Understanding the bottom-up SLR parser

Conference Paper *in* ACM SIGCSE Bulletin · March 1994

DOI: 10.1145/191029.191163 · Source: DBLP

2 authors, including:

Sami Khuri
San Jose State University
86 PUBLICATIONS **970** CITATIONS

SIGSCE 94- 3/94, Phoenix, Arizona,USA

error in the input string as fast as the LR parsers.

The general model of the LR parser is reproduced from [Aho86, page 217] in Figure 1. Basically, the model consists of:

1) An input string of tokens. These are produced by the previous phase, the lexical analyzer, to which

general model is reproduced from [Hill90, page 386] in Figure 2. The only picture of the outside world the "eye" sees is that portion of the world it captures in the window.

describes a complete scene. In order for Synthetica to capture a scene as a cel, it requires a camera definition. Therefore, the first construct of every scene that the compiler expects is a camera definition (see Figure 3a,

```
1-  /* an example file */
2-  camera { position   < 0,1,0.5 >
3-           look_at    < 0,0,0 >
4-           up         < 0,-0.5,1 >
5-         }
6-  #define trapped = < object { sphere } object {cube } >
7-  object { trapped rotate < 50,0,5 > scale < 0.5,1,1.3 > }
8-  end_scene
9-
10- camera { position   < 0,1,0.5 >
11-          look_at    < 0,0,0 >
12-          up         < 0,-0.5,1 >
13-        }
14- object { trapped rotate < 10,0,5 > translate < 5,0,0 > }
15-                    .
                       .
                       .
    end_scene
```

Figure 3. a) An example file

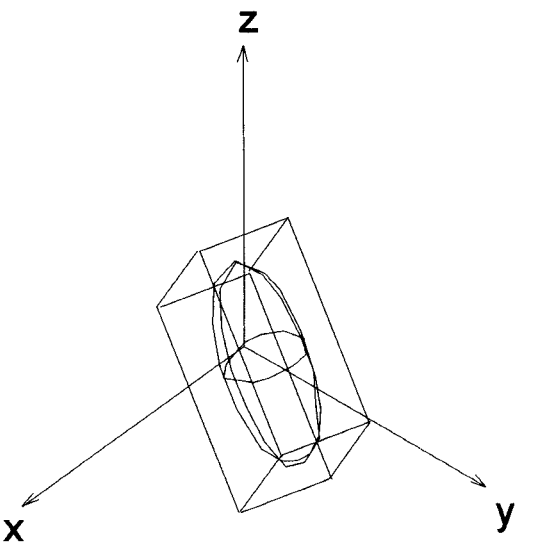

Figure 3. b) Result of first scene

lines 2 and 10). The camera construct requires three vectors: position, look_at, and up (vector v shown in Figure 2). These vectors give the camera's position, where to focus, and which direction is up. In Synthetica a vector is represented by a 3-tuple of the form <x,y,z>. Once the camera is defined a description of a scene can be made.

Primitives can be placed anywhere and in any manner that the user decides by using translations, rotations, and scaling. They may also be grouped together to form larger more complex objects, like the "trapped" object defined in Figure 3a. These groupings of objects are created using the #define directive.

When the compiler encounters a #define directive, it creates the necessary data structure to accommodate the associated primitives and directives. Finally, the object directive instructs Synthetica to draw the object on a virtual screen and to store this information for later use by the animator. For a quantitative description of the matrix operations involved in the transformations: rotate, scale, and translate, the reader is referred to [Hill90].

