A Process-Visible Compiler Aimed for Teaching Assistant

Xiwen Chen
Sophia Team of East China Normal University
Shanghai, China

Hanfei Lin
Sophia Team of East China Normal University
Shanghai, China

Yufei Liang
Sophia Team of East China Normal University
Shanghai, China

Xiaoming Ju
Sophia Team of East China Normal University
Shanghai, China
xmju@sei.ecnu.edu.cn

***Abstract*--The course of Compiling Principles has always been difficult for students to understand, for the related algorithms are quite complex, and not all students possess the ability to comprehend those algorithms easily. In order to facilitate the learning of compiling techniques, we have designed a special compiler which displays the whole process of compiling for observing, and it's suitable to assist teaching work. Based on the subset of C grammar, this compiler can show the executing process of compiling algorithms with abundant graphs and messages. As an assistant tool for teaching, not only does it contribute to the quick, efficient understanding of compiling principles, but it also add some fun to the course.**

***Keywords*--compiler; process-visible; teaching-assistant**

I. INTRODUCTION

With the development of software industry, high-qualified software developers are much in demand. At the same time, as a core course in IT education, such as Compiling Principles[4], is key to the cultivation of programmers, cause via the process of understanding the essence of compiling students will develop the ability to comprehend complex algorithms so as to become able to design sophisticated algorithms by themselves. While teaching, however, teachers always find it difficult to reach this goal, for these algorithms are abstract and hard to illustrate, which impedes students' comprehension.

To improve learning efficiency, we developed a process-visible compiler which assists the teaching work. The term 'process-visible' means that all procedures in compiling can be dynamically showed in a step-by-step way, which hasn't been achieved by any other compilers before. Making use of this tool, students will firstly get a perceptual cognition towards compiling techniques, and then come to understand the whole compiling scheme clearly. Since every compiling step and its result is visible, the algorithms will be mastered in an easy way, turning this troublesome course into an interesting one.

II. RELATED WORK

There has been some work bout process-visible compilers. In [3], the result of lexical analysis is saved in files, and during syntax analysis the First sets, Follow sets as well as LL(1) table is displayed on the interface. However, its visualization of compiling is not enough, since only the syntax part is visualized.

The authors of [1] take a different measure: they insert new codes which conduct visualization into original programs with YACC, and then compile the new program in the common way. While those inserted codes are executed, the pursued visual effect will come into view, such as printing the current position of compiling. Though this viewpoint is novel, codes that can be inserted are limited, and it can only show some superficial work of compiling.

In [2], the moves of state machines in lexical analysis and stack status through syntax analysis are demonstrated, along with the error messages produced while compiling. This article displays the most excellent work among all related researches, but it doesn't cover the visualization of semantic analysis and intermediate representation.

So based on these work, we established a sound process-visible compiling system, which functions not just as a normal compiler, but can also vividly display the procedures of lexical, syntax, semantic analysis, and intermediate representation. Moreover the dynamic display of symbol table as well as user-friendly error prompts are also supported, strongly facilitating the teaching of compiling algorithm.

III. THE ARCHITECTURAL DESIGN OF A PROCESS-VISIBLE COMPILER

A. Overall Structure

This compiling system contains the compiling part and the visualization part, among which the former just acts like a common compiler that finishes lexical analysis, syntax analysis, semantic analysis and intermediate implementation.

Correspondingly, the visualization part exhibits real-time status of algorithm executing while compiling. Every step is displayed on the interface dynamically in a transparent and continuous way, helping students come to know each compiling procedure respectively.

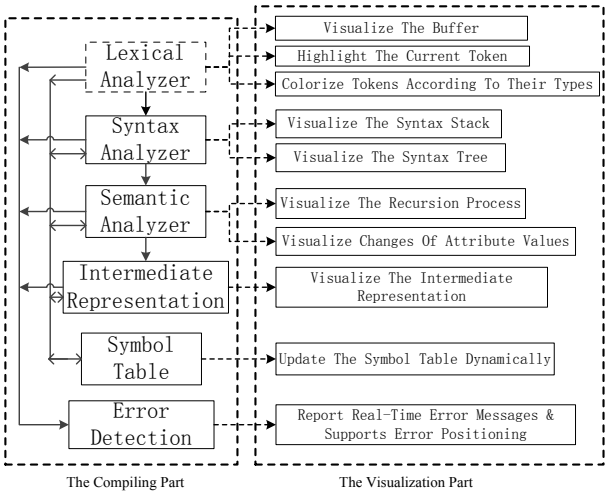

Figure 1: The architecture of compiler and visualization

The system architecture is as Figure 1, The visualization part contains following functions:

