Parser Delegation
An Object-Oriented Approach to Parsing

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
WWW: http://www.pt.hk-r.se/~bosch

Abstract

Conventional grammar specification and parsing is generally done in a monolithic manner, i.e. the syntax and semantics of a grammar are specified in one large specification. Although this might be sufficient in static environments, a modular approach is required in situations where the syntax or semantics of a grammar specification are subject to frequent changes. The problems with monolithic grammars are related to (1) dealing with the complexity, (2) extensibility and (3) reusability. We propose the concept of *parser delegation* as a solution to these problems. Parser delegation allows one to modularise and reuse grammar specifications. To achieve this, the notion of a production rule is specialised into (1) overriding, (2) extending and (3) delegating production rule types. To experiment with parser delegation, we have developed D-YACC, a graphical tool for defining grammars. Parser delegation has been applied for constructing a translator for an experimental language and is currently applied in other domains.

[*]This work has been supported by the Blekinge Forskningsstiftelse.

1 Introduction

Traditionally, compiler constructors have taken a functional approach to parsing program text. Generally, the process consists of a lexical analyser, converting program text into a token stream, and a parser, converting the token stream into a parse tree. Both the lexical analyser and the parser are monolithic entities in that only a single instance of each exists in an application. Although this might be adequate for static environments, a more modular approach is required in cases where the syntax or semantic actions are subject to changes, or when one can recognise a clear partitioning in the grammar for parts of the input text.

The monolithic approach to grammar specification is becoming increasingly problematic due to, at least, two trends one can recognise. First, the emergence of special purpose languages. Whereas previously applications were build using one of the few general purpose languages, nowaday we often see that specialised, application (domain) specific languages are used. Examples of this can be found in the fourth generation development environments, e.g. Paradox[1], and formal specification environments, e.g. SDL. Each of these environments defines its own language as an interface to the user. The development of these special purpose languages calls for modularisation and reuse of grammar specifications. A second trend is the use of grammars to obtain structured input from

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

the user 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* and *tele-conference*. The digits are parsed by a parser and the requested service is activated for the user. A second example is the use of intermediate files to store application data. The retrieval of the data requires a parser to parse the file to recreate the stored structures. The described trends would benefit from a means to modularise and reuse grammar specifications as it would reduce the effort put in to the construction of parsers.

In this paper we describe *parser delegation*, a novel mechanism that allows one to modularise a syntax specification into multiple parsers. The base parser can instantiate other 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.

The parser delegation mechanism is not only used for modularisation, but also for reusing existing grammar specifications. The designer can specify, when defining a new grammar specification, that one or more existing grammar specifications are reused. The new grammar is extended with the production rules and the semantic actions of the reused grammar, but has the possibility to override and extend reused production rules and actions.

To evaluate the mechanism, we constructed a tool D-YACC for specifying grammars and grammar delegation and reuse. D-YACC uses an extended YACC specification syntax. The reason to use YACC is largely pragmatic. It is available on numerous machines and using YACC allowed us to concentrate on applying and evaluating *parser delegation*, rather than implementing our own parser generator. D-YACC allows the user to work on multiple grammar specifications simultaneously and to test parsers generated from the grammars.

To illustrate the *parser delegation* mechanism, we use the *layered object model* (LAYOM), the object-oriented research language we are currently investigating. As any research language, the syntax and semantics change frequently, although the changes are generally small. However, to be able to experiment with the language, we constructed a translator which translates LAYOM program text, e.g. class descriptions into C++ code, e.g. C++ class descriptions. In LAYOM an object is encapsulated by, so called, *layers*. These layers extend the behaviour of the class in many ways. Each layer type defines a certain type of functionality, but in general a layer defines a relation with an other object, e.g. an *inheritance*, a *conditional delegation* or an application defined relation, like *works for*, or a constraint, e.g. a *concurrency* or *real-time* constraint, on the behaviour of the object. However, often a new layer type, e.g. an application specific relation type, with its associated syntax and semantics, is introduced or the semantics or syntax of an existing layer are changed. When we would use a monolithic parser, we have to change the parser very often. Instead, we define a parser for each layer type and instantiate the layer parsers from within the class parser.

The remainder of this paper is organised as follows. In the next section, we briefly describe a number of problems with traditional, monolithic grammar specification techniques. In section 3 we propose a solution to these problems, i.e. the concept of *parser delegation*, which is explained by using the layered object model example. In section 4, D-YACC, the tool implementing parser delegation, is described and its use is illustrated by using LAYOM examples. Section 5 discusses work that is related to the parser delegation concept. The last section contains conclusions and a description of the future work.

2 The Problems of Monolithic Parsers

Traditionally, parsers are constructed as often large, monolithic entities. These parsers contain all the production rules for all of the syntax and all of the semantic information that is required. Below we describe three problems of these monolithic parsers.

