An Object-Oriented Approach to C++ Compiler Technology

Cristian Sminchisescu[1], Alexandru Telea[2]
[1] Department of Computer Science, Rutgers University, USA
[2] Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands

Abstract

This paper focuses on the use of object-oriented approaches to syntactical and semantical analysis for complex object-oriented languages like C++. We are interested in these issues both from a design and implementation point of view. We implement a semantic analyzer in an object-oriented manner, using the C++ programming language. We base our approach on design patterns in order to improve the flexibility of the implementation. The purpose of this research is twofold. First, we are interested whether the object-oriented approach to compiler technology (as a possible alternative to attribute grammars) really produces more modular, concise and reusable code in terms of building blocks or control structures for generating analyzers for a possibly larger class of object-oriented languages. Second, we are interested in the design of the analyzer internal structure to support further development and research in the area of high-level optimizations related to incremental compilation techniques.

1 Introduction

Traditionally, a compiler or translator front-end comprises several steps like lexical and syntactical analysis, semantic analysis and high-level code optimization (for the time being, we don't consider a separate code optimizer stage).

Although numerous textbooks exist in the area of compiler design, formal languages and parsing techniques ([1], [13], [14]), there is hardly any detailed description regarding the design and implementation of a complete language processor for a complex language like C++.

In particular, the C++ language is no longer based on a context-free grammar. Furthermore, the language is not only ambiguous, but inherently ambiguous, implying that no direct grammar transformation (i.e. one not introducing contextual tokens) exist for transforming it into a non-ambiguous one. Consequently, any traditional lexical and syntactical analysis combination will not be effective in such a context, at least up to the point in which it can be designed and implemented in a modular, independent fashion.

In order to address the above problems, we introduce a new, separate stage between the usual lexical (Lex) and syntactical (Yacc) analysis stages. This stage, called LALEX (lookahead LEX) will take over the context dependency present in a language like C++ by doing special processing and introducing contextual tokens to disambiguate the input stream. The advantage of such an approach is manifold: it allows us to use existing tools like Lex and Yacc and keep their design and semantics simple and decoupled, but also minimizes the processing pipeline design and maintenance costs, and make it simple to understand. The processing flow can be depicted in Fig. 1.

The semantic analysis stages, although not emphasizing external fragmentation as the language processing part described above, are subject, however, to internal differentiation. It is inside this stage where the object-oriented techniques prove to be extremely useful and elegant. Internally, this stage can be further differentiated into name analysis (binding each use of a name to its declaration making thus available context and type information) and type analysis dealing with type synthesis for expressions. One should notice the particular complexity and pressure on the semantic analysis stage in the case of a language with such rich semantic structure like C++.

2 Internal Representation. Concrete and Abstract Syntax

The syntax of a programming language deals with the set of its legal programs and with relations between

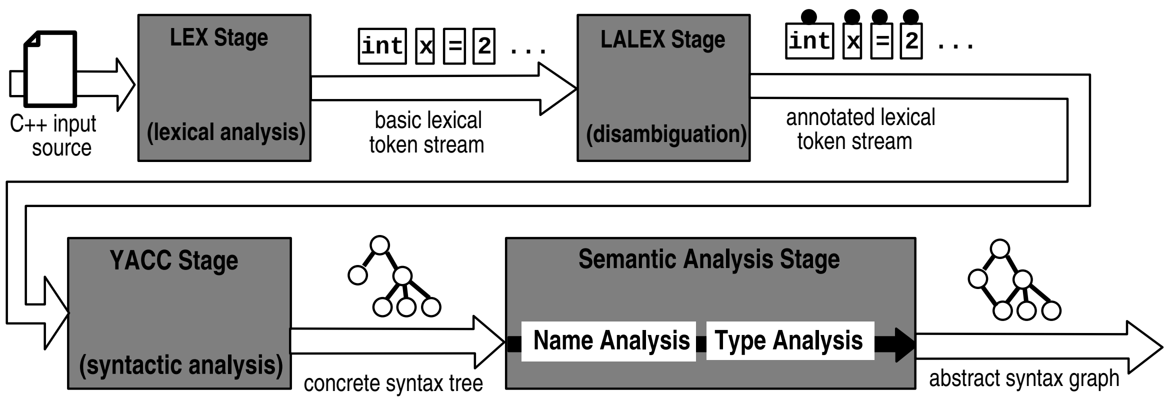

Figure 1: C++ language analysis pipeline with the LALEX stage

the symbols and the phrases occurring in them ([10]). A necessary distinction of syntax can be made into concrete and abstract syntax.