- In lexical analysis, in simulation of the moves of buffer pointers during compiling, on the interface the current lexeme being analyzed will be highlighted, and after analyzation all lexemes are colored differently according to their token types.
- In syntax analysis, changes of the syntax stack are dynamically exhibited, including stack content, stack pointer and stack growth orientation. At the same time a syntax tree is drawn on the basis of production rules being used, and is displayed on an independent window.
- In semantic analysis, nodes that should be assigned values are marked and colored on the syntax tree.
- Intermediate representation shows the current sentence being analyzed and the according assembly codes.
- Contents of the symbol table are updated as the compiling goes on.
- Errors are reported as soon as they are discovered, and the causes can be located in the source programs.

B. The Visualization of Lexical Analysis

Lexical analyzer is the basis function module of a compiler, which scans and splits source programs into tokens and then return them to the syntax analyzer after an initial scan of source codes. The lexical analyzer is called by syntax analyzer; every time it is called a token will be split out and then delivered for syntax analysis, after which it will stop and just wait for the next call.

We use such a logical to extract tokens: firstly regular expressions of all token types (e.g. identifiers, numbers) are established, then scan the source program character by character. Since between every two "split-sign" (signs that can partition identifiers and numbers, including delimiters like brackets, semicolons, as well as operators like plus, minus) there will be a identifier or a number, so every time the lexical analyzer gets a split-sign we can just return the string in front of it, after which this split-sign is also passed to syntax analyzer as a token.

In addition, a functional lexical analyzer is also supposed to finish some auxiliary tasks such as filtering comments and storing identifiers into symbol tables. The visualization of symbol table will be elaborated later.

The visualization of lexical analysis is implemented in this way:

In the book [4] a buffer with two pointers are used during lexical stage. The pointer beginning points to the first character of the current token, and the pointer forward moves onward as the reading of source program continues. When the string between beginning and forward forms a token, it is returned, and the two pointers are reset for the next analyzation.

To display this procedure, while tokenizing the current lexeme being analyzed will be highlighted, and all tokens that have been recognized are colored according to their types. For example, identifiers are X, and numbers are X. So the steps of lexical analysis can be well traced, making it more direct to observe how lexical analyzer works.

C. The Visualization of Syntax Analysis

Syntax analyzer is the core of a compiler's front-end part. It invokes lexical analyzer to get tokens one by one, which are compared with given production rules. If some tokens conform to one of those rules, they will be regarded as a right syntax unit and then be passed on to semantic analyzer. If none of the rules can be applied errors will be reported.

Since the whole process-visible compiler is written by hand, compared to LR(1) which can only be implemented by tools such as YACC here the relatively simpler algorithm LL(1) is adopted. After calculating First and Follow sets for production rules, an LL(1) syntax table is established, and at the same time a syntax stack is formed to analyze the tokens provided by lexical analyzer. When a legal syntax block is recognized its according syntax tree is drawn, which is made use of by subsequent analysis.

The visualization of syntax analysis is implemented in these ways:

The virtualization of syntax stack. Syntax stack is the most significant data structure during syntax analysis, and is also the most inaccessible part for students. Our process-visible compiler aimed for teaching assistant will redraw the stack on the interface every time its state changes, so that students could observe the inputs and outputs of syntax analysis clearly.

The virtualization of syntax tree. Whenever the stack outputs a production rule (meaning that the current code part being analyzed is judged as right in grammar), a new sub-tree is appended to the current syntax tree. If "single step" mode is selected students can watch how a huge syntax

tree is formed step by step, which visualizes the analysis process clearly.

D. The Visualization of Semantic Analysis

Semantic analysis is mainly responsible for type checking and symbol table updating. It can also detect some simple errors such as a zero divider. On the basis of semantic analysis it's convenient to implement intermediate representation.

