Experiences in Building C++ Front End

Fuqing Yang, Hong Mei, Wanghong Yuan, Qiong Wu, Yao Guo
Department of Computer Science and Technology
Peking University
Beijing 100871, P. R. China
E-mail: whyuan@hotmail.com

Submitted to SIGPLAN Notices for publication.

Abstract It is difficult, if not impossible, for the code analyzers to employ front end from compilers, because these front ends extract different program information using different strategy. This paper describes our experiences in building C++ front end as a part of code analysis toolset. The front end customizes the lexical analyzer incorporating a special preprocessing technique to accurately associate program entities with physical source code location. To support analysis of different C++ languages, the front end employs YACC to generate the parser, and uses token lookahead technique to disambiguate C++ grammar for YACC.

Keyword compiler, front end, C++, CASE, object orientation, program analysis

1. Introduction

Many CASE environments need to parse program source code. Front end usually transforms source code into an internal representation, such as structure parse tree, symbol table, program database, and so on. Analysis tools then use the internal representation to generate executable code, class dependencies graph, message passing graph, test cases, metrics data, etc.

CASE tools that require parsing of source code can be broken into three classes: syntax-directed editors, compilers and interpreters, and code analyzers[1], which are quite different in the program information needed. Syntax-directed editors, for example, IPE[2] and Cornell program synthesizer[3], need relatively simple parse tree and symbol table to assure syntactic correctness while editing code. Most information kept in the parse tree and symbol table is about syntax and is not needed after the end of editing. Compilers and interpreters need a great amount of information about the source code to generate the executable code. It is only a short time for them to keep a complete parse tree and an elaborate symbol table, which are usually not needed after the compiling session. Code analyzers, say, BDCom-C++[4], CIA[5] and CIA++[6], need a large deal of program information, which is kept in the parse tree or the symbol tables while parsing. After parsing, they usually store the information about source code permanently into files, database, and so on, for later analysis.

These differences mean the front ends of these tools extract different program information according to different philosophy. In a code analyzer, it is important to represent the program accurately and precisely, therefore, the front end must associate program information with the physical source code location. It is a difficult task for programs using preprocessing mechanisms, of which the most notable are C++ programs. As a result, even the compiling technique is full-blown and compiling tools are ready-made, it is difficult, if not impossible, for the code analyzers to employ front end from other parsing tools designed to other applications.

We built a front end for C++ as a part of JBPAS(JadeBird Program Analysis System), a toolset of code analysis for C++ programs. Instead of using standard compiler tool LEX[7], We customize the lexer(lexical analyzer) for the front end, incorporating a special preprocessing into lexical analysis. Receiving physical source code, the customized lexer can extract program information, say, comments and include files relationship, needed for program analysis. Moreover, it accurately associates C++ program entities, such as class, object, statement, macro, and so on, with their physical location in the source code. To easily update the front end to support analysis of different C++ languages, we implemented the parser around YACC[8]. Normally YACC accepts LALR grammar, but C++ grammar is inherently ambiguous and definitely not LALR. We employ token lookahead technique to disambiguate the parser generated by YACC.

This paper describes our experiences in building C++ front end and is organized as follows. First, section 2 presents an overview of JBPAS. Section 3 details C++ preprocessing, in particular, macro substitution, in the C++ front end. Section 4 discusses parsing C++. Finally, section 5 gives the current status and future work.

2. JBPAS Toolset

The initial objectives of our C++ front end are to support our research of program understanding, and we have implemented the prototype version of a C++ program understanding system: BDCom-C++[4]. BDCom-C++ consists of three major components: an information extractor, an information manager and a user interface, as shown in figure 1:

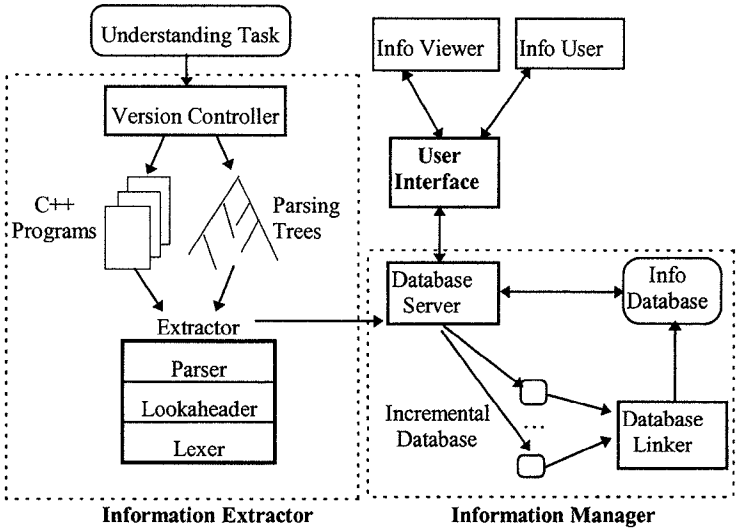


Figure 1. BDCom-C++ System Overview[4]

The essential part of the information extractor is the C++ front end, which analyzes C++ source code by means of incremental parsing and extracts program information to store into the program information database. Later we saw several code analysis tools based on the front end and anticipate more would appear in the future. Therefore we design a tools kit, called JBPAS, to support research in software maintenance, software reuse, reverse engineering, software metrics, and so on.

Figure 2 shows the architecture of JBPAS:

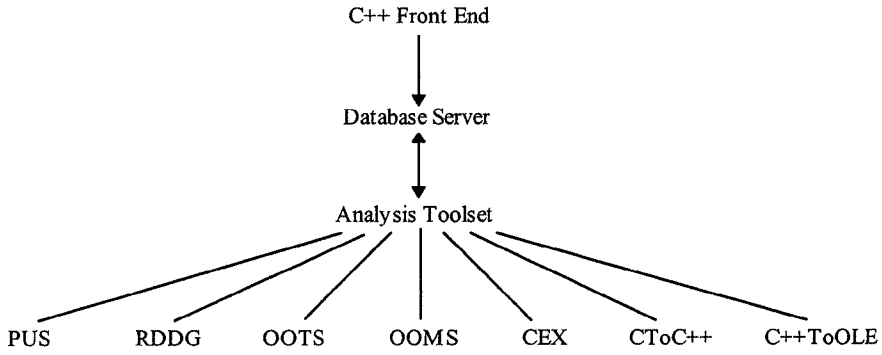


Figure 2. JBPAS Architecture

- ***PUS:*** *Program Understanding System.* PUS loads information from the program database, then organizes and shows the information according to the abstract views needed, and at the same time, switches between the abstract views to represent the program from different perspectives.
- ***RDDG:*** *Reverse Design Document Generator.* The available design documents is often either inadequate or incorrect for technological and management reasons. Reverse design document generator use the information extracted from the program to automatically recover C++ program's object-oriented design[9] documents.
- ***OOTS:*** *Object-Oriented Test Supporter.* Traditional testing tools are inadequate for object-oriented programs because of their new features. OOTS helps to determine test cases based on the program information and supports C++ program testing.
- ***OOMS:*** *Object-Oriented Metrics Supporter.* Object-oriented metrics are an integral part of object technology and of good software engineering[10], and the useful measures should be automatible to collect metrics data[11]. The object-oriented metrics supporter collects metrics data on code from the program information.

- *CEX: Component Extractor.* Software reuse, of which an important part is component-based reuse, is considered as a practical and feasible approach to solving the software crisis[12]. Based on program understanding, the component extractor extracts reusable component through implementing re-engineering on the class or class cluster acquired from existing software.
- ***CToC++:** C To C++ Translator.* Because C++ is the superset of C, the C++ front end and database server can also apply to C programs. C to C++ translator helps user restructure C programs, locates data structure and its relevant functions, and translates into functionally equivalent C++ program.
- ***C++ToOLE**: C++ To OLE Translator.* OLE(Object Linking and Embedding) provides a way to integrate and inter-operate between applications[13]. With the help of the C++ to OLE translator, programmers can encapsulate C++ applications to OLE components conforming to OLE interface standard.

The front end analyzes C++ source code, extracts program information and stores it into the program information database through the database server. All the above analysis tools use the program information via the database server. Only the front end needs to know the exact syntax of the C++ language, which means that when the target C++ language changes, it only affects the front end.

3. Preprocessing in JBPAS

In a typical C++ compiler, the source code usually passes through following processing phases: line splicing, tokenization, preprocessing, string concatenation, translation, and linkage[14].