- *Complexity*: When working with a large grammar, the designer can 'get lost' in the grammar. For instance, when working on one aspect, he makes changes that influence other parts of the grammar. Because conventional grammars do not provide a modularisation mechanism, this problem is unavoidable unless one can decompose a grammar specification into modules.

- *Extensibility*: A problem resulting from the complexity is the extensibility of a grammar. Software, being a model of a part of the real world, changes regularly and these changes should be incorporated in a, preferably, natural manner. However, a monolithic grammar specification does not provide for extension of the grammar. Basically, extending a grammar results in editing the original grammar specification. This, however, easily results in, again, the complexity problems of editing a grammar specification which was, possibly by someone else, defined some time ago. It would be preferable to be able to define the extensions to a grammar separate from the grammar itself.

- *Reusability*: When constructing a new grammar while having a related grammar available, one would like to reuse the existing grammar and extend and redefine parts of it. However, as described above, apart from the copy-paste facilities are no mechanisms available for reusing an existing grammar specification. Again, it would be advantageous to separate the reused grammar from the new grammar specifications.

In the categorisation above, we do not claim to be exhaustive in the problem identification. We primarily make a case for a technique that allows a designer to modularise, reuse and extend grammar specifications. Also others have identified one or more of these problems, see e.g. [Minör 94, Aksit 90]. In this paper we present the concept of *parser delegation* as a solution to the identified problems.

3 The Parser Delegation Concept

The *parser delegation* concept, as mentioned before, aims at supporting modularisation and reuse of grammar specifications. This is important for systems that are likely to be extended or reused. Conventionally, a grammar specification was a large monolithic description which was difficult to extend or reuse. We believe that applying an object-oriented approach, in particular the concept of *parser delegation*, to the specification grammars and the process of parsing will be very beneficial. In the following, the layered object model example will be introduced first. Then the parser delegation concept is described and illustrated by using this example. Due to space constraints, we concentrate on grammatical analysis and do not discuss lexical analysis, nor the sematic actions associated with production rules.

In this paper, we apply *bottom-up* parsing, rather than top-down parsing. The reason for this is twofold. First, bottom-up parsing handles a larger class of grammars than top-down parsing and, second, most software tools tend to use bottom-up methods. For example, YACC accepts LALR(1) grammars, but also has disambiguating rules to handle higher order LALR grammars. However, the concept of *parser delegation* is not depending on bottom-up parsing, it can also be applied to top-down parsing.

3.1 Example: Layered Object Model

The layered object model (LAYOM) [Bosch 94b] is an extension[2] of the conventional object model. The centre of the object consists of instance variables and methods, like the conventional object model, but the object is encapsulated by, so called, layers. These layers specify additional functionality of the object, generally in terms of relations with other objects and constraints on the object itself. Messages sent to or by the object have to pass the layers. These layers have the ability to

[2]In this paper, we use a simplified version of the LAYOM. The LAYOM object model supports many additional features, e.g. states, conditions and dynamic layer creation. However, these aspects are not relevant for the discussion in this paper.

change, delay, redirect and respond to messages send to and from the object. But a layer does not have to function in response to the receipt of a message. It can also be an active object, sending messages and monitoring the context it is placed in.

As each type of layer has its distinct functionality, it also has its own specification syntax. Although layers can share large parts of their specification syntax, most layers add some unique syntax elements. In figure 1, an example LOM class definition is shown.

In class *WaterTemperatureSensor* (see also figure 2) three layers are defined, i.e. a partial inheritance layer *pi_TS*, a delegation layer *d_WE* and a mutual exclusion layer *cc*. *pi_TS* defines a partial inheritance relation with class *TemperatureSensor*. The star denotes that it will inherit all methods from *TemperatureSensor*, but the third element, i.e. *'(calibrate)'*, indicates that the *calibrate* method is not inherited. Upon creation of an instance of class *WatertemperatureSensor*, the *pi_TS* layer will also create an instance of class *TemperatureSensor*. All messages sent to the object requesting a method implemented by class *TemperatureSensor* will be sent to the instance created by *di_TS*. The second layer *d_WE* delegates messages requesting the methods *checkSeal* and *calibrateConstant* to an external object called *WaterEquipment*. The third layer *cc* defines a mutual exclusion constraint of '1', meaning that not more than one thread is allowed to be active within the object. The semantics of LAYOM are only discussed very briefly here. We refer to [Bosch 94a, Bosch 94b, Bosch 94c] for a detailed description.

Each layer type has its own, independent syntax and semantics. Often, a new type of layer is added to the system. Also, the syntax and semantics of the existing layer types are changed regularly. If the parser for LAYOM would be implemented as a monolithic parser, we would have to change the parser very often with all the associated problems discussed in section 2. A mechanism that supports modularisation and reuse of grammar specifications would be extremely helpful in dealing with the complexity of constructing and maintaining a parser and code generator for LAYOM.

