Delegating Compiler Objects

An Object-Oriented Approach to Crafting Compilers

Jan Bosch*

Department of Computer Science and Business Administration,
University of Karlskrona/Ronneby,
S-372 25, Ronneby, Sweden.

E-mail: Jan.Bosch@ide.hk-r.se,
URL: http://www.pt.hk-r.se/~bosch

Abstract. Conventional compilers often are large entities that are highly complex, difficult to maintain and hard to reuse. In this article it is argued that this is due to the inherently *functional* approach to compiler construction. An alternative approach to compiler construction is proposed, based on object-oriented principles, which solves (or at least lessens) the problems of compiler construction. The approach is based on *delegating compiler objects* (DCOs) that provide a structural decomposition of compilers in addition to the conventional functional decomposition. The DCO approach makes use of the *parser delegation* and *lexer delegation* techniques, that provide reuse and modularisation of syntactical, respectively, lexical specifications.

1 Introduction

Traditionally, compiler constructors have taken a functional approach to the process of compiling program text. In its simplest form, the process consists of a lexical analyser, converting program text into a token stream, a parser, converting the token stream into a parse tree and a code generator, converting the parse tree into output code. Both the lexical analyser and the parser are monolithic entities in that only a single instance of each exists in an application.

The monolithic approach to compiler construction is becoming increasingly problematic due to changing requirements on compiler construction techniques. Whereas previously applications were built using one of the few general purpose languages, nowaday often a specialised, application domain languages is used. Examples can be found in the fourth generation development environments, e.g. Paradox[2], and formal specification environments, e.g. SDL. Another example is the use of compilation techniques to obtain structured input from the user as in, e.g. modern phones. In a modern phone exchange, the user can request services by dialing the digits associated with a service. Example services are *follow-me*

* This work has been supported by the Blekinge Forskningsstiftelse.

[2] Paradox is a trademark of Borland International, Inc.

and *tele-conference*. The digits are parsed by a parser and the requested service is activated for the user. The available services in most systems, however, are subject to regular change. A third example are the *extensible* language models. An extensible language can be extended with new constructs and the semantics of existing language constructs can be changed. In this article, an extensible object-oriented language, LAYOM, is used as an example.

The changing requirements described above call for modularisation and reuse of compiler specifications. These new requirements would benefit from means to modularise and reuse compiler specifications as it would reduce the required effort in the construction of compilers.

In this article *delegating compiler objects* (DCOs) are proposed as an approach to compiler construction that supports modularisation and reuse of compiler specifications. The DCO approach to compiler development allows one to recursively decompose a compiler into *structural components*, i.e. nested compilers. The DCO concept provides a structural decomposition of a compiler in addition to the traditional functional compiler decomposition. When using DCOs, an input program text is not compiled by a single compiler, but by a set of cooperating compiler objects. A compiler object can delegate parts of the compilation process to other compiler objects. The advantage of this approach is that reusability and maintainability are increased considerably due to the modular approach.

To evaluate the mechanism, a tool, PHEST, has been implemented that implements the DCO concept. Using this tool, compilers have been constructed that convert LAYOM code into C++ and C code.

The remainder of this article is organised as follows. In the next section, the problems of traditional compiler construction techniques that we identified are described. Section 3 describes the layered object model that will be used as the running example throughout the paper. Section 4 describes the DCO approach to compiler construction, whereas sections 4.2, 4.3 and 4.4 respectively describe the *parser delegation*, *lexer delegation* and *parse graph node object* techniques employed by the DCO approach. Section 5 describes the PHEST tool supporting the techniques discussed in this paper. Section 6 describes related work and the paper is concluded in section 7.

2 The Problems of Conventional Compilers

Traditionally, the compilation process is decomposed towards the different *functions* that convert program code into a description in another language, e.g. lexing, parsing and code generation. We have identified three problems of this approach to compiler construction:

- *Complexity*: A traditional, monolithic compiler tries to deal with the complexity of a compiler application through decomposing the compilation process into a number of subsequent phases. Although this indeed decreases the complexity, this approach is not *scaleable* because a large problem cannot *recursively* be decomposed into smaller components. In order to deal with

large, complex or often changing compilers, the one level decomposition into lexing, parsing, semantic analysis and code generation phases we have experienced to be insufficient.
- *Maintainability*: Although the compilation process is decomposed into multiple phases, each phase itself can be a large and complex entity with many interdependencies. Maintaining the parser, for example, can be a difficult task when the syntax description is large and has many interdependencies between the production rules. In the traditional approaches, the syntax description of the language cannot be decomposed into smaller, independent components.
- *Reuseability*: Although the domain of compilers has a rich theoretical base, building a compiler often means starting from scratch, even when similar applications are available. The notion of reusability has no supporting mechanism in compiler construction. Nevertheless, crafting a compiler generally is a large and expensive undertaking and reuse, when available, would be highly beneficial.