Concrete syntax involves analysis, that is the recognition of legal programs from texts and their unambiguous parsing into phrases (traditionally, this means concrete syntax can be further divided into lexical analysis and syntactical analysis).

Abstract syntax deals only with the compositional structure of phrases from programs, ignoring how that structure might have been determined or ignoring those details of the syntax tree that have no semantic significance. Therefore, abstract syntax provides a better interface between the syntax and semantics.

The internal representation of a program in the C++ language in our implementation is based on abstract syntax graphs. We use an object-oriented approach to abstract syntax. The abstract syntax graphs consist of leaf nodes representing terminals in the language as well as nonterminal nodes representing syntactic constructs. These constructs are modeled by means of a type hierarchy with a common root node. This hierarchy directly maps the non-terminal symbols in the grammar into node classes. Consequently, we construct node classes representing declarations (declaration specifiers and declarators), expressions, statements, actually all syntactic constructs in the language.

Additionally to the nodes mapping directly to the syntactic constructs, we also introduce nodes modeling semantic constructs (the ones corresponding to types or blocks) as well as nodes making the connection between the syntax and semantics. The latter we called "doors" ([7]) since they make the transition between fully syntactic and fully semantic entities. They are, in fact, symbol table entities. It is the introduction of these latter semantic nodes that make our internal representation a graph rather than a tree, since, for example, during semantic analysis, each name application (use) is bind to its name declaration, making thus the type and the context of a declared name available to the point where it is used.

The advantage of a type hierarchy with a common root is manifold: it allows us to write the semantic rules for each type of node as a set of methods specific to that particular node and provide a single control mechanism, to be applied on the entire hierarchy. This semantic definition mechanism is actually a procedural form of an attribute specification. The structure of the hierarchy root node is rather simple. It provides a name, links to its children, the location in the initial source code, and a set of values used for signaling syntax or semantic errors or employed during tree traversals and evaluation.

The result of the language processing stage is an unresolved abstract syntax tree, since it doesn't yet contain symbol and type synthesis information. After semantic analysis phase has been triggered, we obtain a resolved abstract syntax graph where name analysis and type analysis were performed, and symbol table information has been generated.

3 C++ Language Processing

Lexical and syntactical analysis for C++ proved to be very complicated processes both from a theoretical and implementation point of view. This is due to the fact that the declaration part of the grammar for the C++ language is inherently ambiguous and certainly, not LALR. This means that a C++ token sequence cannot be parsed by a standard shift/reduce syntactical analyzer, using a fixed (known) number of lookahead tokens. The next section outlines the C++ language ambiguities that make the language processing part difficult.

3.1 Language Ambiguities

Although a detailed discussion of the ambiguities in the grammar is beyond the scope and size of this paper, a short overview of the situations generating ambiguities can be useful. A detailed analysis can be found in ([12]).

A generally used technique for removing ambiguous constructs (illustrated by conflicts in the grammar) is the one of introducing terminals (semantic tokens) to remove the pressure from the parser. In this situation, the lexical analyzer should keep its own scoped symbol table and returns tokens based on contextual information. A technique like this one permits avoiding the identifier versus typedef-name conflict illustrated by the possible interpretations of the following line of code: which could be either a re-declaration of a local

```
f(*a)[5];
```

variable or a function call depending on whether *f* was or not previously declared as a typedef name (note here that this construct is ambiguous even in ANSI C).

However, the above technique doesn't help solving the conflicts between function like casts and declarations where, for instance, a single line of code could be parsed in two ways, both syntactically correct. Which one to choose, then ? The underlying guideline beyond the Reference Manual ([5]) is that if a token stream can be interpreted by an ANSI C compiler in favor of a declaration then a C++ compiler should follow the same interpretation. However, this approach seems to require a backtracking parser, which should try to parse a token stream as a declaration, and, if this fails, retry the parsing as in a statement context.

