Software Visualization of LR Parsing and Synthesized Attribute Evaluation

Elizabeth L. White[*]
Department of Computer Science
George Mason University

Laura Denise Deddens[†]
Department of Computer Science
University of California, Santa Cruz

Jeffrey Ruby
Department of Computer Science
George Mason University

Abstract

Visual YACC is a tool that automatically creates visualizations of the YACC LR parsing process and synthesized attribute computation. The Visual YACC tool works by instrumenting a standard YACC grammar with graphics calls that draw the appropriate data structures given the current actions by the parser. The new grammar is processed by the YACC tools and the resulting parser displays the parse stack and parse tree for every step of the parsing process of a given input string.

Visual YACC was initially designed to be used in compiler construction courses to supplement the teaching of parsing and syntax directed evaluation. We have also found it to be useful in the difficult task of debugging YACC grammars. In this paper we describe this tool and how it is used in both contexts. We also detail two different implementations of this tool, one that produces a parser written in C with calls to Motif and a second implementation that generates Java source code.

Keywords: Visualization, LR Parsing, Syntax Directed Translation

[*]Supported by the National Science Foundation under contract CCR-96-25-202
[†]Supported by the CRA–NSF Distributed Mentorship Program.

Introduction

One of the challenging aspects of teaching a course in compiler construction theory is teaching students about the details of different parsing techniques and syntax directed translation. Both top–down (LL) and bottom–up (LR) parsing have their complexities, but LR parsing is the more challenging of the two because the choices made by the parser are non-intuitive.

An example LR parse is shown in Figure 1, where both the forest of parse trees (left) and the LR parse stack (right) is shown after each of five grammar reductions for the infix expression `3 + 4 * 2`. The parse tree for the expression is built bottom up using both the shift of new symbols and the merging or reduction of some set of trees into a single tree. For each stage of the process, the nodes at the roots of each of the trees are the elements on the LR parse stack. The task of an LR parser is to determine when it is appropriate to reduce by one of the given productions and when it is appropriate to shift a new terminal symbol (token). Different actions may be taken at different times for similar stacks. For example, `expr + expr` is shown on the stack for phases two and four; however, different actions are taken for these two phases.

Syntax directed translation is another difficult concept for some students to grasp because it is recursive in nature and it is closely tied to the parsing process. Consider the standard textbook problem of evaluating infix expressions during an LR parse. A straightforward way of solving this problem is to associate an attribute with each internal node of the parse tree and to compute the value of these attributes using the attribute values in the child nodes. Attributes that are computed using only information from direct descendants in the parse tree are called *synthesized* attributes. Computation of synthesized attributes is straightforward during an LR parse, as seen in Figure 1. Each of the `expr` nodes in the tree has an attribute (shown in a box) that holds the value computed when the subtrees are merged into a single tree. An `expr` node with a single `NUM` child copies the value associated with the `NUM` token. An `expr` node with `expr` children and an operator computes its values based on the attribute values of its children. For example, during the fifth reduction, the `expr` node at the root gets an attribute value of 11 because its `expr` children have values of 3 and 8 respectively and its operator child is a '+'. The process of filling in the values of attributes is often called *annotating* the tree.

Understanding about both the LR parsing process and how synthesized attributes are evaluated during the parsing is critical in effective use of standard LR parsing tools such as YACC[1]. These concepts are generally taught during classtime by writing numerous YACC specifications and then drawing and annotating multiple parse trees similar to that of Figure 1. However, experience has shown us that this method is not successful for all students. Experimenting with different grammars and different annotation rules is helpful in this learning process; however, tools such as YACC do not provide students with much feedback in terms of what rules were chosen during a parse and hence, how the resulting attribute values were computed is non–obvious.

The initial goal of this work was to provide students a hands–on way to learn about these techniques. In particular, we wanted students to be able to write and annotate LR grammars and then see how their grammars operate on different inputs. Visualization was an obvious way to provide what we wanted. We decided early in the development process to work with Lex/YACC specifications because of the popular use of these tools in teaching environments. While knowledge of Lex and YACC is required to use the tool, we did not want the students to have to be aware of how the visualization was done. In other words, we wanted to create a tool that took any legal YACC specification and produced an executable that would show students how the specification would parse a given input. An advantage of this choice is that once visualization is no longer needed, the specifications can be input to YACC directly. We had two important constraints on the visualization tool we wanted to produce.