Summarising, the conventional approach to compiler construction results in large, complex modules that result in the aforementioned problems. This is due to the one-level functional decomposition. We consider an object-oriented approach to compiler construction that supports reuse and modularisation of compiler specifications provides a solution the aforementioned problems.

3 Example: Layered Object Model

The layered object model (LAYOM) [6] is an *extensible* object model that extends the conventional object model to improve expressiveness. An object in LAYOM (see figure 1) consists of five major components, i.e. *variables* (nested objects), *methods*, *states*, *categories* and *layers*. The LAYOM object model can be further extended by adding new types of layers or new object model components.

The variables and methods of a LAYOM object are defined as in most object-oriented languages. A state, as defined in LAYOM, is a dimension of the abstract object state [5]. The notion of *abstract object state* provides an a systematic and structured approach to make the *conceptual* state of the object accessible at the interface. A category is used to define a client category, i.e. a distinguishing characteristic of a group of clients that are to be treated similar.

The *layers* encapsulate the object such that messages sent to the object or sent by the object itself have to pass all layers. Each layer can change, delay, redirect or respond to a message or just let is pass. Layers are, among others, used for representing and implementing inter-object relations [4] and design patterns [7].

4 Delegating Compiler Objects

The delegating compiler object (DCO) approach aims at modular, extensible and maintainable implementations of compilers. In section 2 it was concluded that

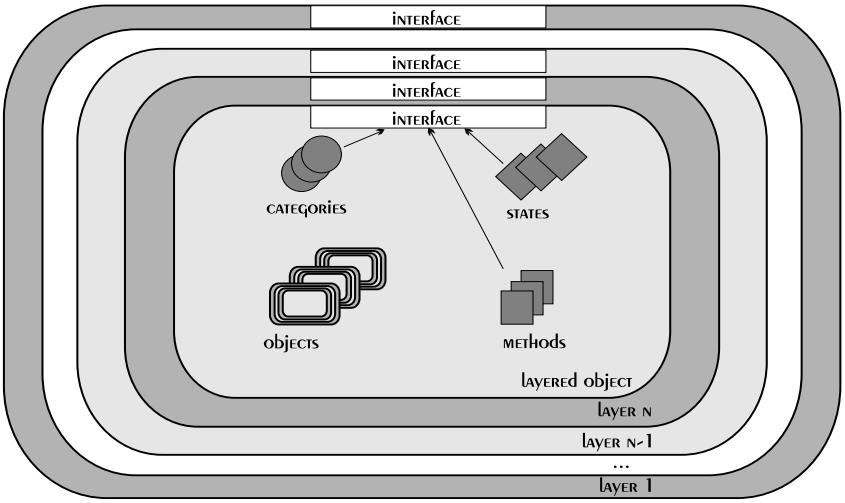

Fig. 1. LAYOM object

traditional approaches to compiler compilation had difficulty providing required features due to the lack of modularisation and reuse. The underlying rationale of DCOs is that next to the functional decomposition into a lexer, parser and code generator, we offer another structural decomposition dimension that can be used to decompose a compiler into a set of subcompilers. Rather than having a single compiler consisting of a lexer, parser and code generator, an input text can be compiled by a group of compiler objects that cooperate to achieve their task. Each compiler object consists of one or more lexers, one or more parsers and a parse graph. A compiler object, when detecting that a particular part of the syntax is to be compiled, can instantiate a new compiler object and delegate the compilation of that particular part to the new compiler object.

The delegating compiler object concept makes use of *parser delegation* and *lexer delegation* for achieving the structural decomposition of grammar, respectively, lexer specification. These techniques will be described in section 4.2 and 4.3. Section 4.4 discusses the parse graph node objects

4.1 Delegating Compiler Objects

A delegating compiler object (DCO) is a generalisation of a conventional compiler in that the conventional compiler is used as a component in the DCO approach. In this approach, a compiler object consists of one or more lexers, one or more parsers and a parse graph.

A consequence of decomposing a compiler into subcompilers is that the syntax of the compiled language should be decomposed into the main constructs of the language. Each construct is then compiled by a DCO. Each DCO can instantiate and interact with other DCOs. The compilation process starts with the

instantiation of an initial compiler object. This DCO generally instantiates other DCOs and delegates parts of the compilation to these DCOs. These DCOs can, in turn, instantiate other compiler objects and delegate the compilation to them.