Another source of conflicts present in the C++ language stems from the mixing of types and expressions. This situation could appear recursively in the left context, so the ambiguity is perpetuated by the token sequence and the obvious counterpart in the parsing machinery is that it is forced to defer reductions indefinitely, making thus impossible to use a parser based on a fixed number of lookahead tokens.

One could notice, again, that types and expressions have been kept separated in the C language where we indeed, encounter a clear separator ('=') between a declarator and its initializing expression. In terms of separation between types and expressions, the following two distinctions are present. First, abstract declarators are allowed, but no analogy is provided in expressions (also, abstract declarators include the possibility of trailing '*' tokens). Second, the binding of elements in a declaration is quite different the one in expressions. Mainly, the declaration-specifiers are bound separately to each declarator in the comma separated list of declarators (e.g. int a, *b, c;). With most of expressions, a comma provides a complete isolation between expressions.

The major violations the policy of keeping declaration and expressions separate in C++ can be found in parenthesized initializers that drive constructors, free-store expressions without parenthesis around the type, conversion function names using arbitrary type specifiers, and function like casts.

One should notice two possible lines of reasoning in conflict disambiguation for the efforts of designing a standard C++.

The first one is to parse tokens in the longest possible declarator, and identify the syntax errors that result. This tendency focuses on a solution within the grammar which should be cleaned of those constructs that make it inherently ambiguous.

The second trend is to preserve all of the existing constructs and use an enhanced lexer using minimal recursive descent parsers to look ahead so that the parser doesn't misparse valid language constructs or induces syntax errors. This is the solution able to fully support the standard of the language at present time.

3.2 Language Processor Implementation

The implementation we present here is following the second trend presented above. At our present knowledge, a similar approach has been chosen by the original cfront and current GNU g++ compiler (although, for instance, in the g++ compiler, the lookahead stage is mixed within the lexical analyzer which makes it more difficult to maintain, scale and understand).

The idea we follow is to introduce a separate stage, LALEX (lookahead Lex) between the lexical and syntactical analyzer. This stage will generally pass tokens received from lexical analysis. However, in some special situation requiring context differentiation it will process tokens for those syntactic constructs which are difficult to parse by either introducing new terminal tokens or change existing ones in order to disambiguate the input stream.

When designing the present implementation, we try to use standard employed tools whenever possible. So, the idea wasn't to hand-code a full syntax analyzer from scratch, but to use Lex and Yacc. However, the intention is to keep them as simple and elegant as possible and we imposed that the grammar specification be close to the one in the Reference Manual ([11], [5]).

In order to correctly parse the declaration part, we need a backtracking, top down parser. Yacc is a non-backtracking bottom up parser. Consequently, we needed a separate stage between Lex and Yacc which will perform top down parsing needed for processing declarations. Moreover, LALEX and Yacc specification have complementary behavior. The modifications made by the LALEX stage concern the processing of declarations, the processing of identifiers and other processing for different difficult to parse constructs.

LALEX tries to parse a declaration whenever a declaration could be present in the input stream. For doing this, it uses a set of recursive descent (top-down) parsers. These top-down parsers model the declaration part of the C++ grammar in a direct way. For example, the rule synthesizing a declaration is something like:

declaration: declaration_specifiers declarator_list

In our implementation, we construct a function *procDeclaration* which will call in turn *procDeclSpec* and *procDeclaratorList*. This functions will in turn call other functions corresponding to the children nonterminals synthesizing them.

It is very important to notice that it is not the task of LALEX to perform a Yacc style analysis of the input stream. What LALEX tries to do is to ensure that a certain syntactic construct is processed by a certain part of the grammar, which is adequate for processing that construct. Therefore, it is very possible for a certain token or construct to successfully pass LALEX and fail to pass Yacc analysis (e.g., LALEX will not check that an array index represents a valid syntactic expression). LALEX's task is to identify where declarators (both normal and abstract) begin in the declaration and to categorize left parenthesis (this signaling is done by inserting terminal tokens). This is sufficient to ensure correct parsing of declarations.

