input string: ()((()))

Press 'Enter' to continue!!!

The parsing actions:
--------------------------------------------------------------
Parsing Stack                     Input         Action
--------------------------------------------------------------
$A                                ()((()))$     A->(A)A
$A)A(                             ()((()))$     match
$A)A                               )((()))$     A->Z
$A)                                )((()))$     match
$A                                  ((()))$     A->(A)A
$A)A(                               ((()))$     match
$A)A                                 (()))$     A->(A)A
$A)A)A(                              (()))$     match
$A)A)A                                ()))$     A->(A)A
$A)A)A)A(                             ()))$     match
$A)A)A)A                               )))$     A->Z
$A)A)A)                                )))$     match
$A)A)A                                  ))$     A->Z
$A)A)                                   ))$     match
```

Figure 2: Derivation of string of Example 1.

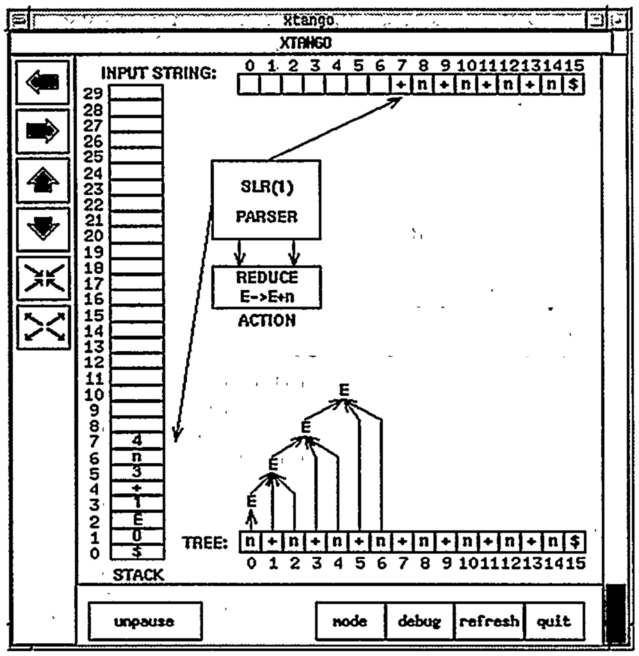
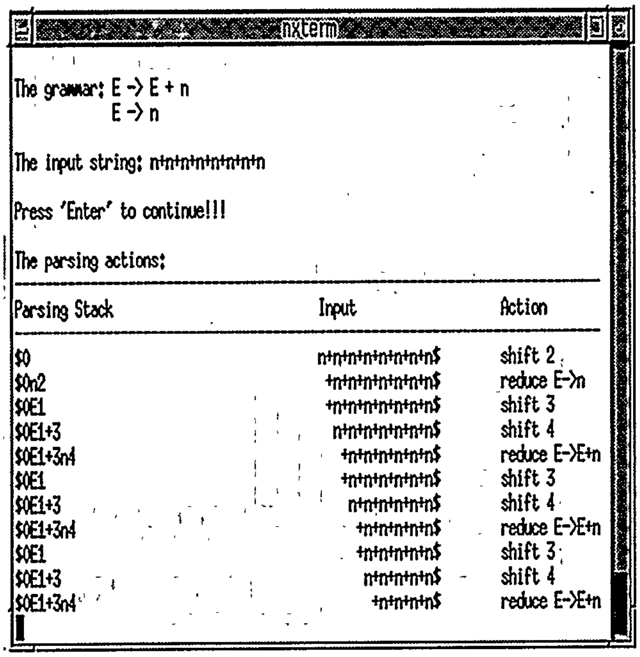


Figure 3: Snapshot of parsing process for Example 2

$E \rightarrow E + n|n$ The string: n+n+n+n+n+n+n+n is parsed by the SLR(1) compiler for that grammar. A snapshot of the parsing process in execution can be seen in the XTANGO window shown in Figure 3. The compiling process started with an empty stack, with the string n+n+n+n+n+n+n+n in positions 0 to 14 in the "input string" section of the window, and with an empty tree. Figure 3 gives a snapshot of the compiling process after the first seven characters of the input string have been processed. In the tree, the first seven characters have already been processed. The compiler is about to build one more layer in the tree with 3 children: E (the root of the tree in Figure 3), "+", and "n" in positions 7 and 8, respectively. In other words, the new tree will have one more layer with E as root node and three children: E (the current root of the tree in Figure 3) as lefmost child, "+" in position 7 as middle child and "n" in position 8 as rightmost child. This is also indicated by the action part in the XTANGO window. To reduce by $E \rightarrow E + n$ means to replace the right hand side of the production rule, i.e., E+n, by the left hand side, i.e., E. Incidently, the other possible actions for the SLR(1) parser are shift, reject, and accept. In the next step, the "handle" (E+n) is popped from the stack (along with the state numbers 4, 3, and 1 - see the stack in Figure 3), and the tree is augmented by moving one layer up, as explained above. The pointer to the stack will move accordingly. Figure 4 shows all the derivations that occurred from the beginning of the compilation process to exactly the same point as the snapshot of the compiling process shown in Figure 3.

As mentioned in the introduction, we believe that the package should be used to introduce the two parsing techniques to the students. They can also use it to test their own understanding of the parsing process by creating their own parse table and feeding it to Parser along with an input string to see if their work is correct. Eventually, the students should be able to write their own animation of the parsing process.

In most universities, a programming project is an integral part of a compiler course. Students are generally asked to implement a top-down (such as LL(1)) or bottom-up (such as SLR(1)) parser for a given grammar. Sometimes this implementation is part of a bigger project, where they also have to implement the preceding phase: the lexical analyzer which feeds the tokens to the parser, and sometimes the project solely consists of the parser. But moving from this kind of project to animating the algorithms in the project does not require an overwhelming effort. XTANGO lends itself very well to the task. The student will have to write the code, as is done in any "traditional" compiling course, and then add a few control commands. Then, the student will have to create a second file, that responds to the first one's function calls, to actually build the animation. All this work can be done in an open lab.