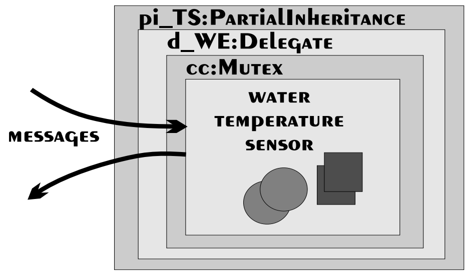


Figure 2: WaterTemperatureSensor object structure

3.2 Parser Delegation Concept

A monolithic grammar can be defined 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. [Aho 86, Sun 92]), a grammar in this definition has a name, and the start symbol is simply denoted by the production called *start*.

The concept of *parser delegation* allows one to delegate parsing of sections of the input token stream to parsers dedicated to that part of the syntax. In the example described in the previous section, the layer specifications are parsed by dedicated parsers of, respectively, the *PartialInheritance*, *Delegate* and *Mutex* type. The base parser, *ClassSpecification*, instantiates these layer parsers when it has parsed the layer name and the layer type. Parser delegation can be used in two ways, i.e. for reusing an existing grammar specification and for partitioning a large grammar into a collection of smaller grammar specifications.

A parser can be seen as an object and the input token stream can be viewed as messages to the object. The right-hand sides of the production rules are methods with method names equivalent to the associated grammar specification. If, after the receipt of a token, a production rule can be executed, the parser object will do so, otherwise the token is placed on the stack. If a production

```
class WaterTemperatureSensor
    layers
        pi_TS : PartialInheritance( TemperatureSensor, (*), (calibrate));
        d_WE : Delegate( WaterEquipment, (checkSeal, calibrateConstant));
        cc : Mutex(1); // concurrency constraint
    variables
        voltage : Real;
        calibrateValue : Real;
    methods
        calibrate (arg_1, ..., arg_n) returns Boolean
        begin
            ...code to calibrate the sensor...
        end;
        readTemperature() returns Integer
        begin
            ...code to calculate the temperature...
        end;
end WaterTemperatureSensor;
```

Figure 1: Example Class *WaterTemperatureSensor*

rule matches, the parser object will (1) execute the semantic actions associated with the production rule and (2) subsequently replace the right-hand side elements with the left-hand side nonterminal. This process of parsing continues until the top-level production rule has been executed.

In this paper, we concentrate on applying the concept of *delegation*, rather than inheritance, to achieving *reuse* in parsing. The rational for this is two-fold. First, others have defined inheritance mechanisms for parsing, and we, therefore, prefer investigating an alternative approach. Second, and more important, the concept of delegation provides a uniform framework for both reusing *and* modularising grammar specifications, whereas inheritance does not. We elaborate on this in section 5.

3.2.1 Reusing a Grammar Specification

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[3] uses the reused parser by *delegating* parts of the parsing process to the reused parser.

When reusing an existing grammar specification, a crucial aspect is the ability to override and exclude production rules from the reused grammar. To achieve this, the reusing parser must control the productions performed by the reused parser. A second important aspect is that the reusing grammar is able to extend productions declared at the reused parser. To allow *overriding* and *extension* of reused production rules, two types of production rule specification are defined. The first is the *overriding* specification, denoted by '*:*', and the second is the *extending* specification, denoted by '*+:*'. The semantics of the production rule specification types are defined below.

- $n : v_1\ v_2\ ...\ v_m$, where $n \in N$ and $v_i \in V$
  All productions $n \rightarrow V^*$ from G_{reused} are ex-

[3]The term 'reusing parser' is a shorthand for the parser generated from the reusing grammar specification. Similarly, 'reused parser' refers to the parser generated from the reused grammar specification

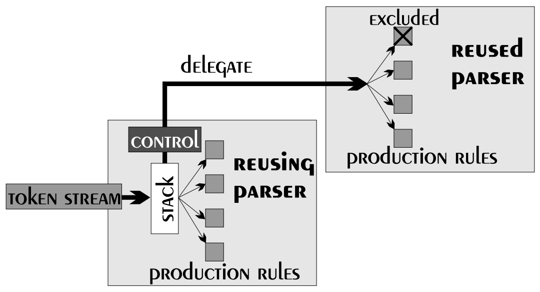


Figure 3: Parser Delegation for Grammar Reuse

cluded from the grammar specification and only the productions n from $G_{reusing}$ are included.

- 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 production rule types do not allow the designer to specify *exclusion* of production rules in the reused grammar. Exclusion is specified together with the reuse specification, i.e. when the name of the grammar is specified. Thus, when a grammar G_1 reuses a grammar G_2, G_1 can define a set $N^{excluded}_{G_2} \subseteq N_{G_2}$ such that if $n \in N^{excluded}_{G_2}$ then $n \notin N_{G_1}$.