As with any scene description in Synthetica, Figure 3a begins with a camera definition giving the camera's position: x=0, y=1, z=0.5, the coordinates of where the camera is looking: x=0, y=0, z=0 (the origin of the world), and which direction is up: x=0, y=-0.5, z=1. Each triple represents a vector with its tail at the origin and head at the position specified by the triple. Line 6 of the example file gives the definition of the first object that will be drawn in this scene. This line instructs Synthetica to set up a data structure in the symbol table that will hold two objects, a sphere and a cube, each with its origin at the same position. Nothing is displayed until line 7. This line instructs Synthetica to draw the "trapped" object onto a virtual screen. Whenever the compiler encounters the object statement outside of a #define statement it begins to build a list of animation instructions. In this case, line 7 instructs Synthetica to draw "trapped" at the origin rotated 50 degrees about its x-axis, 0 degrees about its y-axis, and 5 degrees about its z-axis. Moreover, line 7 tells the compiler to scale the "trapped" object 0.5 times along its x-axis, 1.3 times along the z-axis, and no scaling along its y-axis. The overall effect will shrink "trapped" along its x-axis and stretch "trapped" along its z-axis. Once these operations are completed, the compiler will then draw "trapped" on a virtual screen. The information drawn on this screen is captured in a cel for later use by the animator. Figure 3b represents what is captured in a cel after the execution of line 7; the coordinate axis are shown only as an aid to the figure-they would not appear in the generated cel. In the next section we give a more detailed description of Synthetica's compiler.

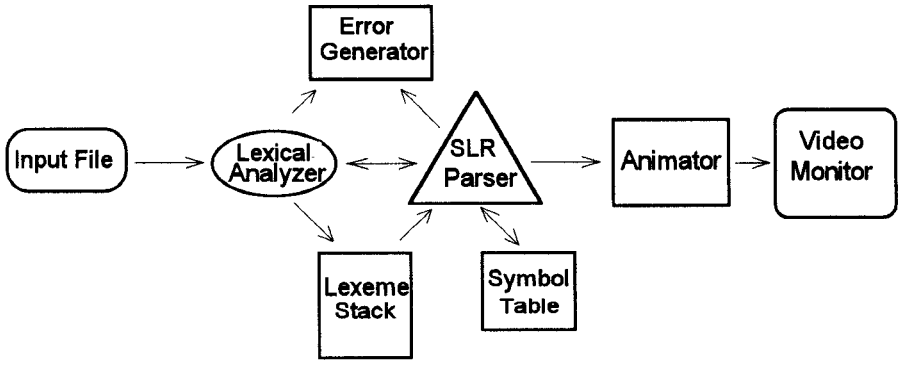

Figure 4. The Compiler Environment

3.2 SYNTHETICA'S COMPILER

There are two phases in the process of creating animation with Synthetica: the SLR parsing phase and the display phase. The SLR parser was developed to convert the scene descriptive language into information that Synthetica's animator can use. As is the case with the traditional use of parsing, in our application, the SLR parser has three distinct phases: lexical analysis, syntactic analysis, and intermediate code generation. Each of these phases work in conjunction with one another to produce the final animation (see Figure 4).

In Figure 4, the three phases of the compiler are shown graphically to depict their relation with one another. The input file (similar to our example shown in Figure 3a) is read and broken into tokens by the lexical analyzer. These tokens are retrieved one by one from the input file, via a buffer, and fed to the parser upon request. The SLR parser, modeled in Figure 1, then begins syntactic analysis of these tokens. During this phase the compiler makes the appropriate semantic actions to produce the animation instructions.

The first phase of the Synthetica compiler is lexical analysis. Synthetica's lexical analyzer uses a 13x19 state table reproduced in part in Figure 5, for generating tokens and detecting errors. In the top row of the table, "l" stands for letter, "d" stands for digit, "sp" for space, and "p" for general punctuation. The column labeled "Back up" informs the analyzer when to move back one character in the input string. The underlined states in the table represent accepting states, which signal the acceptance of a token.

The analyzer is a basic finite state machine with a few embellishments. It handles several special cases, such as when to ignore white spaces and when it signifies the end of a token, the removal of comments, the separation of keywords and identifiers, and notification of illegal characters. Each state of the analyzer is found by consulting the lexical analyzer state table (see Figure 5). Depending on the current state and current input character the analyzer knows what action to perform: produce a token, generate an error, or move to another character in the buffer and change states. All input to the lexical analyzer comes from a buffer which holds a small amount of data read from the input file.