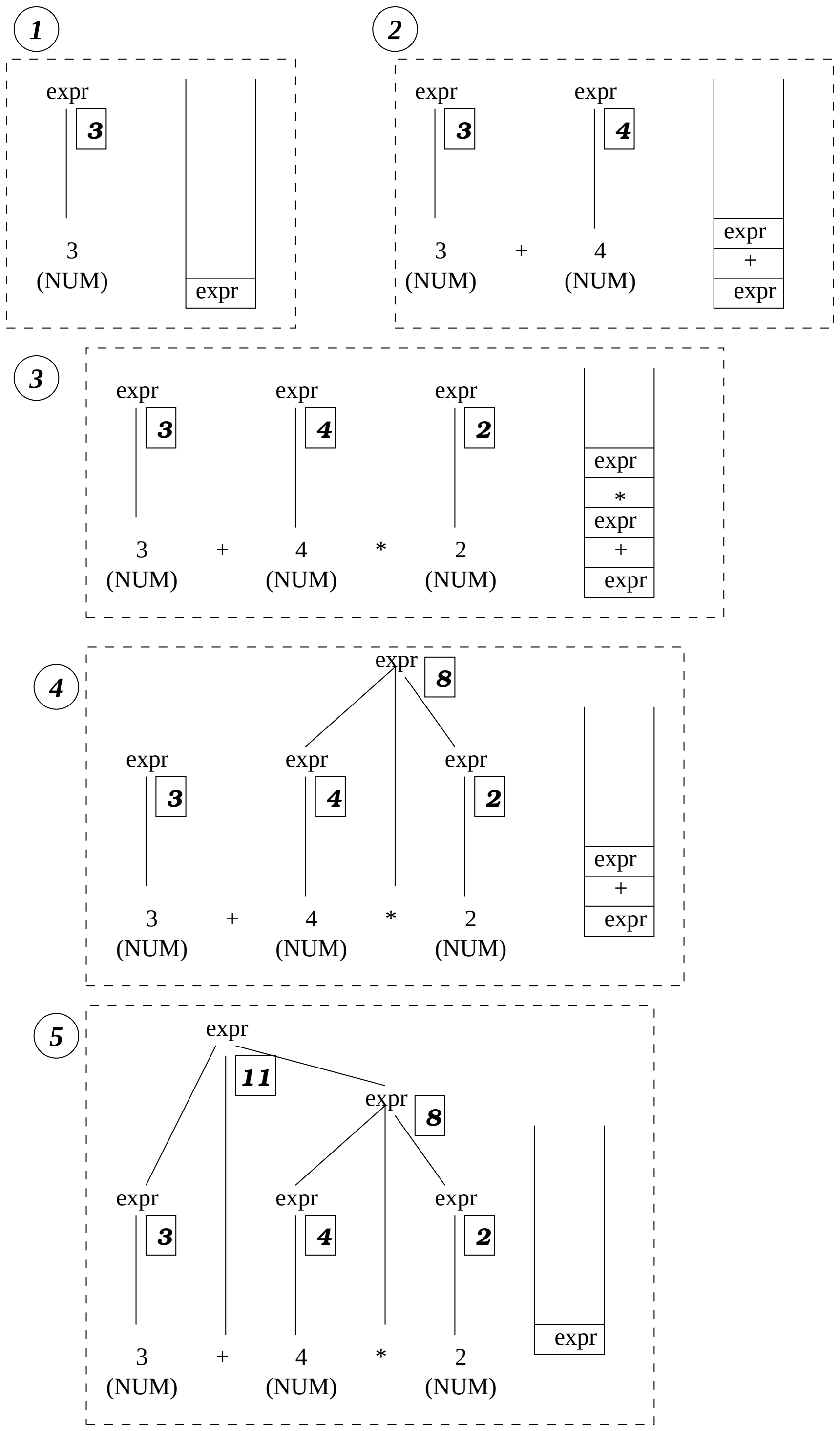

Figure 1: Attributed LR Parse of input `3 + 4 * 2` using grammar of Figure 2.

- The visualization must exhibit exactly the behavior described in the input grammar. It must recognize the same language and that it must choose the shifts and reductions in the same order as the original grammar for that language during the LR parsing process.
- The visualization must provide a way for the user to see attribute values associated with the grammar symbols. Because the emphasis of this work was on teaching rather than providing a complete visualization of parsing, we decided to limit visualized attributes to integers, although we do not restrict how many values are displayed.

In addition to teaching, a tool that meets the above constraints can also be used for debugging purposes. When a language requires a large number of productions, determining why a given string in the language generates a syntax error (or a string not in the language parses correctly) can be difficult. The parse tree generated during the visualization can assist in the process.

Visual YACC

The primary goal of this tool is to support student learning about LR parsing and syntax directed translation. This means that what we want to display to students is an abstraction of the LR parse stack and the attributes associated with its elements. The reason that we consider what is shown to be an abstraction is because we do not display all information used internally by the parsing process; we present a high–level view of the process. When the parser chooses to do a shift, we display a new element on top of the stack and when the parser reduces by a production, we show the removal of elements from the stack and the push of the non–terminal on the left side of the production.

A tool such as YACC does not actually build the parse tree during the parsing process; the primary internal data structure is the stack described above. However, a LR parse stack grows and shrinks continuously during the parsing process and only conveys a partial picture of the process. Because of this, we decided that visualizing the current parse tree abstraction was useful as well. A parse tree is a more intuitive data structure for most students and it conveys more complete information about the parsing done so far.

Using an existing Visual YACC parser is straightforward. The user executes the parser and provides a string to be parsed. A Visual YACC window (see Figure 3–8) has two sub–windows, one for displaying the LR stack and a second for displaying the parse tree. During a parse, the user can navigate within the two different views (stack and tree) to examine parts of the data structures. The main window has five buttons that allow the user to direct the parsing process. In addition to **start** and **quit** buttons, the user may choose to:

- **STEP** – Perform one step in the parsing. (i.e. shift one token or reduce by a single production)
- **MULTISTEP** – Perform some user–specified number of steps.
- **GO** – Complete the parsing without further interaction.

Consider the YACC grammar, shown in Figure 2, for parsing infix expressions of integer constants. This grammar could have been used to produce the parse tree in Figure 1 for the input `3 + 2 * 4`. The actions ($ notation) after each production cause the input expression to be evaluated during the parsing process. As described earlier, the evaluation is done by associating attributes with the nodes in the parse tree and evaluating these attributes using the attribute values at the child nodes. In YACC, the $$ stands for the

```
%union{
  int ival;
  char *name;
}

%token <name> NAME
%token <ival> NUMBER
%type <ival> expr

%left  '+'
%left '*'
%%
statement       : NAME '=' expr  { printf(``%s = %d\n'', $1, $3); }
        ;
expr      : expr '+' expr { $$ = $1 + $3; }
          | expr '*' expr { $$ = $1 * $3; }
          | '(' expr ')' { $$ = $2; }
          | NUMBER {$$ = $1;}
        ;
%%
```

Figure 2: YACC grammar for infix expressions composed of integers. The `%union` defines the attributes used by the specification and the `%type` and `%token` declarations associate attributes with particular nodes. The \$ notation defines the attribute evaluation. This grammar assumes that the tokens NUMBER and NAME are given values in the lexor.

attribute associated with the non–terminal on the left side of a production. The $\$n$ symbols stand for the attributes associated with the n^{th} symbols (terminals and/or non–terminals) on the right side of the production. A statement like \$\$ = \$1 + \$3 specifies the computation of the attribute for the non-terminal on the left side of the production as the sum of the attributes associated with the first and third symbols on the right side of the production.