3.1 Preprocessing Phase

During the preprocessing phase, the compiler usually processes file inclusion, conditional compilation, and macro substitutions.

File inclusion treats the contents of include file as if it appeared in the source program at the point where the directive ***#include*** appears. Typically, constant, macro definitions, and complex data types are organized into include files and then use the directive ***#include*** to add them to any source file.

Conditional compilation determines which text blocks are passed on to the compiler and which text blocks are removed from the source file during preprocessing by testing a constant expression or identifier. This ability allows a single source file to generate different programs, say, to include debugging information only in debug versions.

Macro is typically used to associate macro name, a meaningful identifiers, with macro body, which can be constants, keywords, and commonly used statements or expressions. When the macro name is encountered in the following program source text, it is replaced by a copy of the macro body. If the macro accepts arguments, the actual arguments following the macro name are substituted for formal parameters in the macro body. Macro definition and substitution is easily described with an example. Suppose that a file called MACRO.CPP has following code segment:

```
[1]  #define PI                    3.14
[2]  #define AREA( r )             PI * ( r ) * ( r )
[3]  #define VOLUME(r , h)         ( h ) * \
[4]                                  AREA( r )
[5]
[6]  float v = VOLUME( 5.0, 4 ) ;
```

Figure 3. Code Segment in MACRO.CPP

A normal preprocessor will expand *VOLUME(5.0, 4)* to *(h) * 3.14 * (r) * (r)* .

3.2 Preprocessor of JBPAS

Preprocessing mechanisms certainly can increase programmer productivity; but on the other hand, it unfortunately incurs an additional burden of program understanding. The antinomy lies in the fact that typically the compiled program is based on the preprocessed code, however, the program understander interacts with the physical source code.

To response to various requirements based on program understanding, we design special preprocessing algorithm to avoid the penalties resulted from the use of preprocessing facilities, and thereby reducing program understanding burden. Instead of translating source code through these phases as described above, JBPAS customizes the lexer to incorporate line splicing, tokenization, preprocessing, and string concatenation into the lexical analysis. The lexer tokenizes the source code to pass on to the parser, and invokes the preprocessor only when it encounters preprocessing directives.

To incorporate line splicing, tokenization, preprocessing, and string concatenation into the lexer has, at least, following advantages:

- ***Increase speed and save space:*** As source code passes through only one phase, the parsing speed is obviously improved. Moreover, JBPAS does not need to store the intermediate result of the phases of line splicing, tokenization, preprocessing, and string concatenation; as a result, it effectively saves space, which is major concern in code analysis.
- ***Keep comments information:*** A comment is text that the typical compiler ignores and can treat as white space. However, as code annotation, comments are important and necessary to program understanding, especially for those self-documentation programs with comments. Without replacing comments with white space, the lexer receives source code with comments, and therefore can extract comment information accurately.
- ***Extract include hierarchies:*** When the preprocessor inserts include file into the current parsing file, the lexer can keep track of processing include files, which is useful to analyze include hierarchies, i.e. the include files and their interdependencies.
- ***Physically locate:*** It is important for a program analysis tool to associate program entities, such as class, object, statement, macro, and so on, with their physical locations in the source code. The lexer can easily locates tokens and macros, for it gets them from the physical source code without previous line splicing and macro expansions.

For these above reasons, it is difficult, if not impossible, for code analyzers to use ready-made preprocessors. Therefore we construct our own preprocessor. Algorithm of file inclusion and conditional compilation is relatively simple, so will not be discussed here. In the following, we will focus on macro processing, the most difficult part of C++ preprocessing.

3.3 Algorithm of macro processing

Like GHINSU's GPP[15], JBPAS employs a dictionary to map macro name to the required replacement information: the file name and location where the macro is defined, and a replacement table. Replacement table contains macro body information, which may be divided into definition substrings by the formal parameters, if any, and records the formal parameter's number and position, which are important and necessary to replace formal parameters with actual arguments.