As an example, the structure of the compiler for LAYOM is shown in figure 2. As mentioned, LAYOM is an extensible language, requiring its compiler to be extensible. In addition, the layer types part of LAYOM have their individual syntax and semantics, but with considerable overlap. The LAYOM compiler is constructed using DCOs since modularisation and reuse of the compiler specification is provided. The initial compiler object of the LAYOM compiler is the *class* DCO. The parser of the class DCO instantiates the other DCOs and delegates control over parts of the compilation process to the instantiated DCOs.

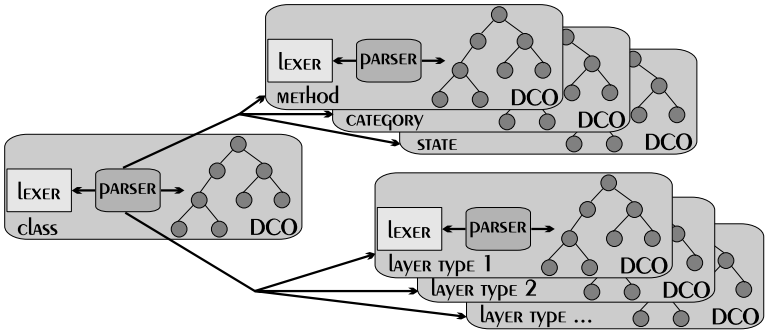

Fig. 2. DCO-based LAYOM compiler

A DCO-based compiler consists, as described, of a set of DCOs that cooperate to compile an input text. Each DCO consists of one or more lexers, one or more parsers and a parse graph, consisting of node objects. When a DCO has multiple parsers or lexers this can be to modularise the parser or lexer specification for the DCO or to reuse existing parser or lexer specifications. The interaction between different parsers or lexers is achieved through *parser delegation* and *lexer delegation*, respectively. In figure 3, an overview of a DCO-based compiler is shown. Each DCO has a parse graph G consisting of node objects. The classes of the node objects are in the *node space* and the DCO specifies which node classes it requires.

The concept of delegating compiler objects makes use of *parser delegation* [3], *lexer delegation* and *parse graph node objects* [6]. These techniques will be discussed in the following sections.

4.2 Parser Delegation

Parser delegation is a mechanism that allows one to modularise and to reuse grammar specifications. In case of modularisation, a parser can instantiate other

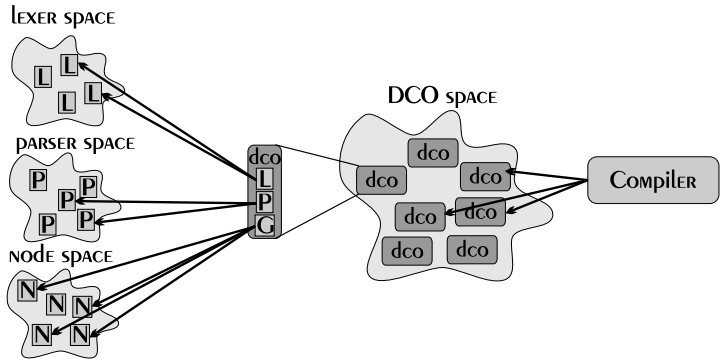

Fig. 3. Overview of a DCO-based Compiler

parsers and redirect the input token stream to the instantiated parser. The instantiated parser will parse the input token stream until it reaches the end of its syntax specification. It will subsequently return to the instantiating parser, which will continue to parse from the point where the subparser stopped. In case of reuse, the designer can specify for a new grammar specification the names of one or more existing grammar specifications. The new grammar is extended with the production rules and the semantic actions of the reused grammar(s), but has the possibility to override and extend reused production rules and actions.

We define a monolithic grammar as $G = (\ I,\ N,\ T,\ P)$, where I is the name of the grammar, N is the set of nonterminals, T is the set of terminals and P is the set of production rules. The set $V = N \cup T$ is the vocabulary of the grammar. Each production rule $p \in P$ is defined as $p = (q, A)$, where q is defined as $q : x \rightarrow \alpha$ where $x \in N$ and $\alpha \in V^*$ and A is the set of semantic actions associated with the production rule q. Different from most YACC-like grammar specifications (e.g. [1, 19]), a grammar in this definition has a name, and the start symbol is simply denoted by the production called *start*.