Figures 3–8 show an abbreviated version of the Visual YACC visualization created for this annotated expression grammar and the input `x = 3 * 4`. On the left side of each window we see the current LR parse stack and attribute values associated with these symbols. For example, in Figure 4, the symbol `expression` in the parse stack has the value 3 because that is the value it got from its `NUMBER` child. The current forest of parse trees is shown on the right side of each window. In Figure 4, we have three different trees, two of which only contain a single element. During the parse, trees will be added and merged until we have the single parse tree for the input, as seen in Figure 8.

Using Visual YACC for Teaching

We have used Visual YACC on a voluntary basis in a teaching environment. The version used by the students includes about 10 examples, some of which are taken from a Lex/YACC text[3] and some of which were written by other students. The students are encouraged to start by looking at these examples and running them on several different input, both legal and illegal. Some of these examples are more oriented

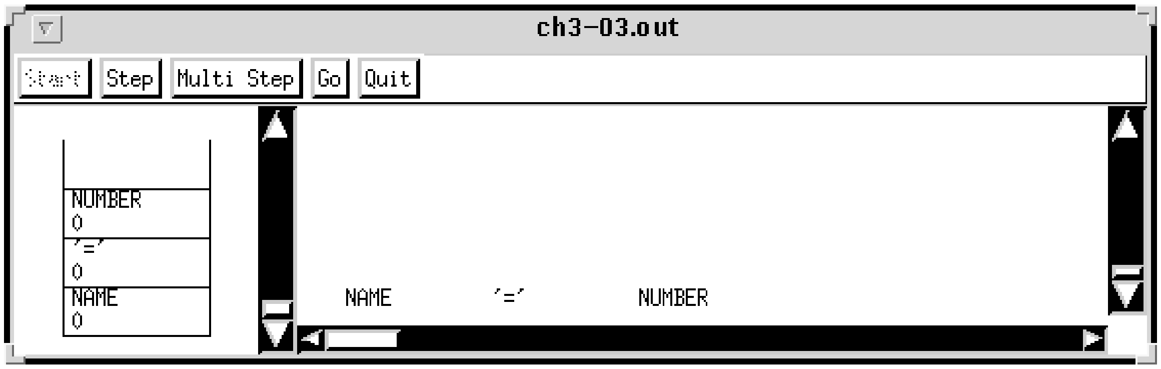

Figure 3: Visual YACC window during a parse. In this window, input `x := 3` has been processed.

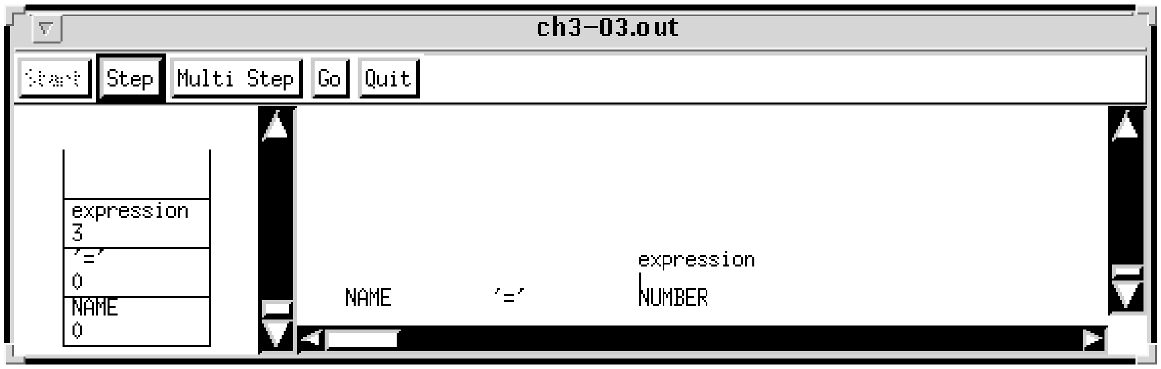

Figure 4: The production `expression` → `NUMBER` is reduced and the stack and tree are updated accordingly.

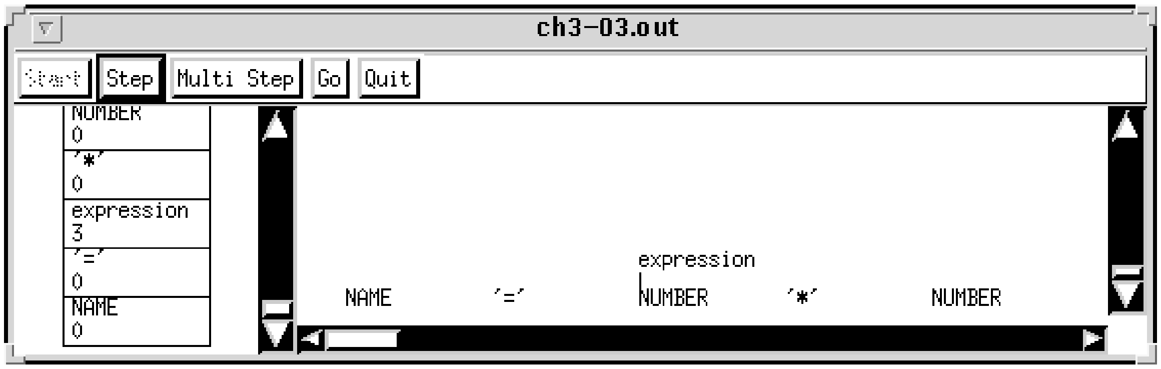

Figure 5: Two more input tokens ('*' and NUMBER) are shifted onto the LR parse stack.