In figure 3 the parser configuration for reuse is shown. The reusing parser has a reference to the reused parser. The reused parser has controlled access to the stack of the reusing parser. The access has to be controlled in order to be able to exclude production rules defined in the reused parser. As the production rules defined in the reusing grammar are always tried before the parser attempts to execute the reused production rules, no additional control for overriding productions is required.

3.2.2 Modularising a Grammar Specification

In the preceding text we described how grammar specifications can be reused. We will now describe how a grammar specification can be modularised. When modularising a grammar specification, the grammar specification is divided into a collection of grammar module classes, of which one is the *base* grammar class. A third type of production rule specification, called *delegating* production rule has been defined to coordinate between modules. When the parser object executes a *delegating* production rule, it creates a new parser object. The new parser object is an instance of the class specified by the production rule. 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.

The delegated parser starts with an empty token (or message) queue and an empty stack. It parses the input token stream from the current location l_x until it reaches the end of its grammar specification, at which point it will have reached location l_y. The delegating parser continues parsing at location l_y as if no tokens existed between l_x and l_{y-1}. The result of the delegating production is the instantiated parser object.

A delegating production rule is denoted by '*[id]:*', where *id* refers to the name of the delegated parser that is instantiated upon the actual production. One, very useful, type of *id* is *$i*, which refers to the value of the i-th element at the right-hand side. The semantics of this production rule type are shown below.

- 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 will return control to the delegating

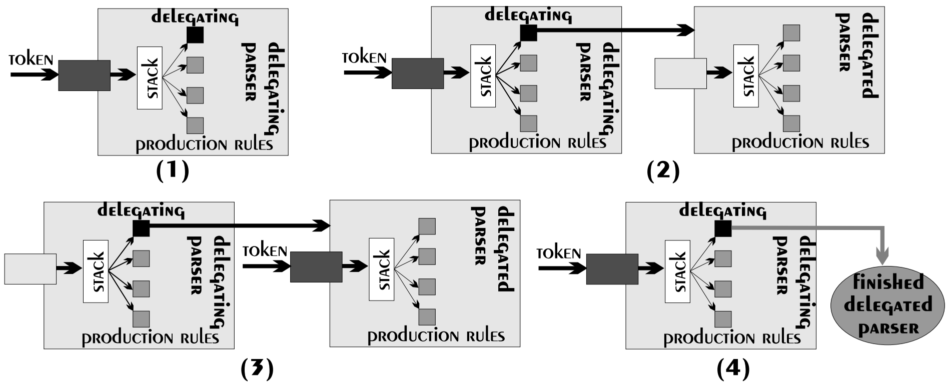


Figure 4: Parser Delegation for Grammar Modularisation

parser, which stores a reference to the delegated parser which contains the parsing result. When $\$i$ is used as an identifier, v_i must have a valid parser class name as its value.

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 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.

3.2.3 Formal Definition of Grammar Reuse

Here, we use the principle of parser delegation for the same reasons delegation is used in object-oriented languages: reuse and modularisation. Although these two goals are sometimes mixed, we have to separate them clearly in grammar specification. Because of the extended functionality of a grammar specification in D-YACC, we define a grammar G as $G = (I,\ R,\ N,\ T,\ P)$ where I is the name of the defined grammar, R is the set of reused grammars, N is the set of defined nonterminals, T the set of defined terminals and P the set of defined productions. R is defined as $R = (r_1, r_2, \ldots, r_n)$, where $r_i = (r_i^{name},\ E_{r_i})$. E_{r_i} is the set of excluded nonterminals of the grammar referred to by r_i^{name}. A Grammar G defined in D-YACC is equivalent to a monolithic grammar M. In figure 5, the relation between a D-YACC grammar G and a monolithic grammar M is defined.

Due to space and complexity reasons we have not included the delegated grammars in the formal definition. The resulting definition is so large and complex that it would not be supportive to this paper. The causes for the complexity of expressing delegated grammars also in an equivalent monolithic parser are the following:

- The production rules of the delegated grammar can only be used in the (monolithic) parsing process immediately after a corresponding delegating production has been executed.
- Similarly, after the delegating grammar is finished the production rules in the delegating grammar can not be used anymore.
- Also, while the monolithic parser is parsing delegated production rules, the production rules originally part of the delegating parser are not allowed to be used.
- Lastly, $\$i$ is used as the identifier in a delegating production rule, the delegated parser class is determined at run-time, i.e. when the input text is parsed. Therefore, it is not possible to define an equivalent monolithic parser in cases where $\$i$ is used.