Parser delegation extends the monolithic grammar specification in several ways to achieve reuse and modularisation. First, one can specify in a grammar that other grammars are reused. In situations where a designer has to define a grammar and a related grammar specification exists, one would like to reuse the existing grammar and extend and redefine parts of it. If a grammar is reused, all the production rules and semantic actions become available to the reusing grammar specification. In our approach, reuse of an existing grammar is achieved by creating an instance of a parser for the *reused* grammar upon instantiation of the parser for the *reusing* grammar. The reusing parser uses the reused parser by *delegating* parts of the parsing process to the reused parser.

When modularising a grammar specification, the grammar specification is divided into a collection of grammar module classes. When a parser object decides to delegate parsing, it creates a new parser object. The active parser object delegates parsing to the new parser object, which will gain control over the input

token stream. The new parser object, now referred to as the *delegated* parser, parses the input token stream until it is finished and subsequently it returns control to the delegating parser object.

Instead of delegating to a different parser, the parser can also delegate control to a new compiler object. A new DCO is instantiated and the active DCO leaves control to the new DCO. The delegated DCO compiles its part of the input syntax. When it is finished it returns control to the delegating compiler object.

To describe the required behaviour, the production rule of a monolithic parser has been replaced with a set of production rule types. These production rule types control the reuse of production rules from reused grammars and the delegation to parser and compiler objects. The following production rule types have been defined:

- $n : v_1\ v_2\ ...\ v_m$, where $n \in N$ and $v_i \in V$
  All productions $n \rightarrow V^*$ from G_{reused} are excluded from the grammar specification and only the productions n from $G_{reusing}$ are included. This is the *overriding* production rule type since it overrides all productions n from the reused grammars.
- $n +: v_1\ v_2\ ...\ v_m$, where $n \in N$ and $v_i \in V$
  The production rule $n \rightarrow v_1\ v_2\ ...\ v_m$, if existing in G_{reused} is replaced by the specified production rule n. The *extending* production rule type facilitates the definition of new alternative right hand sides for a production n.
- n [*id*]: $v_1\ v_2\ ...\ v_m$, where $n \in N$ and $v_i \in V$
  The element *id* must contain the name of a parser class which will be instantiated and parsing will be delegated to this new parser. When the delegated parser is finished parsing, it returns control to the delegating parser. The results of the delegated parser are stored in the parse graph. When $\$i$ is used as an identifier, v_i must have a valid parser class name as its value. The *delegating* production rule type initiates delegation to another parser object.
- n [[*id*]]: $v_1\ v_2\ \ldots\ v_n$, where $n \in N$, the set of all non terminals and $v_i \in V$, the vocabulary of the grammar.
  The element *id* must contain the name of a delegating compiler object type which will be instantiated and the process of compilation will be delegated to this new compiler object. When the delegated compiler object is finished compiling its part of the program, it returns the control over the compilation process to the originating compiler object. The originating compiler object receives, as a result, a reference to the delegated compiler object which contains the resulting parse graph. The delegating parser stores the reference to the delegated DCO in the parse graph using a DCO-node. Next to using an explicit name for the *id*, one can also use $\$i$ as an identifier, in which case v_i must have a valid compiler object class name as its value. The DCO production rule type causes the delegation of the compilation process to another DCO.

In figure 4 the process of parser delegation for modularising purposes is illustrated. In (1) a delegating production rule is executed. This results (2) in the

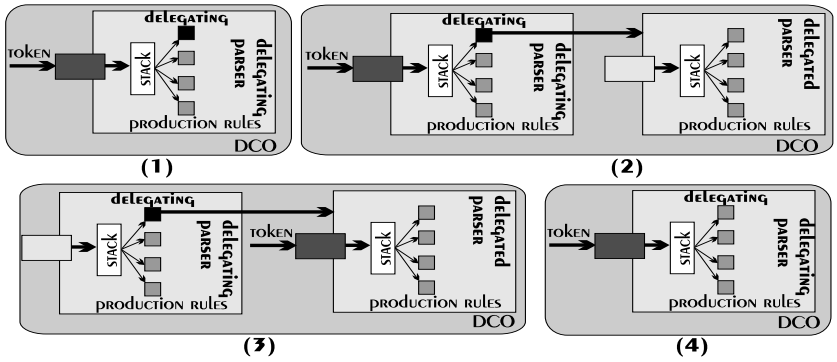

Fig. 4. Parser Delegation for Grammar Modularisation

instantiation of a new, dedicated parser object. In (3) the control over the input token stream has been delegated to the new parser, which parses its section of the token stream. In (4) the new parser has finished parsing and it has returned the control to the originating parser. This parser stores a reference to the dedicated parser as it contains the parsing results. Note that the lexer and parse graph are not shown for space reasons.