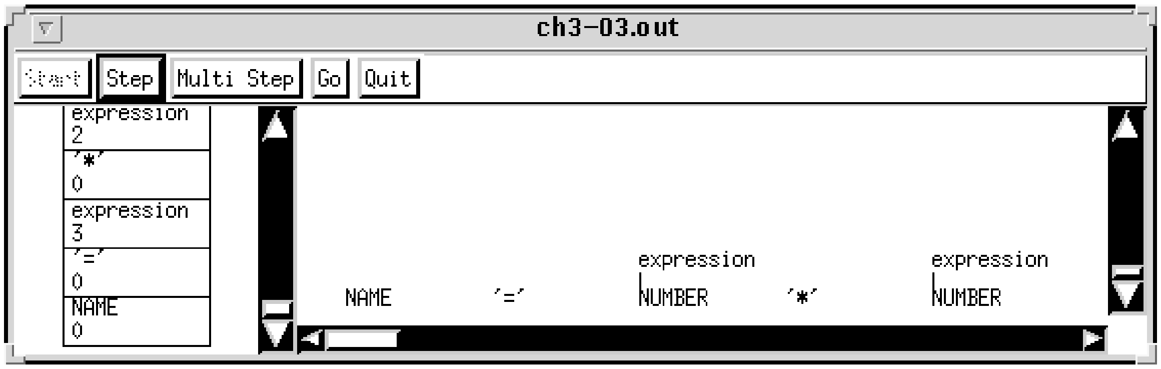

Figure 6: The production `expression` → `NUMBER` is again reduced and the stack and tree are updated accordingly.

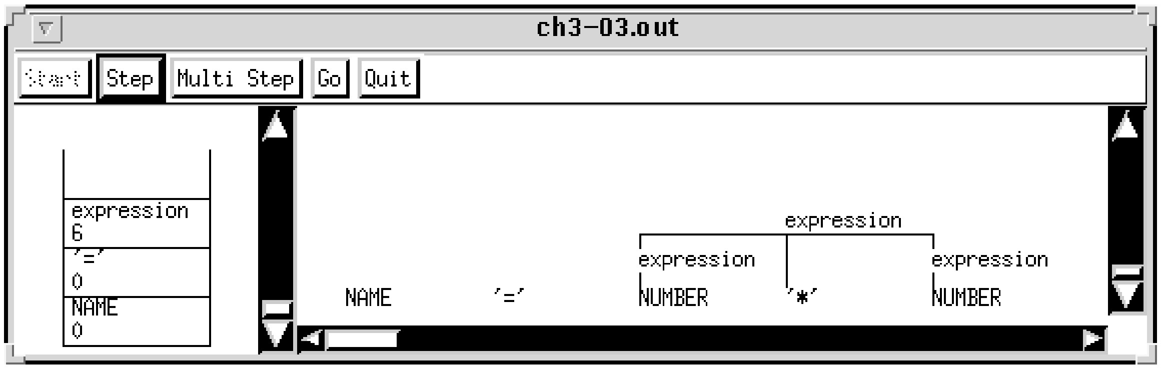

Figure 7: The production `expression` → `expression * expression` is reduced and the stack and tree are updated. `Expression` is given a value $6 (3 * 2)$.

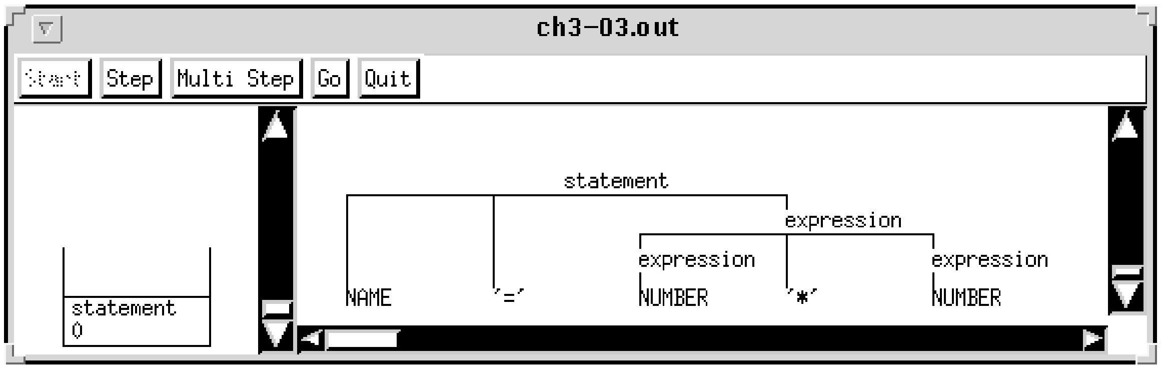

Figure 8: The production `statement` → `NAME '=' expression` is reduced and the stack and tree are updated.

toward helping students understand LR parsing and others also demonstrate evaluation of synthesized attributes. These starting examples provide a basic look at LR parsing and syntax directed translation.

One of the canned examples is that of the previous section. By stepping through the parsing process, the student has a chance to see how the value printed at the end of the parse is computed as attributes are sent up the tree. Consider Figures 6 and 7 in sequence. In Figure 6, the two expression trees have the values 3 and 2 on the stack. In the next step, these trees have been merged with the '*' leaf and the `expression` on the stack now has the value 6.