We believe that the difficulty of specifying in an equivalent monolithic grammar for a D-YACC

A grammar G, as defined within the D-YACC environment,

$$G = (I_G, R_G, N_G, T_G, P_G)$$
$$R_G = (R_G^1, \ldots, R_G^q)$$
$$R_G^i = (I_{R_G^i}, E_{R_G^i}) \text{ where } I_{R_G^i} \text{ refers to the reused grammar } G_{R_G^i}$$
$$G_{R_G^i} = (I_{R_G^i}, R_{R_G^i}, N_{R_G^i}, T_{R_G^i}, P_{R_G^i})$$

has an equivalent monolithic grammar $M = (I_M, N_M, T_M, P_M)$, defined as

$$I_M = I_G$$
$$N_M = N_G \cup N_{R_G^1}^{included} \cup \ldots \cup N_{R_G^q}^{included}, \text{ where}$$
$$N_{R_G^i}^{included} = \{n \mid n \in N_{R_G^i} \wedge n \notin E_{R_G^i}\}$$
$$T_M = T_G \cup T_{R_G^1} \cup \ldots \cup T_{R_G^q}$$
$$P_M = P_G \cup P_{R_G^1}^{included} \cup \ldots \cup P_{R_G^q}^{included}, \text{ where}$$
$$P_{R_G^i}^{included} = \{p = (q, A), q : x \rightarrow \alpha \mid p \in P_{R_G^i} \wedge p \notin P_G \wedge x \notin E_{R_G^i}\}$$

Figure 5: A D-YACC grammar G and its equivalent monolithic grammar M

grammar incorporating delegated grammars is a clear indication that the expressiveness of D-YACC grammar specifications is larger than the expressiveness of conventional, monolithic grammar specifications.

3.3 Illustrating the Concept of Parser Delegation

In this section we illustrate the concept of *parser delegation* by using examples based on the LAYOM object model described in section 3.1. We first show an example of reusing a grammar and subsequently an example of modularising a grammar specification.

3.3.1 Reusing an Existing Grammar

We will now illustrate the reuse of an existing grammar specification with the example in figure 6. The *PartialInheritance* layer syntax has much in common with the *Inheritance* layer syntax, but extends it to allow exclusion of methods of the inherited class. In the grammar specifications, first the *Inheritance* layer syntax is specified and subsequently the *PartialInheritance* layer syntax. The latter reuses the production rules of the former and extends the *ClassDeclaration* rule with an additional right hand side that specifies, next to the class name, the methods that are to be inherited from the class. This allows the designer to reuse only a subset of the methods of a class.

3.3.2 Partitioning a Grammar Specification

Next to using parser delegation for reusing existing grammar specifications, it can also be used for modularising a grammar specification. When the base parser executes a production of the delegating rule type, it instantiates a parser of the specified parser type. After instantiation the delegating parser stores a reference to the delegated parser and subsequently hands it the control over the input token stream. The delegated parser parses the input token stream until it reaches the end of its grammar specification. It then returns control to the delegating parser which continues from the position in the input token stream where the delegated parser stopped. Do note that parser delegation is transitive in that the delegated parser can instantiate again a new parser and pass the control over the input token stream to it. In figure 7, the section of the *ClassSpecification* grammar specification for parsing layers is shown. In this example, when the Layer rule matches, the action part is executed, which instantiates and delegates to the new parser object of type \$3.

```
Inheritance

start            : '(' ClassDeclaration ')'
                 ;
ClassDeclaration : ClassName
                 ;
ClassName        : identifier
                 ;
```

```
PartialInheritance[Inheritance]

ClassDeclaration +: ClassName '(' MethodNames ')'
                 ;
MethodNames      : MethodName ',' MethodNames
                 | MethodName
                 ;
MethodName       : identifier
                 ;
```

Figure 6: Example of Parser Delegation for Grammar Reuse

```
LayerDeclaration    : 'layers' Layers
                    ;
Layers              : Layer ';' Layers
                    | /* empty */
                    ;
Layer               [$3]: identifier ':' identifier
                    ;
```

Figure 7: Example of Parser Delegation for Grammar Modularisation

4 Delegating YACC Parser Tool

To investigate and apply the concept of *parser delegation* in real applications, we have developed D-YACC[4], a graphical tool for specifying grammars that can reuse from and delegate to other grammars. For pragmatic reasons, we decided to make use of YACC for the actual parser generation. By doing this we were not required to build a parser generator, but could focus our effort on constructing a translator and a C++ preprocessor. For space reasons, we do not discuss the lexical analysis, nor the semantic actions, but concentrate on the grammatical analysis.

[4]D-YACC is currently a prototype tool, but we plan to improve it and, perhaps, make it publically available.

D-YACC can be seen as being composed of three parts:

- *User Interface*: The user-interface allows the grammar designer to work on the definition of grammar specifications. Grammars are specified in the D-YACC syntax which is an extension of the YACC specification syntax. The full specification of the D-YACC syntax can be found in appendix A. The tool allows the designer to work with multiple grammars and to test a generated parser using test input and examining the resulting test output.