Figure 4 shows the data structures used for macros:

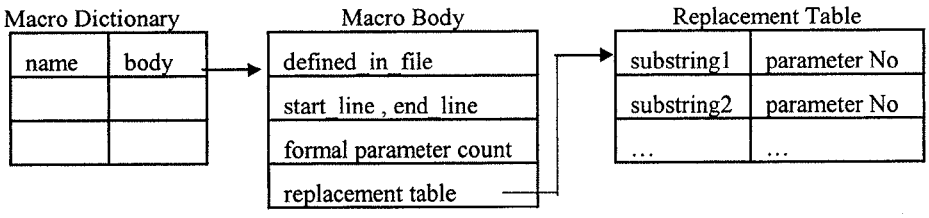


Figure 4. Data Structure for Macros

When the ***#define*** directive is encountered, the lexer regards the following identifier as macro name, and creates a formal parameter list, if the macro has any parameters. The macro body process is straight forward: identifiers in the definition, if matched with parameters in the parameter list, are extracted as formal parameter, whose number is add to the replacement table, and the remainder of macro definition is divided into definition substrings and are stored in the replacement table.

For code segment as shown in figure 3, after macro definition, the macro dictionary will look like figure 5:

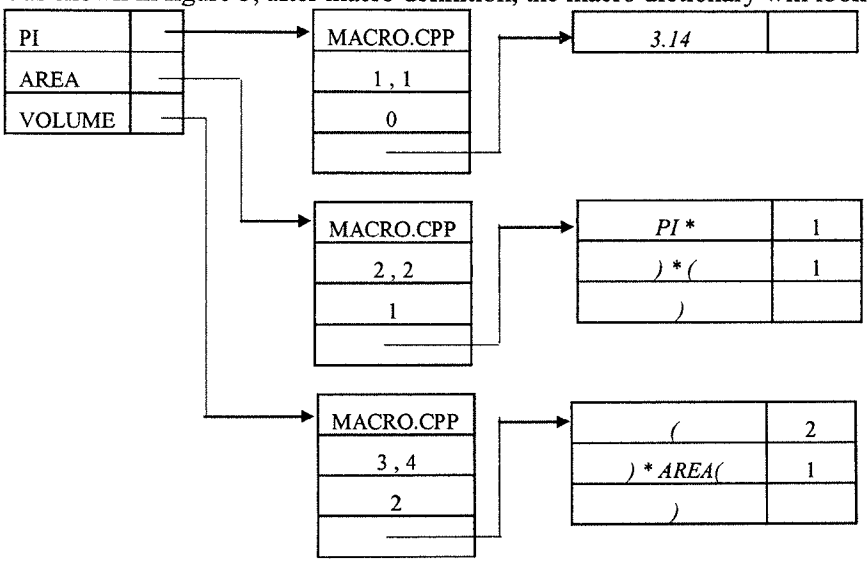


Figure 5. Macro Dictionary in MACRO.CPP

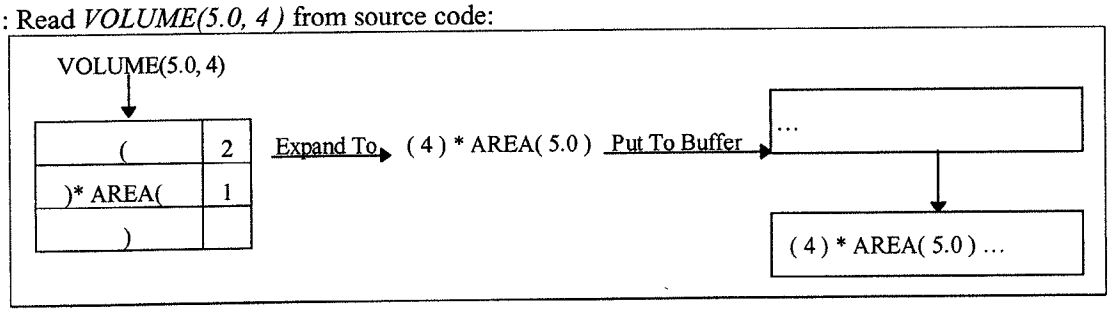
Whenever the name of macro is recognized by looking up the macro dictionary, macro expansion value must be calculated and substituted for the macro and its parameters, if any. Macro expansion and substitution are more complex than definition process, especially when met nested macro substitution as shown in MACRO.CPP. Some preprocessors, say, GPP[15], usually calculates macro's substitution value after complete expansion, i.e. all embedded macros are expanded and substituted. In this way, it needs to store result structure. The preprocessor of JBPAS uses a different method to replace macro name with its definition string. The definition string is just the same as the macro body expect substituting formal parameters with corresponding actual argument strings, and the embedded macros, if any, are expanded only when they are currently analyzed. This approach is more like the means by which programmers understand macro source code manually.