After looking at the starting examples, the students are encouraged to experiment with them, modifying the grammar or attribute computation. They also can write their own YACC specifications for visualization. After transforming a new or modified grammar via the Visual YACC tool, the student can step through the parse of inputs and watch what happens to the tree, to the LR parse stack and to attribute values associated with stack symbols. Feedback from the students that have chosen to use the tool has been positive.

Using Visual YACC for Debugging

During development of Visual YACC, we realized that it could also be used for the difficult task of YACC grammar debugging. In general, debugging a YACC grammar involves both using the built–in `yyerror()` routine to get information about where in the input string syntax errors were found and manually inspecting the grammar and the auxiliary files generated by the YACC that describe the LR parse tables.

There are many different possible mistakes that can be made when trying to develop a grammar for a language. These mistakes generally manifest themselves in one or more of the following ways.

- The resulting parser does not accept a string known to be in the language.
- The resulting parser accepts a string that is not in the language.
- Attribute values are incorrect in the tree.

In the first two cases, finding the problem means determining where an unexpected shift or reduction has occurred during the parsing processs because the problem lies in some production or set of productions of the grammar. For the first case above, information about what line the syntax error occurred on might be helpful in determining what production or productions are in error. However, when the input string is deemed correct, the automatically generated YACC parser terminates normally, giving no useful information.

To detect the source of these types of problems using Visual YACC, a grammar can easily be visualized on input for which the parser does not behave as intended. By stepping through the parse, the user can identify the incorrect action (shift or reduce) by watching the parse tree and LR parse stack for unexpected behavior. Once the problem is detected, the user can edit the grammar, run this new grammar through the tool and re–visualize to check to see if the correct actions are taken by the fixed grammar. Once debugging is done, changes the user made to the grammar to make it visualizable do not have to be undone because the input grammar will still be standard YACC.

The third case is caused either by incorrect actions associated with the grammar, or by the more subtle situation where, although a parse tree can be produced, it is not the intended parse tree. For example, in Figure 2, changing the + symbol to the * symbol in the action `$$ = $1 * $3` will not change the parse

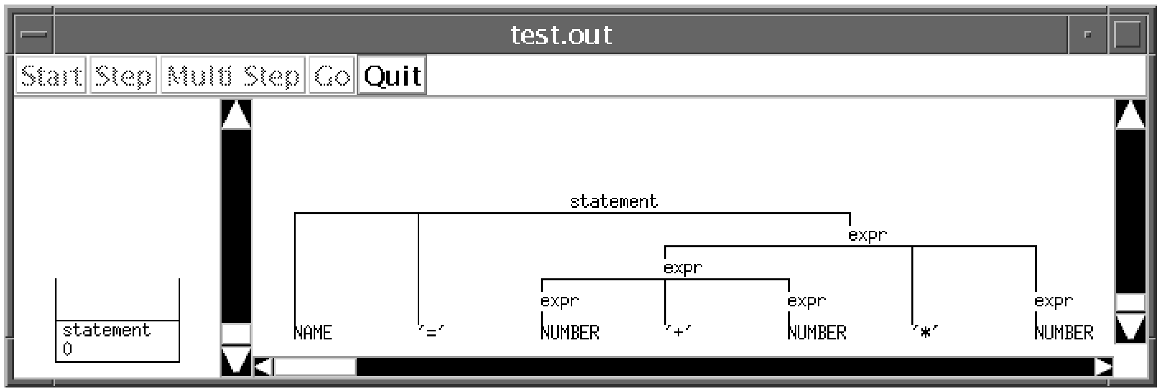

Figure 9: Visual YACC window showing incorrect parse tree for grammar with incorrect precedence for addition and multiplication.

tree generated, but will change attribute computation and what is printed at the final reduction. Without a visualization of the parse, the problem is exhibited only in the final answer and finding the problem involves checking the correctness of all of the actions. With visualization, the parts of the computation that cause the incorrect answer can easily be seen when stepping through the parse.

A different small modification to the grammar of Figure 2 produces an unexpected parse tree and incorrect answers for some strings. The `%left` statements give associtivity and relative precedences to the multiplication and addition operators. If the order of the two statements are switched, the resulting YACC parser will produce a parse tree for the input `x = 3 + 4 * 2`. However, this parse tree, shown in Figure 9, is clearly not the expected tree. Without a visualization, the only information comes from the incorrect answer, just as with the incorrect action described above. With visualization the problem with the parse tree is apparent.

Initial Implementation: C and Motif

Although we wanted to build a visualization environment for parsing, we also wanted to take advantage of YACC's processing power as much as possible. Past experience in building NewYacc[4, 5] has shown us that we can instrument a YACC specification with additional capabilities and this is the approach we took for our implementations.

The process involved in visualizing a grammar with Visual YACC is relatively straightforward. Figure 10 shows the structure of the Visual YACC system. Visual YACC takes as input standard YACC specifications where the user actions and auxiliary code are written in C. The heart of the system is a set of tools capable of automatically annotating an input YACC specification and associated C actions with the appropriate Motif[6] calls to draw how the action (shift or reduction) updates the graphical LR parse stack and the parse tree. This updated YACC specification is further processed into a visualization. Because the visualization must be linked to numerous libraries that the original parser did not require, we provide a shell script that calls all of the appropriate tools required to process the given Lex and YACC code into an executable. The resulting executable can be run by the user with any desired input. The visualization itself is "live", shown during an actual parse, rather than reconstructed from information extracted during the parse.