- D-YACC *to* YACC *Translator*: The grammars are specified by the designer using the D-YACC syntax. As we make use of YACC, the input grammar has to be compliant with it. The translator converts a D-YACC grammar specification into a YACC grammar specification. For pragmatic reasons, i.e. to be able to make use of YACC, the reused grammars and the selected grammar are converted into one YACC grammar specification, rather than implementing it as described in section 3.2.1. Do note, however, that this makes no difference from the user's perspective as the generation is done automatically. This approach, therefore, does not suffer from the problems described in section 2. Delegated grammars for modularisation are implemented as described in section 3.2.2.

- C++ *preprocessor*: YACC generates a C++ file containing the parser, but it uses predefined function names. This predefined naming will cause name conflicts when we have multiple parsers in one application. Therefore, we preprocess the C++ files to rename all potentially conflicting names, e.g. **yy**foo, into unique identifiers, e.g. **mygrammar**foo.

In figure 8, the conceptual organisation of D-YACC and the components it uses in its course of operation are shown. We will now describe what happens if a user decides to test a grammar. The D-YACC environment generally contains multiple grammars which have some relation with each other. The selected grammar is analysed to determine what grammars it uses. The selected grammar and all reused grammars are converted into one YACC grammar specification. Each grammar used for modularisation purposes, specified in D-YACC syntax, is translated into a corresponding YACC specification. Each YACC specification is converted into a C++ program by YACC. The D-YACC preprocessor will convert the resulting C++ program into an equivalent C++ program in which all YACC specific identifiers are renamed to unique names. This last conversion is required to have multiple parsers coexisting in a C++ application. The translator has also generated a *makefile* at the generation of the YACC specification. When it has converted, generated and preprocessed all the required grammar parsers, D-YACC will start the *make* tool to compile and link all necessary files and the result is an executable program containing all the specified parsers. The user can now test the grammar by typing text into the input text window and starting the executable program. The result is printed in the output text window of the user interface. In figure 9 the user interface of the tool is shown.

5 Related Work

The need for grammar reuse and modularisation has been recognised by several researchers. In [Minör 94], the authors mention the importance for grammar reuse and draw the analogy between code reuse and grammar reuse, but they defer this to future work. Instead they try to deal with the complexity of grammar specification through the decomposition of a grammar specification into an *abstract* and *concrete* grammar. Extensibility and reuseability are not addressed.

In the field of *attribute grammars*, object-oriented concepts are, among others, applied by [Hedin 89, Grosch 90, Kastens 92]. However, the concepts are, in general, not applied to reuse and/or modularise grammar specifications, but to deal with attributes and attribute computation. Hedin [Hedin 89] describes an object-oriented notation for attribute grammars where sub(child)productions are specified as subtypes of their super(parent)productions. This results in the grammar specification being represented analogous to a classification hierarchy. Although this reduces the complexity of the grammar specification, little support is offered for modularising, extending or reusing an existing grammar specification. A similar approach is taken in [Grosch 90], where attributes and attribute computations are inherited from the supertype. Kastens and Waite [Kastens 92] take a slightly different approach by defining *attribution modules* containing abstracted semantics, i.e. attributes and attribute computations, that can be reused.

One approach to reuse of grammar specifications is *grammar inheritance*. In [Aksit 90] a mechanism for 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

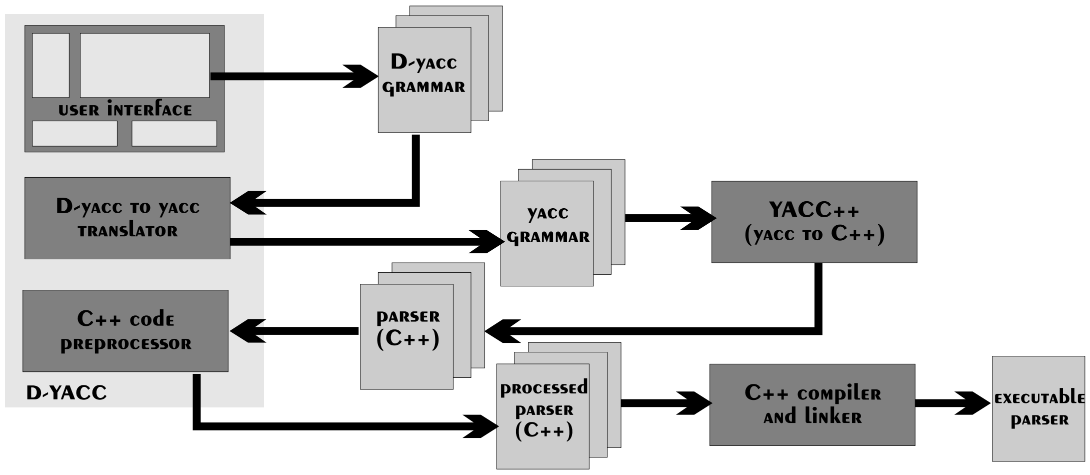


Figure 8: Conceptual view of D-yacc

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.