Processing identifiers means differentiating them. The reference manual considers separate identifiers for typedef names, class, struct or enum names, template names and enum type names. All tokens are returned by Lex under the generic *identifier*, but are further differentiated by LALEX which uses its own symbol table.

More specifically, LALEX is able to return separate terminal tokens like *typedef-name*, *enum-name*, *class-name*, *union-name*, *template-name*, by using contextual information in its symbol table. It is worth mentioning here that the symbol table in LALEX has only the above limited application. The real symbol table for the program is actually generated as a step of semantic analysis (name analysis) when declarators are transformed into symbol table entities (doors) and linked appropriately in the abstract syntax tree, making thus context and type information available to them.

The final set of modifications performed by LALEX are related to several constructs which are difficult to parse. They are related to the *new* operator syntax, template syntax, class and function definitions and external linkage. New terminal tokens are inserted here in order to disambiguate the parse. Template processing require special attention, since the code defining a generic type should be kept into a temporary area and submitted to the parser only when a template instantiation process actually takes place as a class or function definition.

In the end of our discussion about C++ parsing, we shall take a brief look at Lex and Yacc specifications. They are relatively simple. Lex specification returns the keywords of the language or other special symbolic characters (i.e. "+=" as *ADDEQ*). All other names will be returned under the generic *identifier*. Also a simple name node will be created to be subsequently inserted in the syntax tree, as the value of the terminal. It is also Lex's task to return literals, that is integers, floating point, character or string constants and also handle comments for both old style C and C++ comments.

The Yacc specification is simple and elegant. Its main strength is that is as close to the reference manual specification of the grammar as possible, being thus very simple to implement and maintain (e.g. when there is a need to introduce new C++ dialect constructs). The actions associated with the grammar productions serve two main purposes: first to maintain the symbol table for managing identifiers, and second, the generation of the abstract syntax tree.

In an attributed grammar, the value associated with any grammar token is the abstract syntax tree constructed for that token. For identifiers, this value is generated as part of lexical analysis as an abstract simple name node. For other nonterminals, the simple rule associated will call it's corresponding constructor which will build a new node and link it with its children. The advantage of this approach is that it realizes a direct mapping from the concrete syntax specified by Yacc rules to the abstract syntax specified by the constructor calls.

4 Semantic Analysis

4.1 Overview

Several approaches to attribute evaluation are available when performing semantic analysis.

In *data-driven evaluation* attributes are represented by memory cells and their values can be read and stored. In order to obtain an attribution, the attribute instances are evaluated in topological order, that is according to a topological sort of the dependency graph. In a pure attribute grammar, a simple evaluation such a simple tree walk can sometimes suffice. The order can be precomputed, based on the grammar, at compiler construction time.

The *demand-driven evaluation* is the alternative evaluation strategy. Here, each attribute is associated with its semantic function. The attributes are thus not stored using this technique, since the access to the value of an attribute is implemented by calling its semantic function. In this way all the attributes are automatically available and consistent. Here, one could use dynamic ordering, since attributes are computed when their arguments become available. In a demand-driven evaluator, synthesized attributes can be mapped directly to virtual functions. Inherited attributes can be implemented by virtual functions in the father node, but since the parent node may have many sons of the same class, an extra parameter is needed in this function to let the parent node decide which equation to apply (an alternative technique to supporting multiple semantic functions could be the use of behavioral patterns).

The design we follow here is a compromise. We impose a procedural order of evaluation (depth first) for computing basic semantic information and the computed values were stored directly as attributes (depending on whether are used only for semantic evaluation or are part of the final resolved abstract syntax graph, they could be either private or public). Furthermore, in some specific situations as in case of declaration or expression processing, additional passes are needed over the abstract syntax tree for complete resolution and detailed analysis.