We refer to [3, 6] for more detailed discussion of parser delegation.

4.3 Lexer Delegation

The *lexer delegation* concept provides support for *modularisation* and *reuse* of lexical analysis specifications. Especially in domains where applications change regularly and new applications are often defined modularisation and reuse are very important features. Lexer delegation can be seen as an *object-oriented* approach to lexical analysis.

A monolithic lexer can be defined as $L = (I, D, R, S)$, where I is the identifier of the lexer specification, D is the set of definitions, R is the set of rules and S is the set of programmer subroutines. Each definition $d \in D$ is defined as $d = (n, t)$, where n is a name, $n \in N$, the set of all identifiers, and t is a translation, $t \in T$, the set of all translations. Each rule $r \in R$ is defined as $r = (p, a)$, where p is a regular expression, $p \in P$, the set of all regular expressions, and a is an action, $a \in A$, the set of all actions. Each subroutine $s \in S$ is a routine in the output language which will be incorporated in the lexer generated by the lexer generator. Different from most lexical analysis specifications languages, a lexer specification in our definition has a *identifier* which will be used in later sections to refer to different lexical specifications.

Lexer delegation, analogous to parser delegation, extends the monolithic lexer specification to achieve modularisation and reuse. The designer, when defining a new lexer specification, can specify the lexer specifications that should be reused by the new lexer. When a lexer specification is reused, all definitions,

rules and subroutines from the reused lexer specification become available at the reusing lexer specification. In a lexical specification, the designer is able to exclude or override *definitions*, *rules* and *subroutines*. Overriding a reused definition $d = (n, t)$ is simply done by providing a definition for d in the reusing lexer definition. One can, however, also *extend* the translation t for d by adding a $=+$ behind the name n of d. Extending a definition is represented as:

`n =+` $t_{extended}$

The result of extending this definition is the following:

`n =` $t_{extended}$ `?` t_{reused}

A reused rule $r = (p, a)$ can also be overridden by defining a rule $r' = (p, a')$, i.e. a rule with the same regular expression p. One can interpret extending a rule in two ways. The first way is to interpret it as extending the action associated with the rule. The second way is to extend the regular expression associated with an action. Both types of rule extensions are supported by *lexer delegation*. Extending the regular expression p is represented as follows:

p' `|+` p

When a lexer specification is modularised, it is decomposed into smaller modules that contain parts of the lexer specification. One of the modules is the *initial lexer* which is instantiated at the start of the lexing process. The extensions for lexer modularisation consist of two new actions that can be used in the action part of rules in the lexer specifications. Lexer delegation occurs in the action part of the lexing rules. The semantics of these actions are the following:

- **Delegate(<lexer-class>)**: This action is part of the action part of a lexing rule and is generally followed by a *return(<token>)* statement. The delegate action instantiates a new lexer object of class *<lexer-class>* and installs the lexer object such that any following token requests are delegated to the new lexer object. The delegate action is now finished and the next action in the action block is executed.
- **Undelegate**: The undelegate action is also contained in the action part of a lexing rule. The undelegate action, as the name implies, does the opposite of the delegate action. It changes the delegating lexer object such that the next token request is handled by the delegating lexer object and delegation is terminated. The lexer object does not contain any state that needs to be stored for future reference, so the object is simply removed after finishing the action block.

For a more detailed discussion of lexer delegation we refer to [6].

4.4 Parse Graph Node Objects

In the delegating compiler object approach, an object-oriented, rather than a functional approach, is taken to parse tree and code generation. Instead of using passive data structures as the nodes in the parse tree as was done in the conventional approach, the DCO approach uses *objects* as nodes. A node object is instantiated by a production rule of the parser. Upon instantiation, the node object also receives a number of arguments which it uses to initialise itself. Another difference from traditional approaches is that, rather than having an separate code generation function using the parse tree as data, the node objects themselves contains knowledge for generating the output code associated with their semantics.

A parse graph node object, or simply node object, contains three parts of functionality. The first is the *constructor* method, which instantiates and initialises a new instance of the node object class. The constructor method is used by the production rules of the parser to create new nodes in the parse graph. The second part is the *code generation* method, which is invoked during the generation of output code. The third part consists of a set of methods that are used to access the state of the node object, e.g. the name of an identifier or a reference to another node object.

The grammar has facilities for parse graph node instantiation. An example production rule could be the following:

```
method   :   name '(' arguments ')' 'begin' temps statements 'end'
             [ MethodNode($1, $3, $6, $7) ]
         ;
```

The parse graph, generally, consists of a large number of node objects. There is a root object that represents the point of access to the parse graph. When the compiler decides to generate code from the parse graph, it sends a `generateCode` message to the root node object. The root node object will generate some code and subsequently invoke its children parse nodes with a `generateCode` message. The children parse nodes will generate their code and invoke all their children.