An approach to facilitate evolutionary parser development is described in [Hucklesby 89]. They use a parsing library with classes that represent nodes in the grammar specification. Extensibility is supported by inserting intermediate superclasses of nodes. It, however, does not support grammar decomposition, nor does it facilitate reuse of grammar specifications as a whole. Another approach, based on the aformentioned work, is described in [Grape 92] where the authors aim at a high degree of seperation between syntax and semantics as a means to support extensibility. This is achieved by modelling syntax trees separately from parse trees and by defining actions that convert the parse tree into a syntax tree. The syntax tree is supposed to be more stable than the parse tree, because the latter changes for every grammar change whereas the set of keywords and operators, which makes up the syntax tree, tends to be more stable. The work of [Grape 92] does not provide means for modularising, reusing or extending grammar specifications.

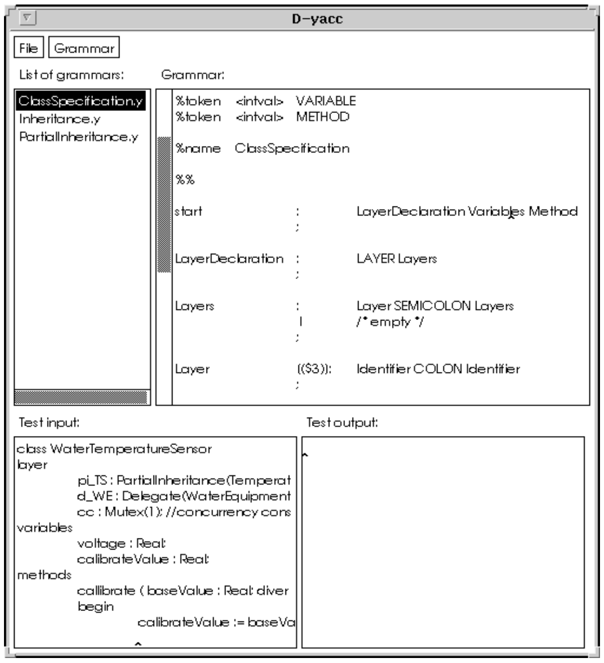
In a way, one could view a *preprocessor*, e.g. the c++ preprocessor, as a means to modularise and extend an existing grammar. A preprocessor can process the input text and replace parts of the input text with text that can be parsed by the parser. We consider this solution to be inferior to *parser delegation* for, among others, the following reasons:

- It requires the semantics of the preprocessed input text to be expressable in terms of the grammar on which the parser is based.
- It provides no means for reusing, overriding or extending parts in the parser.
- It does not allow for changing the semantic actions, e.g. the way the parse tree is constructed, in the parser.

To the best of our knowledge, no approaches for modularising grammar specifications have been defined. Also, the application of the concept of *delegation* for the reuse and extension of grammar specifications we believe to be novel.

6 Conclusion and Future Work

The traditional, monolithic approach to grammar specification has a number of problems when



```
%token   <intval>   VARIABLE
%token   <intval>   METHOD

%name   ClassSpecification

%%

start               :           LayerDeclaration Variables Method
                    ;

LayerDeclaration    :           LAYER Layers
                    ;

Layers              :           Layer SEMICOLON Layers
                     |          /* empty */
                    ;

Layer               (($3)):     Identifier COLON Identifier
                    ;
```

```
class WaterTemperatureSensor
layer
        pi_TS : PartialInheritance(Temperat
        d_WE : Delegate(WaterEquipment
        cc : Mutex(1); //concurrency cons
variables
        voltage : Real;
        calibrateValue : Real;
methods
        callibrate ( baseValue : Real; diver
        begin
                calibrateValue := baseVa
```

Figure 9: User Interface of the D-YACC Tool

the syntax and semantics are subject to frequent changes. These problems are related to (1) dealing with the complexity of a large grammar specification, (2) the difficulty of extending a grammar specification and (3) the impossibility of reusing a grammar specification in a satisfying manner. In this paper we have proposed *parser delegation*, a novel concept to deal with reusing and modularising grammar specifications. The concept of parser delegation provides a solution to the problems associated with conventional, monolithic grammar specification.

To investigate and apply parser delegation in real applications, we have developed D-YACC, a graphical tool for specifying grammars that can reuse from and delegate to other grammars. For pragmatic reasons, this tool converts a grammar specification in D-YACC into a YACC grammar specification. D-YACC modifies the C++ code generated by YACC to allow multiple parsers in a single C++ application. D-YACC is currently a restricted prototype, but we plan to improve it and, perhaps, make it publically available.