The analysis is conducted by combining syntax-directed translation with recursive syntax tree reference. Every production rule is enlarged with an attribute grammar, and once the analysis is launched the syntax tree will be recursively traversed, the attribute grammar of whose nodes are executed afterwards. Syntax tree is established by syntax analyzer, and its nodes contain related production rules, as well as terminals or non-terminals along with their attributes.

The visualization of semantic analysis is implemented in these ways:

The visualization of recursion. The process of recursively traversing syntax tree is displayed by changing the color of the node that is currently being visited. Every time the semantic function is called, tree nodes that it covers will be highlighted by a conspicuous color, so that students can observe the whole procedure of semantic analysis in detail. Of course, "single step" mode may offer much clearer representation.

The visualization of changed attribute values. When a node is under syntax-directed translation its attribute values might be updated, and since this is reflected on the interface students will be able to understand how semantic analysis deals with attributes.

E. The Visualization of Intermediate Representation

The computer can only handle three addresses while dealing with assembly programs, so compilers are supposed to transfer high-level language programs into intermediate three-address codes firstly. Using the same idea of recursion in semantic analysis, the intermediate code generator produces corresponding three-address codes for every syntax block which has been semantically analyzed.

The visualization of intermediate representation is implemented in this way:

The code generator will print the current source codes being analyzed and the according intermediate codes onto the interface, so that students can easily check their correspondence.

F. The Visualization of Symbol Table

The symbol table specially stores attribute values of identifiers, and the information in symbol table will be employed during the whole compiling procedure. For instance, in semantic stage identifier attributes are used for semantic check such as examining if the usage of an identifier name applies to its definition.

The visualization of symbol table is implemented in this way:

The contents will be updated dynamically through the whole analyzation work. Being able to see the real-time changes of symbol table, students will indeed understand the role it plays in compiling.

G. The Visualization of Error Prompt

How to deal with errors is a key criterion for judging the function of a compiler. This process-visible compiler is equipped with a sound error-prompting scheme: when compiling finishes, all errors will be printed in a table with accurate and complete descriptions, and from the table it's also known at which each error occurs, where the according source codes will be highlighted when this error message is double-clicked in the table. At the same time, the interface area which displays results of syntax analysis will also scroll to the corresponding location, so that students can figure out how it goes wrong.

Errors can fall into the following categories in this compiling system:

Lexical errors. This can be caused by some illegal token names (such as 100jsj), and we define such token type as Unknown for just reporting its appearance. However, once an error emerges during lexical stage the following analyses will definitely not be able to go smoothly. Though there are ways to do some simple correction, most errors cannot be rectified, so in our logic all subsequent stages won't start as long as errors are discovered during lexical analysis, and only lexical analyzation will be completed at last. In the interface area of lexical results, the token type of each token can be referred, so it's easy to correct lexical errors manually.

Syntax errors. There are two situations in syntax error: redundant component or lost component. As in lexical analysis, once errors appear during syntax stage, semantic stage and intermediate representation won't start. With reference to results of syntax analysis it's also convenient to rectify errors, since every production rule printed is marked with the location of its corresponding source code part.

Semantic errors. There are also two situations: variables are used for calculation before being assigned values to, or the divider appears to be zero.

IV. A SIMPLE EXAMPLE

In this section we show how to use this process-visible compiler to assist the teaching of compiling algorithms with the case of assignment statement.

A. The Main Interface

As in Figure 2, the main interface can be divided into three parts--up, middle and down. The up area contains menus and buttons for operation; the middle area shows real-time results of each analysis stage; And finally in down area the contents of symbol table as well as error messages will be dynamically updated.

B. The Usage

At first, choose the File menu->Open Source File to import a code segment which is to be compiled, and its content will appear in the Source Code box; or we can input

codes directly in the box. Then choose View the Syntax Tree and press the OK button. This will allow us to see how syntax trees for each syntax block are constructed in a new window.