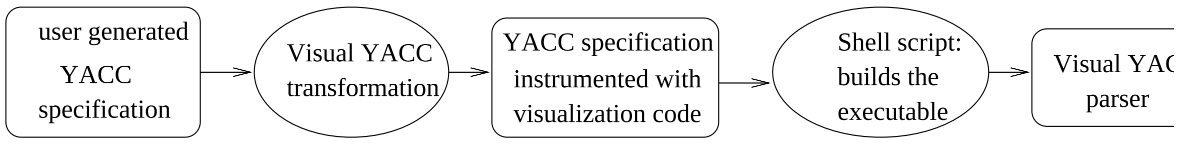
user generated YACC specification
Visual YACC transformation
YACC specification instrumented with visualization code
Shell script: builds the executable
Visual YAC
parser

Figure 10: Visual YACC performs a source–to–source transformation on the original YACC specification to annotate it with Motif calls. A shell script directs the process, linking in the Motif and Visual YACC libraries into the resulting executable.

Drawing LR parse trees

The first issue that we had to resolve during the implementation was the format of the visualization itself. As described earlier, we wanted to display the current parse tree forest at each step. Initially, we believed that this would require a general tree drawing algorithm, but we soon realized that LR parse tree forests grow from the bottom up in only two ways, corresponding to shifts and reductions respectively. During a shift, a new tree (leaf) is added to the forest as the right–most tree. The drawing algorithm we use simply puts all the leaves at the bottom of the screen in the order they are added.

During a reduction, a LR parse tree forest changes when some number of the right–most subtrees are merged into a single tree. Again, the drawing algorithm was straightforward in this case. The new root is placed one level higher than the highest subtree to be merged and at the midpoint of these trees. Lines connecting the root of the new tree to the subtrees were squared off to prevent any of these lines from crossing.

Input Language

One of our primary development constraints was to minimize the changes that a user would have to make to their Lex and YACC specifications in order to use the Visual YACC tool. This meant that we could neither extend the YACC input language (or the Lex input) for the transformation process nor required additional YACC code to be added simply in order to visualize the parse. This constraint turned out to cause many problems; our initial compromise required the user to define and use attributes in the YACC specification in a structured way, including the addition of a data structure intended to hold attributes that needed to be visualized. Although this meant that the input to Visual YACC could be processed directly by Lex and YACC without change, it was not as transparent as we wanted.

We have since added a preprocessor hides these attribute details from the user. Using this preprocessor, the LR parsing process (with no attribute visualization) for a grammar can be visualized with no changes to the grammar. If, in addition, the user would like to see the values associated with some integer attribute `A`, they simply annotate the declaration in the `%union`, changing "`int A`" to "`VYint A`".

Given a YACC specification with or without additional `VYint` annotations, the preprocessor works by both restructuring the attribute data structures to ensure that they are understandable by the transformation tools and by modifying the user code in both the Lex specification and the YACC specification to reflect the new structure. Given this preprocessor, use of the Visual YACC requires only knowledge of how to write Lex and YACC specifications.

YACC annotations

The first fundamental implementation problem was determining how to manually annotate a YACC specification so that as parsing progressed, the visualized stack and tree structures were updated as described previously. This is a syntax–directed translation problem by itself. The resulting parser not only had to compute and pass around the attributes used by the initial specification, but it also had to have data structure attributes that store the parameters that must be passed to the Motif calls. As a simple example, when reducing by a production, it is clear that some number of records (based on the the number of symbols on the right side of the production) needs to be removed from the LR stack representation and some new record needs to be put onto the stack. This information can be determined statically when we are processing the YACC specification. However, where on the screen the old records were and where to draw the new record cannot be determined during specification processing and needs to be dynamically computed based on other Motif attributes in the tree.

During a parse, we need to graphically show the results of both reductions and shifts. Considering reductions first, suppose we start with the following user–generated YACC production:

```
production  :  nonterm1 nonterm2 nonterm3 { user_actions }
            ;
```

This production must be rewritten to include calls to functions that make the appropriate Motif library calls that update both the stack and the tree. These added calls should be placed after the user specified actions.

```
item      :  nonterm1 nonterm2 nonterm3 { user_actions
                          { ParseTreeListHdl list = NULL;
                          $$.tree = NULL;
                          CreateNonLeafParseTree((ParseTreeHdl *)&($$.tree), "item", strlen("item"));
                          SetParseTreeOrgValue($$.tree, $$.x[0], 0);
                          CreateParseTreeList(&list);
                          AddFirst(list, $$.tree);
                          AddLast(list, $1.tree);
                          AddLast(list, $2.tree);
                          AddLast(list, $3.tree);
                          StackReduction(&list);
                          }
          }
          ;
```

In the above code, `tree` is the data structure that holds the dynamically computed Motif window information. `$$.x[0]` is a displayable attribute associated with the `item` non-terminal. If more displayable attributes were specified, they would be used similarly. Information about the rule, such as the name of the new non–terminal and the number of elements involved in the reduction, is used in the required code.

The above techniques work quite well for reductions; however, showing the user the shift of a token from the lexor onto the stack required a different approach. Although it may seem that adding actions inside the production after each token may work, we did not take this approach because this does not guarantee the same behavior as the original grammar and also might require us to make changes to the user written actions to handles the numbering change. This left us with three choices for handling this situation: 1) require the user to make more changes to the Lex specification to extract the appropriate information; 2) make changes to the standard YACC parsing process to determine this information; or 3) modify the grammar with additional productions that signal a shift.

We chose the third option, modifying the grammar. This choice allowed us to use the mechanisms and data structures that we were already using for regular reductions and did not place any additional burden on the user. Visual YACC rewrites the grammar in the following way:

- For each terminal symbol T in the input language, we create a new non–terminal P with an associated rule $P \rightarrow T$.
- Every instance of terminal symbol T is replaced with non–terminal symbol P.
- Actions are associated with this reduction that correspond to the updates required for a shift of a non–terminal.