Synthetica's lexical analyzer uses a buffering technique similar to double buffering [Pars92]. Each time that the parser requests a token the lexical analyzer retrieves a string from the buffer and converts it to the appropriate token. Once the lexeme is converted to a token, it is passed to the parser as a single character. A token is only collected from the input buffer on request by the parser. Retrieving tokens is not the only function of Synthetica's lexical analyzer. The analyzer also saves critical data, such as: identifiers, vector triples, and floating point values. These data are pushed onto the lexeme stack, shown in Figure 4, and later popped from this stack by the parser during a semantic action. Without this lexeme stack any identifiers, triples, or floating point values, would be lost when converted to a token. Therefore, when the lexical analyzer encounters an object name, vector triple, or floating point value, it immediately pushes these values onto the lexeme stack, thus saving their values for later use. Using a stack for storing this data accomplishes two things: conservation of memory and increased speed. The code for Synthetica's analyzer is based on the procedure LEX_SCAN given in [Pars92].

The second phase of the Synthetica compiler is syntactic analysis. This is the heart of Synthetica and is accomplished through the use of the bottom-up SLR parser. This parser is constructed with four major interfaces (see Figure 1), and several small support systems. The major interfaces are: a stack, a table, an input, and an output. The support systems include a lexeme stack, a symbol table, and an error generator (see Figure 4). The stack for the parser is simply a state stack, similar to that for the bottom-up SLR parser discussed earlier and in [Pars92]. Synthetica's SLR stack only holds the states of the parser, no grammar symbols are pushed. As described earlier in part 4 of the LR parser model, each action is found by consulting the SLR parsing table and the top of the SLR stack.

The SLR table for Synthetica has 68 states and

Current state	l	d	{	}	I /	n *	p =	u <	t >	#		sp	p	Back up
1	2	4	7	8	9	13	14	15	16	17	19	1	19	yes
2	2	2	3	3	3	3	3	3	3	3	3	3	3	no
3	1	1	1	1	1	1	1	1	1	1	1	1	1	yes
4	5	4	5	5	5	5	5	5	5	5	6	5	5	no
5	1	1	1	1	1	1	1	1	1	1	1	1	1	yes
•							•							•
•							•							•
•							•							•
19	1	1	1	1	1	1	1	1	1	1	1	1	1	no

Figure 5. Lexical Analyzer State Table

In this paper we have illustrated the concepts behind the bottom-up SLR parser. These concepts were used to create our computer animation example, Synthetica. We believe that our application is suitable for students enrolled in a compiler class for the following reasons:

1) Hands-on experience: It offers the hands-on use of the LR parsing technique in a novel and interesting application.
2) Simplicity: The model is comparatively easy to understand. It consists of 3 camera actions and 3 geometric symbols.
3) Realistic: Most parsing examples in the popular compiler textbooks are based on grammars with at most twenty to twenty-five states. Our model has sixty-eight states. It is far from matching the huge number of states found in commercially available parsers, but nevertheless, it is closer to reality and yet not too huge for tracing through the parser with pencil and paper.

[Ho93] Ho C. F., Morgan C. L. & Simon. "An Advanced Classroom Computing Environment and its Applications." *ACM SIGSCE Bulletin.* Vol. 25(1), February 1993: 228-231.

[John75] Johnson, S. C. "YACC--yet another compiler-compiler." *C.S. Tech.* Report no. 32., Murray Hill, N.J., Bell Telephone Laboratories, 1975.

[Khur86] Khuri, S. "Counting Nodes in Binary Trees." *ACM SIGCSE Bulletin.* Vol. 18(1), February 1986: 182-185.

[Pars92] Parsons, T. W. *Introduction to compiler construction.* New York: Computer Science Press, 1992.

[Wells93] Wells, Drew and Cris Young. *Ray Tracing Creations: Generate 3-D Photorealistic Images on the PC.* CA: Waite Group Press, 1993.