The semantic processing will be initialized by a call to the function *semanticProcess* on the root of the abstract syntax tree. This function will process a current semantic state, equivalent to the inherited attributes in an attribute grammar. The result after applying this function will be a resolved abstract syntax tree containing new semantic nodes representing types, blocks and doors (symbol table entities). Blocks nodes are made available to the nodes where they begin and end, the doors nodes replace the corresponding declarators and type nodes are linked to both doors and expression nodes. The syntax trees representing expressions are changed to reflect the language semantics. The tentative nodes are processed. For example, when the parser sees the expression:

```
func();
```

it doesn't really know whether it is an invocation of a function represented or pointed to by *func* or the application of an overloaded call operator on a class object *func*. In the second case, the corresponding tree should be reconstructed and replace the call subtree. Other processing involves making casts explicit, adding *this* parameter to member functions or adding conversion as actual function call nodes. Also. the member rewriting rule is applied, meaning that the inline function body is parsed only after the entire class declaration is seen. This will ensure correct name binding. In order to do this, the inline functions defined inside a class definitions are split into a declaration and an inline function definition.

4.2 Evaluation Algorithm

The theoretical roots of the evaluation algorithm we devised can be found in the depth-first order evaluation algorithm given by ([1]) for L-attributed definitions. The algorithm is outlined in Fig 4.2 However, the above algorithm is modified reflecting the object-oriented view of the whole approach as well as the combination of the data-driven and demand-driven evaluation schemes (Fig. 2). In particular, the function *semanticProcess* will be called for any node (with the default parameter "this"), it will call *inheritedProcess* for evaluating and/or propagating the inherited attributes (mainly updating a semantic state), calls itself for the children of the specific node, and then calls

```
procedure dfvisit(n:node)
begin
    for each child m of n, from left to right
    do
       begin
          evaluate inherited attributes of m;
          dfvisit(m)
       end;
    evaluate synthesized attributes of n
end;
```

```
void AstNode::semanticProcess(SemanticState& cs)
{
   ListIterator iter;
   AstNode anode;
   Boolean errorFlag;

   // don't process static nodes
   if (isNotEvaluable()) return;

   cs->currentNode = this;

   inheritedProcess(cs);

   errorFlag = FALSE;
   if (num_sons > 0) {
      for (sons(iter); iter; ++iter)
      {
           anode = (*iter);
           if (!anode->isNotEvaluable())
           {
             anode->semanticProcess(cs);
             errorFlag |= anode->semanticError;
           };
        };
    };
   cs->currenNode = this;

   synthesizeProcess(cs);

   if (errorFlag) makeErrorNode();
};
```

Figure 2: Semantic processing algorithm

synthesizeProcess when returning. This function can be equivalated with the one computing the synthesized attributes in [1] evaluation algorithm, although it does it on a dynamic basis. The code implementing the control structure of the evaluation is given in Fig. 2, with the three major points highlighted (other irrelevant parts from the real code has been removed since it could have only obviated the comprehension without introducing any semantic significance).

The functions *inheritedProcess* and *synthesizeProcess* are node dependent processing functions. As a consequence, in our object-oriented approach, they will be virtual, so dispatched based on node type. The function *semanticProcess* implements only the control structure of the evaluation, and is defined at the root of the node hierarchy. Another fact to be noticed here is that not all nodes are semantically processed, since some of them could be either nodes without semantic significance or nodes representing already processed semantic entities (e.g. types).

In the next two sections we shall give a brief overview on name and type analysis. One should note that although presented separately, they are indeed interleaved processes, mainly realized under the control provided by the function *inheritedProcess*. This is the place where the amount of node dependent code is gathered during declaration processing. Mainly, this function deals with type creation, doors (symbol table entities) creation and their insertion into the symbol table.

4.3 Name Analysis

As mentioned earlier, name analysis is one of the most important steps in semantic analysis. During this stage the binding process will be performed, that is each name application is associated with the corresponding name declaration making thus all the related information available. The process is also known as declaration processing since it applies only on the declaration nodes.

More precisely, the semantic analysis phase actually creating symbol table entities is *synthesizeProcess*. Inside this function, we iterate on each of the declaration specifications and we invoke node dependent (virtually defined) *processDeclarator* function to build a declaration state object. This state will be consequently used with each declarator to build new corresponding doors and enter them into the symbol table. Moreover, each declarator will be replaced with the corresponding door. Additionally, types are also generated within this stage, as will be emphasized in the next section.