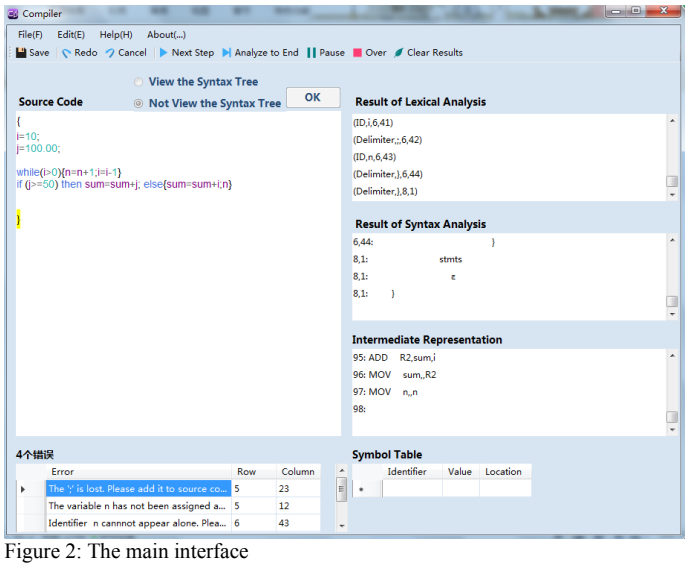

Figure 2: The main interface

If Analyze to End is clicked, then the whole analyzation will be executed automatically. Teachers should click Next Step manually when they intend to explain every step in the process to students.

While compiling, in Source Code box the current token being analyzed will be highlighted, and at the same time its information is appended to the Result of Lexical Analysis box. Matched productions are printed in the Result of Syntax Analysis box, and every time a correct syntax block is formed its corresponding syntax tree is drawn.

For example, if an assignment statement is formed, the syntax analyzer firstly draws its syntax tree, and in next stage the semantic analyzer traverses the tree recursively to calculate attribute values for tree nodes. In the case of assignment statement, the semantic analyzer will calculate the value of node arithexpr (the third child node of node assgstmt) ,after which it updates the ID attribute value of the first child node of node assgstmt in symbol table. Figure 3 is the snap-shot of the time when node simpleexpr is being visited. In this way details in syntax and semantic analysis can be lucidly presented, which is easy for teachers to explain and vivid for students to observe.

Meanwhile the intermediate representation is also in progress, and its analyzation result is exhibited in Intermediate Code box.

During the whole compiling procedure the symbol table will be updated as long as some attribute value of an identifier is changed, and error table gives all error messages dutifully. Clicking on a certain message will lead to the positioning of possible causes in Source Code box and Syntax Result box, so that it's convenient to analyze the reason, which helps to deepen students' understanding.

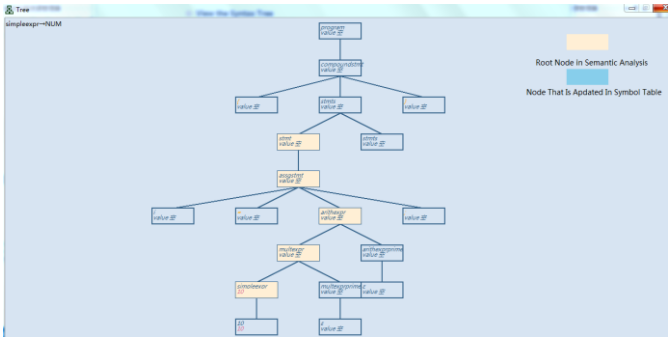

Figure 3: The interface

V. CONCLUSION

Our process-visible compiler aimed for teaching-assistant consists of a lexical analyzer, a syntax analyzer, a semantic analyzer and a three-code address generator whose working procedures are all dynamically visible. The whole compiling process is displayed by graphs and colorization, which promotes students' thorough understanding of compiling algorithms. Educational practices show that this compiler has served as a good assistant tool in the teaching of the course of Compiling Principles.

REFERENCE

[1] Dan,J.P.& B.T.Du.The Visual Implementation of programs, Based on Compiling Technology[J].Computer application research,2002(10):51-52,70.

[2] Li,X.Research And Implementation on Visualization of Compiling Procedure[D].He Bei:Hebei University of Technology, 2011:1-65.

[3]Yiming.THE VISUAL IMPLEMENTATION of CLASSIC ALGORITHM of COMPILATION PRINCIPLE[D].Chang Sha:Changsha University of Science & Technology, 2012:1-47.

[4] Alfred V. Aho. Monica S. Lam. Ravi Sethi. Jeffrey D. Ullman. Compilers: Principles,Techniques, and Tools [M]. New Jersey: Macmillan, 1985: 42-45.