As an example, consider the following YACC production where non–terminals are lowercase and terminals are uppercase:

```
item       :  nonterm1 TOKEN2 nonterm3 { user_actions }
           ;
```

Based on the above technique, this single production becomes multiple productions:

```
item       :  nonterm1 nonterm2 nonterm3 { user_actions
           { inserted motif reduction actions as above} }
           ;
nonterm2   : TOKEN2 {  $$.tree = NULL;
                       CreateLeafParseTree((ParseTreeHdl *)&($$.tree), “TOKEN2”, strlen(“TOKEN2”));
                       SetParseTreeOrgValue($$.tree, $1.x[0], 0);
                       HandleToken($$.tree);
                       }
           ;
```

These new productions are not shown to the user during the visualization; instead they cause the terminal symbol to be added as a new tree in the forest of trees and cause the terminal symbol to be visually shifted onto the LR parse stack. The visualized behavior of the Visual YACC parser is identical to the behavior of the original.

The other fundamental problem we encountered during this work was interfacing a Motif–based window with the YACC parsing mechanisms. In YACC, once parsing function `yyparse()` is called, control does not return until either 1) the input is entirely processed or 2) a syntax error is found that the parser cannot recover. However, the needs of the tool required interaction between the user and the parsing process. Our solution was to modify the standard YACC parser code to allow it to return control after every step of the process and to resume parsing from an intermediate stage in the process.

Source–to–Source Transformation

Once we had developed the appropriate data structures and syntax directed translation techniques to manually write visualization, the next task was to build a tool that would insert this complex code automatically. Source–to–source transformation of the initial YACC specification seemed the obvious approach[7, 8]. To do this transformation, we used NewYacc, a tool that we have used for other transformational activities[9]. NewYACC is built on top of YACC and it provides a simple way to traverse a given input in very directed ways and to rapidly produce source–to–source transformation tools.

For the visualization tool that we wanted to produce, we needed to traverse input YACC specifications both to extract information about the specification and to rewrite it. First we created with a NewYACC grammar for YACC specifications and built a basic tool that could "pretty–print" the input YACC specifications. Based on what we had learned about annotating the specifications manually to handle reductions and shifts, we then extended this basic tool to add Motif calls in the appropriate locations during the pretty–printing process.

Second Implementation: Java

While the first implementation has proven useful for local use and can be installed at sites supporting the relevant technologies, some of our implementation choices introduce platform dependencies. Because we would like to see wide use of this tool, we decided to create a second version in Java. This version of the tool is currently less mature than the first version primarily because Java compiler tools are not as mature as the C based tools.

We used the front–end of the YACC parser from the initial implementation because this allowed us to take advantage of our previous efforts. Therefore, the main tasks in the Java implementation were 1) how to modify the back end of the parser in order to output the visualization in Java calls rather than Motif and 2) how to make the automatically produced Lex and YACC components mesh together. The first task was accomplished quickly and we do not describe it here. In the remainder of this section we focus on the second task which continues to be an engineering challenge.

To begin, we needed versions of YACC and Lex that would generate Java code. We selected BYACC with Java extensions[10] and JLex[11], a version of Lex that generates Java code. Unfortunately, these two tools were developed independently and did not work together as easily as standard YACC and Lex[1].

An early version of BYACC/Java was used. This version allowed only integers and doubles to be used as attributes in the parse tree. However, the ability to pass more complex information was important to implementing the graphics calls in the original tool. In particular, leaf and node window positions were determined from information lower in the tree, as described previously. To overcome the inability of BYACC/Java to pass information other than integers and doubles, a separate Java stack object was implemented. Attribute information including information relevant to the graphics calls was pushed and popped from this new stack in parallel with pushes and pops to the standard LR parse stack.

BYACC/Java does not have an associated Lex/Java program; instead it expects a user–defined token retrieval function called `yylex()` to be included in the main class for the parser object. Rather then requiring the user to write a Java lexor by hand, we choose to use JLex to create a lexor object. In order to incorporate the lexor object with the automatically produced Java parser, we wrapped the JLex lexor

[1]To date, we still have not found a Java Lex/YACC combination that meshes easily.

into the `yylex()` function, adding code to extract the token field of the JLex lexor and return it. Other information was not passed in this version; however the user can add other information manually.

To use this second version of the tool, the user must perform some tasks:

1. The user writes a JLex specification for the lexor. This specification should have a class called `Yytoken` declared with a public integer called `m_index`. Other information can be included in the `Yytoken` class, but only `m_index` is currently passed to the parser.
2. The user writes a BYACC/Java specification.
3. The BYACC/Java specification is transformed automatically using our tool into a BYACC/Java specification containing Java graphics calls for the visualization
4. The user must edit the BYACC/Java file from the NewYacc tool so that token values used by JLex and BYACC/Java will match. This last step would not be required in versions of Lex and YACC designed to be integratable.

The resulting Java visualization of the input has a different look than that of the Motif visualization; however it contains the same information.

Related Work

There is a large body of work related to program animation and the different styles of visualization[12]. Existing general visualization tools such as XTANGO[13] and Balsa–II[14], focus on languages and environments to describe and support visualizations. This work relies on manual modifications of code to annotate it with extra information for the visualizer. This transformed code is input to tools that graphically display the behavior at runtime. This approach has been used to visualize many different data structures and algorithms, most notably sorting.

Another approach to creating visualizations involves instrumenting the input automatically to create the executable. This is the approach we take in our work. By necessity, tools for automatic instrumentation are targeted to a particular domain, such as object–oriented systems[15, 16].