The lexer of JBPAS has a character buffer usually used for those ungeted characters in lexical analysis, and reads character from the buffer when it is not empty. When macro expansion, the lexer gets the macro replacement string from the preprocessor and puts it into the buffer. Then the lexer treats the macro replacement string in the buffer normally as source code text, the only difference is that the lexer dose not move the read pointer of current file, which ensures accurate physical location.

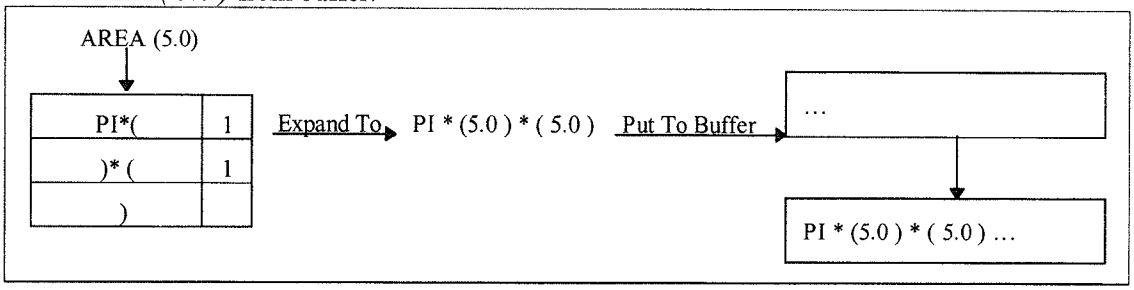
The expansion of macro without argument is relative simple, the definition substring in the replacement table is just the replacement value. If the macro accepts arguments, the actual arguments following the macro name are identified and marked in order. Using the information in the macro dictionary, the formal parameter number are replaced with corresponding actual argument strings, and then definition substrings in the replacement table are concatenated with those inserted actual argument strings and returned as a whole of replacement value.

The steps in the expansion and substitution of *VOLUME(5.0, 4)* proceeds as shown in figure 6:

Step 1: Read *VOLUME(5.0, 4)* from source code:



Step 2: Read *AREA (5.0)* from buffer:



Step 3: Read *PI* from buffer:

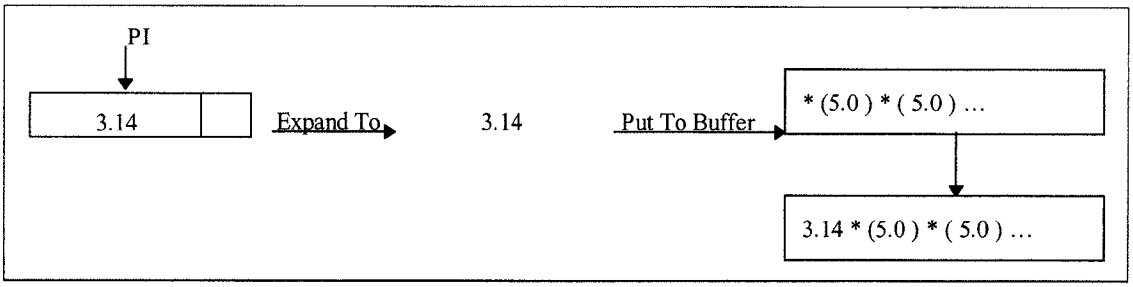


Figure 6. Steps of Macro Expansion and Substitution

4. Parsing C++

We employ the standard compiler tool YACC to generate the parser of the C++ front end. YACC enables us to easily update the front end as the target language changes. The YACC grammar used in JBPAS conforms to the draft proposed ANSI-C++ standard [16] and is enhanced to support Microsoft Visual C++ 4.0[14], which, in turn, is used to implemented the JBPAS system.

4.1 Ambiguity of C++ grammar

C++ grammar, especially those for declarations, is inherently ambiguous and definitely not LALR, therefore, is not directly suitable for YACC. The following statement will illustrate the ambiguity of C++ declaration grammars:

int f (...);

In the above example, there is an ambiguity involving function prototype and constructed declarator with parenthesized initializers. Generally a declarator that looks like: ***int f (...)*** is presumed to be a function prototype, and YACC will first reduce ***f (...)*** and then reduce with *int* . However, with constructed declarators, say, the above statement is actually ***int f (0);*** , it is official to first reduce ***int f*** and then reduce with the initializer, ***(0)*** . Unfortunately, by the time YACC realizes it is initializer, not function arguments, YACC has pushed the separate declarator tokens ***int*** and ***f*** into the YACC stack without combining them, and then YACC will report a syntax error.