5 DCO Tool Set

In order to be able to experiment with the concepts described in this paper, an integrated tool, PHEST, has been developed. The PHEST tool provides the functionality of DCO approach. It incorporates two previously developed tools, i.e. D-YACC and D-LEX, that implement parser delegation and lexer delegation. Another tool part of PHEST supports the definition of parse graph node classes. The PHEST tool itself facilitates the definition of a compiler by composing a cooperating set of DCOs that, when combined, provide the required functionality. A second aspect of the PHEST tool is the composition of compiler objects based on the available lexers, parsers and parse graph node classes.

In figure 5, the user interface of the PHEST tool is shown. A designer using the tool can have several projects, representing different compilers. One project

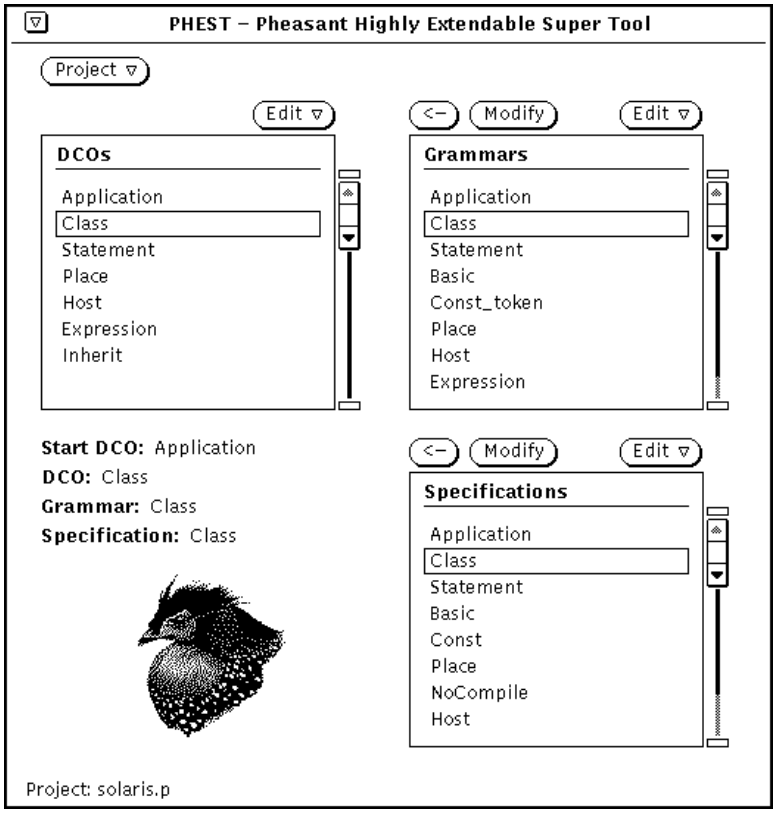

Fig. 5. DCO tool

can be open and worked on using the tool. In the left subwindow, the list of DCOs contained in the current project (or compiler) is shown. The `class` DCO is selected and below the aforementioned window, information on the name of the grammar and lexer used in by the `class` DCO. In this case, the grammar and the lexer also have the name `class`. In the upper right window, the list of grammars contained in the project is shown. Note that the number of grammars is larger than the number of DCOs. The reason for that is that some grammars, e.g. `basic`, are used to as 'abstract classes', i.e. only used for reuse, but never 'instantiated' in a DCO. The lower right window shows the lexers defined within the project.

When satisfied with the configuration, the designer can request the PHEST tool to generate an executable compiler. For pragmatic reasons, PHEST makes use of YACC for the actual parser generation. This is done by first converting a grammar expressed in D-YACC, the extended grammar definition syntax for parser delegation, to an equivalent YACC specification. This specification is converted to C++ by YACC. The resulting C++ code is preprocessed and subsequently

added to the C++ code for the executable compiler. The lexer specifications are treated in an analogous fashion. For the implementation details of *phest* tool, we refer to [6, 18].

The PHEST tool has been used to construct two compilers for the LAYOM object model discussed in section 3. One compiler generates C++ output code for the Sun Solaris environment. The second compiler generates *Neuron* C output code for the Lonworks environment. The students that built the compilers noted that the modularisation and reusability provided by the DCO approach indeed simplified compiler development.