```
nxterm

The grammar: E -> E + n
             E -> n

The input string: n+n+n+n+n+n+n+n

Press 'Enter' to continue!!!

The parsing actions:
--------------------------------------------------------------
Parsing Stack                 Input              Action
--------------------------------------------------------------
$0                      n+n+n+n+n+n+n+n$         shift 2
$0n2                     +n+n+n+n+n+n+n$         reduce E->n
$0E1                     +n+n+n+n+n+n+n$         shift 3
$0E1+3                    n+n+n+n+n+n+n$         shift 4
$0E1+3n4                   +n+n+n+n+n+n$         reduce E->E+n
$0E1                       +n+n+n+n+n+n$         shift 3
$0E1+3                      n+n+n+n+n+n$         shift 4
$0E1+3n4                     +n+n+n+n+n$         reduce E->E+n
$0E1                         +n+n+n+n+n$         shift 3
$0E1+3                        n+n+n+n+n$         shift 4
$0E1+3n4                       +n+n+n+n$         reduce E->E+n
```

Figure 4: Derivation of string of Example 2.

5 Conclusion

In this work, we introduced a package that can be used in compiler or automata theory classes, to explain the parsing process. The package contains more challenging grammars than the easy two examples we have introduced in this paper. Although the use of algorithm animation is not new in computer science, we believe that our package is original since it allows the user to view all the steps that are involved in a parsing process. The user can see the string being scanned, items being pushed and popped from the parse stack, and the tree being built; all at the same time on the XTANGO window. On the console, the user can also follow the derivation of the process. The speed of the animation can be adjusted and there is a "pause" button that allows the user to stop and reflect before continuing with the process. Eventually, the students should write their own animations. We definitely agree with Lawrence et al. [9] who concluded that students who actively participate in the process, who write their own animations, achieve a better understanding of the concepts behind the algorithms being studied. Our package Parser, which will be available over the net, is flexible enough to allow the animation of other compiling algorithms, such as LR(1) and LALR.

Acknowledgments

The authors would like to thank Po-Han Hsueh for helping out with the programs, and one of the anonymous reviewers for suggestions that improved this work.

References

[1] Barwise, J. and Etchemendy, J. Turing's World: A Computer-Based Introduction to Computability Theory. *Kinko's Academic Courseware Exchange* (Santa Barbara, CA 1986).

[2] Bergin, J., Brodlie, K., Goldweber, M., Jiménez-Peris, R., Khuri S., Patiño-Martínez, M., McNally, M., Naps, T., Rodger S., and Wilson, J. An Overview of Visualization: its Use and Design. *Integrating Technology into Computer Science Education, SIGCSE* (28:192-200, March 1996).

[3] Bilska, A., Leider, K., Procopiuc, M., Procopiuc, O., Rodger, S., Salemme, and Tsang, E. A Collection of Tools for Making Automata Theory and Formal Languages Come Alive *Proc of the SIGCSE Technical Symposium* 28 (1:15-19, March 1997).

[4] Blythe, S., James, M., and Rodger, S. LLparse and LRparse: Visual and Interactive Tools for Parsing. *Proc of the SIGCSE Technical Symposium* 25 (1:208-212, March 1994).

[5] Boroni, C., Goosey, F., Grinder, M., Rockford, R., and Wissenbach, P. WebLab! A Universal and Interactive Teaching, Learning, and Laboratory Environment for the World Wide Web. *Proc of the SIGCSE Technical Symposium* 28 (1:199-203, March 1997).

[6] Hartley, S. Animating Operating Systems Algorithms with XTANGO. *Proc of the SIGCSE Technical Symposium* 25 (1:344-348, March 1994).

[7] Johnson, S. YACC-yet another compiler-compiler. *CS Technical Report* (23, Bell Telephone Laboratories, Murray Hill, NJ, 1975).

[8] Khuri, S. and Williams, J. Understanding the Bottom-Up SLR Parser. *Proc of the SIGCSE Technical Symposium* 25 (1:339-343, March 1994).

[9] Lawrence, A., Badre, A., and Stasko, J. Empirically Evaluating the Use of Animations to Teach Algorithms *Technical Report GIT-GVU-94-07* (Georgia Institute of Technology, Atlanta, GA, 1994).

[10] Lewis, H. and Papadimitriou, C. Elements of the Theory of Computation. (Prentice Hall, NY, 1981).

[11] Louden, K. Compiler Construction: Principles and Practice. (PWS, 1997).

[12] Najork, M. and Brown, M. A library for Visualizing Combinatorial Structures. *SRC Research Report* (Digital Systems Research Center, Palo Alto, CA, September, 1994).

[13] Stasko, J. TANGO: A Framework and System for Algorithm Animation. *IEEE Computer* (23:27-39, September 1990).