4.2 Lookahead technique to disambiguate C++ parser

Typically there are two ways to disambiguate the parser generated by YACC. The first is to modify the ambiguous grammar to an equivalent unambiguous grammar[17]. Another practical approach is token lookahead technique, which inserts a separate lookaheader between the lexer and the parser. The lookaheader processes tokens got from the lexer before passing them on to the parser, thereby indicating the context to help disambiguate the parser.

As C++ language is inherently ambiguous, it is a costly and time-consuming work to modify it to an unambiguous grammar suitable to YACC. Even successful, say, the work of James A. Roskind[18], the equivalent grammar might be more complex and less readable, therefore, inflexible to update when the language changes. To keep YACC grammar as clean and simple as possible to that provided in [15], JBPAS adopts lookahead technique similar to that employed by cfront and CPPP[19].

The lookaheader is independent of the lexer and the parser, and maintains a buffer of tokens. Instead of directly passing the output of lexer on to the parser, the lookaheader uses recursive descent parsing technique to scan tokens in the token buffer. Whenever met syntax difficult to parse, the lookaheader handles tokens according to the lexer state. It may substitute some tokens in the buffer with different ones, or inserts a special token into the token buffer to indicate the context, thereby disambiguating the parser. Enlightened by CPPP[19], JBPAS employs the lookahead states translation, as shown in figure 7:

State and Tokens	*Action (Modified or Inserted Tokens)*
class_key T_IDENTIFIER :	I_CLASS_SPEC class_key T_IDENTIFIER :
class_key T_IDENTIFIER {	I_CLASS_SPEC class_key T_IDENTIFIER {
class_key T_CLASSNAME :	I_CLASS_SPEC class_key T_CLASSNAME :
class_key T_CLASSNAME {	I_CLASS_SPEC class_key T_CLASSNAME {
T_CLASSNAME (...) Where T_CLASSNAME is current scope name	T_CONSTRUCTOR (...)
T_TEMPLATENAME (...) Where T_TEMPLATENAME is current scope name	T_CONSTRUCTOR (...)
T_CLASSNAME :: T_ENUMNAME	I_QUAL_TYPE T_CLASSNAME :: T_ENUMNAME
T_CLASSNAME :: T_IDENTIFIER	I_QUAL_ID T_CLASSNAME :: T_IDENTIFIER
T_CLASSNAME :: *	I_QUAL_PTR T_CLASSNAME :: *
T_OPERATOR T_NEW	T_OPERATOR M_NEW
T_IDENTIFIER (initializer)	T_IDENTIFIER M_INIT_LEFTP initializer)

Figure 7. Lookahead States Translation

The lookaheader modifies or inserts tokens in the token buffer. All of the modified tokens begin with an " ***I_*** " prefix, and the inserted ones begin with an " ***M_*** " prefix, both are opposed to the raw tokens received from the lexer, whose name begin with an " ***T_*** " prefix. Following are some examples of modified or inserted tokens and their purposes:

- **I_CLASS_SPEC**: Inserted before class-key (***class***, ***struct***, or ***union***) to distinguish definition uses of the class-key from elaborated type names.
- **I_QUAL_TYPE**: Inserted before a sequence of name qualifiers which refer to an enumeration type name, for example, T_CLASSNAME::T_ENUMNAME.
- **I_QUAL_ID**: Inserted before a sequence of name qualifiers which refer to an identifier, for example, T_CLASSNAME::T_IDENTIFIER.
- **I_QUAL_PTR**: Inserted before a sequence of name qualifiers which refer to a pointer, for example, T_CLASSNAME::*.
- **M_NEW**: Changed from the token T_NEW when the keyword ***new*** is preceded by the keyword ***operator***.
- **M_INIT_LEFTP**: Changed from the token '(' which is used at the start of an initializer in a declaration.