6 Related work

In [10, 11] a different approach to language engineering, TaLE, is presented. Rather than using a meta-language like LEX or YACC for specifying a language, the user edits the classes that make up the implementation using a specialised editor. TaLE not immediately intended for the implementation of traditional programming languages, but primarily for the implementation of languages that have more dynamic characteristics, like application-oriented languages. *Reuse* is one of the key requirements in TaLE and it is supported in three ways: first, language structures are implemented by independent classes, leading to a distributed implementation model; second, general language concepts can be specialised for particular languages; third, the system supports a library of standard language components.

The TaLE approach is different from the delegating compiler object (DCO) approach in, at least, two aspects. First, TaLE does not make use of metalanguages like LEX and YACC, whereas the DCO approach took these metalanguages as a basis and extended on them. This property makes it more difficult to compare the two approaches. Second, the classes in TaLE used for language implementation seem only to be used for language parts at the level of individual production rules, whereas DCOs are particularly intended for, possibly small, groups of production rules representing a major concept in the language.

In [2] a mechanism for reuse of grammar specifications, *grammar inheritance* is described. It allows a grammar to inherit production rules from one or more predefined grammars. Inherited production rules can be overridden in the inheriting grammar, but exclusion of rules is not supported. Although inheritance offers a mechanism to reuse existing grammar specifications, no support for modularising a grammar specification is offered. Therefore, for purposes of modularising a large grammar specification, we are convinced that delegation is a better mechanism than inheritance. The rational for this is that delegation allows one to separate a grammar specification at the object level, whereas inheritance would still require the definition of a monolithic parser, although being composed of inherited grammar specifications. Also, delegation offers a *uniform* mechanism for both reuse *and* modularisation of grammar specifications.

In [8], a persistent system for compiler construction is proposed. The approach is to define a compiler as a collection of modules with various func-

tionalities that can be combined in several ways to form a compiler family. The modules have a type description which is used to determine whether components can be combined.

The approach proposed in [8] is different from the DCO approach in the following aspects. First, although the approach enhances the traditional compiler modularisation, modularisation and reuse of individual modules, e.g. the grammar specification, is not supported. Secondly, judging from the paper, it does not seem feasible to have multiple compilers cooperating on a single input specification, as in the DCO approach.

The Mjølner Orm system [13, 14, 15] is an approach to object-oriented compiler development that is purely grammar-driven. Different from the traditional grammar-driven systems that generate a language compiler from the grammar, Orm uses *grammar interpretation*. The advantage of the interpretive approach is that changes to the grammar immediately are incorporated in the language. A grammar in Orm is represented as an object and consists of four parts: an *abstract grammar* defining the structure of the language; a *concrete grammar* that defines the textual presentation of the language constructs; a *semantic grammar* that defines the static semantics and a *code-generation grammar* that translates the language into an intermediate language. Orm may be used to implement an existing language or for language prototyping, e.g. for application-domain specific languages.

Although the researchers behind the Orm system do recognise the importance of grammar and code reuse, see e.g. [16], this is deferred to future work. The Orm system addresses the complexity of language implementation through, among others, the decomposition of a grammar specification into an *abstract* and *concrete* grammar. Extensibility and reusability are not addressed. Thus, there are several differences between the Orm approach and the DCO approach. First, Orm takes the grammar-interpretive approach, whereas DCOs extend the conventional generative approach. Second, a language implementation can be decomposed into multiple DCOs, whereas an equivalent Orm implementation would consist of a single abstract, concrete, etc. grammar, even when the size of the language implementation would justify a structural decomposition. Furthermore, the goals of the Orm system and the DCO approach are quite different. The Orm system aims at an interactive, incrementally compiling environment, whereas DCOs aim at improving the modularity and reusability of the traditional compiler construction techniques. It is thus difficult to compare these approaches.

7 Conclusion

The requirements on compiler construction techniques are changing due to certain trends that one can recognise, e.g. application domain languages, fourth generation languages and extensible language models. Due to this, the traditional, functional approach to compiler construction has been proven insufficient and leads to *complexity*, *maintainability* and *reusability* problems.

To address these problems, a way to structurally decompose a compiler into subcompilers, in addition to the traditional functional decomposition into a lexer, parser and code generator, is required. In this article, the notion of *delegating compiler objects* (DCO) is proposed as a solution. The major difference with a traditional compiler is that an input text, in the DCO approach, is compiled by a cooperating group of compiler objects rather than be a single, monolithic compiler. The result is a compiler that is much more modular, flexible and extensible than the conventional, monolithic compiler. The DCO approach is based on the ability of compiler objects to instantiate new compiler objects and delegate the compilation of pieces of the input syntax to these specialised compiler objects.