We have used parser delegation in the LAYOM translator. Currently, parser delegation is being applied in the telecommunications domain. A modern phone exchange, nowadays, offers many services which are requested by dialing digits, the $\star$ and #. These services change over time and when the parsing of the service requests would be done by a monolithic parser, this parser would need to be updated regularly. By configuring each type of service with its own parser, the base parser can delegate the service requests to the parser specific for that service type. We intend to apply the concept of *parser delegation* in several other application domains.

Acknowledgements

I would like to thank Bertil Ekdahl, Michael Mattson, Peter Molin, Lennart Ohlsson, Petra Sijtsema and the anonymous referees for their comments on earlier versions of this paper and Jonas Nilsson for implementing the prototype of the D-YACC tool.

References

[Aho 86] A.V. Aho, R. Sethi, J.D. Ullman, "Compilers Principles, Techniques, and Tools," Addison Wesley Publishing Company, March 1986.

[Aksit 90] 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.

[Bosch 94a] J. Bosch, "Paradigm, Language Model and Method," submitted to the *ICSE-17 Workshop on research issues in the intersection of Software Engineering and Programming Languages,* November 1994. Also Research Report 8/94, Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, November 1994.

[Bosch 94b] J. Bosch, "Relations as First-Class Entities in LAYOM," submitted to *ECOOP '95,* November 1994. Also Research Report 9/94, Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, November 1994.

[Bosch 94c] J. Bosch, "Abstracting Object State," submitted to *Object-Oriented Systems,* December 1994. Also Research Report 10/94, Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, December 1994.

[Grape 92] P. Grape, K. Waldén, "Automating the Development of Syntax Tree Generators for an Evolving Language," *Proceedings of Techology of Object-Oriented Languages and Systems (TOOLS 8)*, pp. 185-195, Santa Barbara, Aug. 1992.

[Grosch 90] J. Grosch, "Object-Oriented Attribute Grammars," *Report No. 23,* Project Compiler Generation, GMD, August 1990.

[Hedin 89] G. Hedin, "An Object-Oriented Notation for Attribute Grammars," *Proceedings of the European Conference on Object-Oriented Programming,* pp. 329-345, BCS Workshop Series, July 1989.

[Hucklesby 89] P. Hucklesby, B. Meyer, "The Eiffel Object-Oriented Parsing Library," *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 1),* pp. 501-507, Paris, Nov. 1989.

[Kastens 92] U. Kastens, W.M. Waite, "Modularity and Reusability in Attribute Grammars," *Technical Report CU-CS-613-92,* University of Colorado at Boulder, September 1992.

[Minör 94] 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.

[Sun 92] Sun Microsystems Inc., "Yet Another Compiler Compiler," *Programming Utilities and Libraries,* Solaris 1.1 SMCC Release A Answerbook, June 1992.

A D-yacc Syntax

```
spec       : defs MARK name rules tail
           ;
tail       : MARK
           | /* empty*/
           ;
defs       : /* empty */
           | defs def
           ;
def        : START IDENTIFIER
           | UNION { copy union definition to output }
           | LCURL { copy C code to output file }  RCURL
           | defs rword tag nlist
           ;
rword      : TOKEN
           | LEFT
           | RIGHT
           | NONASSOC
           | TYPE
           ;
tag        : /* empty */
           | '<' IDENTIFIER '>'
           ;
nlist      : nmno
           | nlist nmno
           | nlist ',' nmno
           ;
nmno       : IDENTIFIER
           | IDENTIFIER NUMBER
           ;
name       : IDENTIFIER reuse
           ;
reuse      : /* empty */
           | '[' grammars ']'
           ;
grammars   : grammar ',' grammars
           | /* empty */
           ;
grammar    : IDENTIFIER
           | IDENTIFIER '(' exclusions ')'
           ;
exclusions : IDENTIFIER
           | IDENTIFIER exclusions
           ;
rules      : /* empty */
           | rules rule
           ;
rule       : IDENTIFIER prtype rbody prec
           | aprtype rbody prec
           ;
prtype     : ':' /* overriding rule */
           | '+:' /* extending rule */
           | '[' IDENTIFIER ']:' /* delegating rule */
           ;
aprtype    : '|' /* overriding rule */
           | '+|' /* extending rule */
           | '[' IDENTIFIER ']|' /* delegating rule */
           ;
rbody      : /* empty */
           | rbody IDENTIFIER
           | rbody act
           ;
act        : '{' { Copy action, translate , etc. }  '}'
           ;
prec       : /* empty */
           | PREC IDENTIFIER
           | PREC IDENTIFIER act
           | prec ';'
           ;
```