Lookahead technique enables the parser generated by YACC to disambiguate the target grammar and parse those difficult syntactic constructs. The above statement, for example, ***int f (0);*** becomes ***int f M_INIT_LEFTP 0);*** after lookahead pass, and the parser will definitely consider it as a constructed declarator with parenthesized initializer.

5. Current Status and Future Work

We have implemented the C++ front end, the program understanding system, the reverse design document generator, and the prototype version of the component extractor. The front end and the program understanding system are built through re-engineering of the BDCom-C++ system with the help of BDCom-C++ itself. These implemented analysis tools run on PC under Windows 95 platform, and are currently being widely used at CASE Lab, Peking University to analyzing C++ programs and restructure existing software.

Our future plans includes the following:

1. Enhance the C++ front end to support other C++ languages, such as Borland C++.
2. Implement other analysis tools of JBPAS cited in section 2, encapsulate all JBPAS tools to OLE applications, and integrate them in an integrated environment.
3. Reuse techniques and code from the C++ front end to construct front end for other object-oriented language, for example, Smalltalk and JAVA.

6. Acknowledgements

This work is funded by the National 9th Five-Year Plan and the National Hi-Tech 863 Program, with equipment donated by Intel Co. Xiangkui Chen and Tao Xie participated into development work. Many thanks to Yih-Farn Chen of AT&T Bell Lab and Steve P. Reiss of Brown University. Finally, special thanks are due the anonymous referees of ACM SIGPLAN Notices.

References

[1] A. Von Mayrhauser, K. Srchie, and N. Weber, *"Incremental Parsing for Software Maintenance Tools"*, J. System Software 1993; 23; 235-243.

[2] Raul Medina-Mora, Peter H. Feiler, *"An Incremental Programming Environment"*, IEEE Trans. On SE, Vol. SE-7, No.5, Sep. 1981, 472-481.

[3] T. Teitelbaum and T. Reps, *"The Cornell Program Synthesizer: A Syntax-directed Programming Environment"*, Dept. of CS, Cornell Univ. Ithaca, NY. Tech. Rep. TR80-421, May 1980.

[4] Mei Hong, Yuan Wanghong, Wu Qiong, and Yang Fuqing, *"BDCom-C++□A C++ Program Understanding System"*□Chinese Journal of Electronics, Vol.6, No.2, April 1997, 64-69.

[5] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy, *"The C Information Abstraction System"*, IEEE Trans. On SE, Vol. 16, No. 3, March 1990, 325-334.

[6] J. E. Grass and Y. F. Chen, *"The C++ Information Abstractor"*, USENIX C++ Conf. Proc., 1990, 265-277.

[7] M. E. Lesk, *LEX—a Lexical Analyzer Generator*, CSTR 39, AT&T Bell Laboratories, Murray Hill, N. J. 1975.

[8] S. C. Johnson, *YACC—Yet Another Compiler Compiler*, CSTR 32, AT&T Bell Laboratories, Murray Hill, N. J. 1974.

[9] Peter Coad and Edward Yourdon, *Object-Oriented Design*, Yourdon Press, 1991.

[10] Brian Henderson-Sellers, *Object-Oriented Metrics—Measures of Complexity*, Prentice Hall PTR, 1996.

[11] Grady, R.B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[12] Hafedh Mili, Fatma Mili, and Ali Mili, *"Reusing Software: Issues and Research Directions"*, IEEE trans. On SE, Vol. 21, No. 6, June 1995, 528-562.

[13] Kraig Brockschmidt, *Inside OLE 2*, Microsoft Press, 1994.

[14] —, *Microsoft Visual C++ Version 4.0 Reference Volume*, Microsoft Corporation, 1994.

[15] Panos E. Livadas and David T. Small, *"Understanding Code Containing Preprocessor Constructs"*, Proc. IEEE 3rd Workshop on Program Comprehension, IEEE Computer Society Press, November, 1994, 89-97.

[16] —, *Working paper for Draft Proposed International Standard for Information Systems—Programming Language C++*, AT&T Bell Laboratories, April 1995.

[17] Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[18] James A. Roskind, *C++ Grammar for YACC*, Independent Consultant, Indialantic FL, Portions Copyright(c) 1989, 1990.

[19] Steven P. Reiss and Tony Davis, *"Experiences Writing Object-Oriented Compiler Front Ends"*, working paper, Dept. of CS, Brown Univ. Jan. 1995.