The delegating compiler object approach builds on the *parser delegation* and *lexer delegation* techniques. Traditional parsing approaches suffer from problems related to *complexity*, *extensibility* and *reusability*. Parser delegation offers an object-oriented approach to parsing which does not suffer from these problems. The parser specification syntax has been extended to support reuse and modularisation. Each grammar has a name and possibly a list of grammars it reuses. The production rules syntax has been extended to support extension and overriding of reused production rules.

Traditional lexing approaches, analogous to conventional parsing approaches, suffer from problems related to *complexity*, *flexibility*, *extensibility* and *reusability*. Lexer delegation is proposed as an object-oriented solution to these problems that facilitates modularisation and reuse of lexical analysis specifications. The syntax for lexical analysis specifications has been extended with elements for the specification of modularisation and reuse and for the extension and overriding of reused specifications.

The contribution of *delegating compiler objects*, *parser delegation* and *lexer delegation* is that these techniques comprise a novel approach to compiler construction which supports structural decomposition and reuse of existing specifications. The complexity, maintainability, extensibility and reusability of the resulting compiler is significantly better than the conventional approaches to compiler development.

Acknowledgements

Many thanks to the students that were involved in the construction of the PHEST, D-YACC and D-LEX tools.

References

1. A.V. Aho, R. Sethi, J.D. Ullman, "Compilers Principles, Techniques, and Tools," Addison Wesley Publishing Company, March 1986.
2. M. Aksit, R. Mostert, B. Haverkort, "Compiler Generation Based on Grammar Inheritance," *Technical Report 90-07,* Department of Computer Science, University of Twente, February 1990.
3. J. Bosch, "Parser Delegation – An Object-Oriented Approach to Parsing," in Proceedings of *TOOLS Europe '95,* 1995.

4. J. Bosch, "Relations as First-Class Entities in LAYOM," accepted for publication in *Journal of Programming Languages*, 1995.
5. J. Bosch, "Abstracting Object State," submitted to *Object-Oriented Systems,* December 1994.
6. J. Bosch, "Layered Object Model – Investigating Paradigm Extensibility," *Ph.D Thesis* (in preparation), Department of Computer Science, Lund University, October 1995.
7. J. Bosch, "Language Support for Design Patterns," to be published in proceedings of *TOOLS Europe '96,* 1996.
8. A. Dearle, "Constructing Compilers in a Persistent Environment," Technical Report, Computational Science Department, University of St. Andrews, 1988.
9. B. Fischer, C. Hammer, W. Struckmann, "ALADIN: A Scanner Generator for Incremental Programming Environments," *Software – Practice & Experience*, Vol. 22, No. 11, pp. 1011-1026, November 1992.
10. E. Järnvall, K. Koskimies, "Language Implementation Model in TaLE," *Report A-1993-1,* Department of Computer Science, University of Tampere, 1993.
11. E. Järnvall, K. Koskimies, M. Niittymäki, "Object-Oriented Language Engineering with TaLE," to appear in *Object-Oriented Systems,* 1995.
12. M.E. Lesk, "Lex – A Lexical Analyzer Generator," *Science Technical Report 39,* At&T Bell Laboratories, Murray Hill, 1975.
13. B. Magnusson, M. Bengtsson, L.O. Dahlin, G. Fries, A, Gustavsson, G. Hedin, S. Minör, D. Oscarsson, M. Taube, "An Overview of the Mjølner/Orm Environment: Incremental Language and Software Development," *Report LU-CS-TR:90:57,* Department of Computer Science, Lund University, 1990.
14. B. Magnusson, "The Mjølner Orm system," in: *Object-Oriented Environments - The Mjølner Approach,* J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, B. Magnusson (eds.), Prentice Hall, 1994.
15. B. Magnusson, "The Mjølner Orm architecture," in: *Object-Oriented Environments - The Mjølner Approach,* J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, B. Magnusson (eds.), Prentice Hall, 1994.
16. S. Minör, B. Magnusson, "Using Mjølner Orm as a Structure-Based Meta Environment," To be published in *Structure-Oriented Editors and Environments,* L. Neal and G. Swillus (eds), 1994.
17. V. Paxson, "Flex – Manual Pages," *Public Domain Software,* 1988.
18. Pheasant student project, *Pheasant project documentation,* University of Karlskrona/Ronneby, 1995.
19. Sun Microsystems Inc., "Yet Another Compiler Compiler," *Programming Utilities and Libraries,* Solaris 1.1 SMCC Release A Answerbook, June 1992.

This article was processed using the LaTeX macro package with LLNCS style