Our interest was primarily in the parsing domain, where there are several efforts some of which are built using general purpose tools. First, there is a collection of tools, including LLparse and LRparse[17, 18], that can be used for teaching the concepts of automata theory and formal languages, including the building of LL and LR parsers. For a given input grammar, the student is shown interactively how the parsing tools extract the necessary information from the grammar and how to generate parsing tables. Once the tables have been successfully generated, the user can step through a parse using these tables. Parser[19] is a tool that animates LL and SLR (simple LR) parsing algorithms. In addition to the parse tree and the stack, parser actions are clearly shown, as well as the current status of the input string. The Parser package contains a set of starting animations for the students to use and the stuents can also write their own animations by adding control commands to their code. A third related approach to visualizing parsing is GYACC[20]. This tool takes the output from YACC's processing of input grammars and regenerates the tables. Using these tables, the various panels show various different structures relating to the YACC parse, including the stack, a state transition system, and the parse tree. The derivation forest that is shown is a top–down representation of the right–most derivation, rather than a bottom–up forest.

All three of above parsing efforts focus on the internal mechanics of the parser processes: table generation and use. Our work differs primarily in that we are focusing instead on helping users understand how their grammars work, calculate attributes and more generally, now to use these specification techniques effectively.

Conclusion

Visual YACC is an extension of YACC that shows the parsing process visually for a given grammar and input. It works by transforming a YACC grammar into a YACC grammar with graphics calls that allow the user to see both what symbols are on the current LR parse stack and what the current parse tree looks like. It was initially designed to be used in compiler construction courses for teaching about LR parsing and syntax directed evaluation of synthesized attributes. It can also be used in the difficult task of debugging YACC grammars. In this paper we described two different implementations of this tool, one that produces a parser written in C with calls to Motif and a second implementation that generates Java source code. Both were built using the same basic techniques and both are available as public domain software.

There are a number of extensions planned for this tool. We have realized from our use of Visual YACC as a teaching tool that it would be very useful to display the attributes in the parse tree as well as the stack. The very nature of the LR parse stack means that attribute information will eventually leave the stack. Putting the information in both the stack and the tree increases the usefulness of the tool for teaching. The tools could also be extended to better support debugging YACC specifications. In particular, we could extend the interface to allow the user to step through a parse in a manner similar to standard debuggers. Instead of setting breakpoints within the source code, the user could set "break productions", places where the tool should stop and allow the user to examine this portion of the process more closely. There are probably other similar debugging style extensions that could prove useful in this context as well.

Finally, we are in the process of determining how to upgrade the Java version of the tool to make the integration of the parts transparent to the user. We also are building a web–based interface so that it can be used at remote sites via the WWW.

References

[1] S. Johnson, 'YACC: Yet Another Compiler-Compiler', *Technical report*, AT&T Bell Laboratories, 1979.

[2] G. Cornell and C. Horstmann, *Core Java*, Sunsoft Press, 1997.

[3] T. Mason J. Levine and D. Brown, *Lex & YACC*, O'Reilly & Associates, 1995.

[4] J. Purtilo and J. Callahan, 'Parse Tree Annotations', *Communications of the ACM*, 32, 12, 1467–1477, 1989.

[5] E. White, J. Callahan, and J. Purtilo, 'The NewYACC Users' Manual', *Technical Report 2565*, University of Maryland, College Park, 1990.

[6] M. Brain, *Motif Programming: The essentials and more*, Digital Press, 1992.

[7] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language–Based Editors*, Springer–Verlag, 1988.

[8] R. Cameron and M. Ito, 'Grammar–Based Definition of Metaprogramming Systems', *ACM Transactions on Programming Languages and Systems*, 6, 20–54, 1984.

[9] J. Purtilo and E. White, 'A Flexible Program Adaptation System: Case Studies in Ada', *Journal of Systems and Software*, 129–143 (1992).

[10] B. Jamison, *BYACC/Java, Java extension v. 0.9*, http://www.lincom.asg.com/ rjamison/byacc/, 1997.

[11] E. Berk, *JLex: A lexical analyzer generator for Java*, http://www.cs.princeton.edu/ appel/modern/java/JLex, 1997.

[12] B. Price, R. Baecher, and I. Small, 'A Principled Taxonomy of Software Visualization', *Journal of Visual Languages and Computing*, 4, 3, 211–266, 1993.

[13] J. Stasko, 'TANGO: A Framework and System for Algorithm Animation' *IEEE Computer*, 23,9, 39–44, 1990.

[14] M. Brown, 'Exploring algorithms using Balsa-II', *IEEE Computer*, 21, 5, 14–26, 1988.

[15] H. Dershem and J. Vanderhyde, 'Java Class Visualization for Teaching Object–Oriented Concepts', *Twenth-ninth SIGCSE Technical Symposium on Computer Science Education*, pp. 53–57, 1998.

[16] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides, 'Visualizing the Behavior of Object–Oriented Systems', OOPSLA 1993, pp. 326–337.

[17] S. Blythe, M. James, and S. Rodger, 'LLparse and LRparse: Visual and interactive tools for parsing', *Twenth-fifth SIGCSE Technical Symposium on Computer Science Education*, pp. 208–212, 1994.

[18] A. Bilska, K. Leider, et. al. 'A Collection of Tools for Making Automata Theory and Formal langauges Come Alive', *Twenth-eighth SIGCSE Technical Symposium on Computer Science Education*, pp. 15–19, 1997.

[19] S. Khuri, and Y. Sugono, 'Animating Parsing Algorithms' *Twenth-ninth SIGCSE Technical Symposium on Computer Science Education*, pp. 232–236, 1998.

[20] M. Lovato, and M. Kleyn, 'Parser Visualization for Developing Grammars with YACC' *Twenth-sixth SIGCSE Technical Symposium on Computer Science Education*, pp. 345–349, 1995.