4.4 Type Analysis And Matching

The second category of nodes requiring additional processing are the expression nodes. In our context, this implies additional passes over the corresponding subtrees. These passes are needed due to the circular dependencies between attributes. A circular attribute grammar does not necessarily have a solution for each possible syntax tree. A tree with a cycle in its dependency graph can have zero, one or more solutions. The attribute on a cycle can be evaluated iteratively, and if the consecutive computed values converge, a solution is found. In some special situations, it is possible to decide whether a grammar has always such a converging behavior.

In our implementation, additional passes will be triggered within the function *synthesizeProcess*. It is the purpose of this function to take over both type synthesis and type matching against a desired expression type. The information about the context (the semantic state or the inherited attributes in a standard attribute grammar evaluation), the desired son node, and the desired expression type will be passed to node dependent (virtually implemented functions) *resolveType*, that provide type synthesis and matching. In case a match between a generated type and a desired type is not exact, than a cast expression is created and a sequence of possible conversions is tried in an attempt to obtain the desired type. This process is particularly complex in a language like C++, mainly because of the language support for user-defined conversion and overloaded operations.

Once successful type resolution process has been achieved, the function *synthesizeProcess* also performs some constant folding on the resulting expression and replaces the processed son node with the corresponding resolved one. It is worth noticing that as a secondary result of the execution of the function *resolveType*, some checkings are performed and some error messages are issued (i.e. checking access permissions on private or protected class members).

5 Conclusions

The above presented research into object-oriented compiler technology proved to be both interesting and challenging. A variety of techniques related to both class inheritance and object composition have been used in the implementation process. Several design patterns ([6]) like Builder, Bridge (abstract nodes interface for the syntax tree), Command (during semantic analysis to allow multiple semantic functions to be customized by behavior), Visitor (abstract interface to code generation simulating multi-method dispatching), have been employed. They have certainly made the design elegant and extensible and provided a clear separation between the stages of processing and between the specific behavior node functions (although belonging to a similar processing category) involved during each stage.

Further interest is driven towards high-level optimizations, mainly incremental compilation techniques. These techniques advocate for reusing previous results (abstract syntax tree) instead of reconstructing it from scratch. Additional interest is also related to the customization of the internal structure for supporting a certain class of languages (for example object-oriented or functional). In this way, compilers for a class of languages can be easier obtained by reusing an existing internal structure and overriding it appropriately. Thus, the presented work can provide the basis for building and object-oriented framework for designing code analysis and compiling tools for a large class of languages.

References

[1] A. Aho , R. Sethi, J. Ullman, *Compilers Principles Techniques and Tools*, Addison Wesley, 1986

[2] G. Baumgartner, V. Russo, *Signatures: A Language Extension For Improving Type Abstraction and Subtype Polimorphism in C++*, Technical Report, Purdue University, 1995

[3] W. R. Cook, W. L. Hill, P. S. Canning, *Inheritance is not Subtyping*, Proceedings of the 17th Anual ACM Symposium on Principles of Programming Languages, pages 125-135, San Francisco, California, 1990.

[4] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

[5] M. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, 1994.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[7] G. Hedin, *Incremental Semantic Analysis*, Ph.D. Thesis, Lund University, 1992.

[8] S. B. Lipman, *Inside the C++ Object Model*, Addison Wesley Publishing Company, 1996

[9] B. Meyer, *Object-oriented software construction*, Prentice Hall, 1997.

[10] P. D. Moses, *Action Semantics*, Cambridge University Press, 1992.

[11] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1998.

[12] C. Sminchisescu, *An Object-Oriented Approach to Semantic Analysis for C++*, Proceedings of the XVII-th International Conference on Computer Science and Control, Bucharest, May 1997, vol.2, 1997.

[13] R.Wilhelm, D.Maurer, *Compiler Design*, Addison-Wesley, 1995.

[14] N.Wirth, *Compiler Construction*, Addison-Wesley, 1996.