Interactive Tool Support for Domain-Specific Languages

Elizabeth Bjarnason

CODEN: LUNFD6/(NFCS-3124)/1-112/(1997)
LU-CS-TR:97-192

Lund, November 1997

To Jesus who was with me every step of the way

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund, Sweden

E-mail: Elizabeth.Bjarnason@dna.lth.se

Front cover: Part of the abstract syntax tree of a program expressed in a domain-specific language, controlling an ASEA/ABB Irb-6 robot (picture from the robot manual, digitized by Rolf Braun). Calls in the language implementation to the motion control interface are dynamically bound in the embedded execution environment.

Abstract

Domain-specific languages are used in numerous problem domains, e.g. database handling, robot programming and numerical calculations. The main advantage of using such languages is the ease with which a problem within the domain can be expressed in the language. The resulting programs are easier to debug and maintain, and consist of higher quality code since the language constructs are close to the concepts of the domain. The drawback of using domain-specific languages has, until now, been the huge cost, both in terms of time and skill, in developing a language which covers the domain in a good way. This cost could be reduced by easy-to-use tools which support the language-design process in an interactive way. Furthermore, an interactive programming environment for domain-specific languages would support the programmer in editing programs according to the syntax and static semantics of a language. The aim of this thesis is to present techniques for developing such interactive tool support for domain-specific languages, both at the language-design level and at the programming level.

Rapid language prototyping can be supported by tools built on the principle of grammar interpretation. The programming environment then automatically adapts whenever the grammar is changed. This thesis presents techniques for parsing and static-semantic analysis based on this principle. The resulting tools are used to support language-sensitive text editing and advanced semantic editing of both programs and grammars.

The development of new languages can be further enhanced by reusing existing language specifications. *Object-oriented modular grammars* support such development of languages by combining a number of modules, or 'building blocks'. Furthermore, *multi-layered grammars* can be used to design languages for multi-level systems where a language is implemented in terms of the routines of a library or framework.

APPLAB, an interactive language laboratory and programming environment, has been implemented to support the techniques presented in this thesis. The system has been used in a number of case studies, including a study on robot programming where a full-sized language was implemented. Our experience shows that the presented techniques provide valuable support for interactive development of domain-specific languages, as well as enhanced programming support for such languages.

Acknowledgements

The research presented in this thesis has been carried out within the Software Development Environments Group at the Department of Computer Science, Lund University. I am indebted to my supervisor, Görel Hedin, who introduced me to the field of domain-specific languages, and static-semantic support within programming environments. She very patiently and carefully read drafts of my reports and papers, as well as this manuscript, giving page after page of detailed comments. Thank you for sharing your knowledge of this area, as well as helping me with lots of practical issues (like “How does one do this in Framemaker?”) I also want to thank Klas Nilsson for introducing me to his ‘baby’, one of the ABB robots at the Department of Automatic Control, Lund University, and for giving me insight into the world of robot programming. Thanks for encouraging, as well as helpful comments on my various drafts and on this thesis.

I want to thank the rest of our research group; Boris Magnusson, Roger Henriksson, and Ulf Asklund, as well as previous members Torsten Olsson and Anders Dellien, and newer members, Anders Ive and Patrik Persson. Thank you, Göran Fries, my supervisor during my first term at the Department, for introducing me to the world of Ph.D. studies.

During the thesis process I have learnt a great deal about writing. I am indebted to my mother (and Swedish father) for teaching me the English language from birth (starting with ‘teddy’ and ‘tractor’), as well as teaching me to read and write the language at an early age. Thanks also Peter Middleton and Julian Gillett, for answering detailed questions about spelling and writing in British English.

My children have been a source of great joy, and have forced me to take time off from work for breathers. Thank you, Jonathan and Michaela, for helping me to not loose track of the ‘important’ things in life; like chasing the autumn leaves and discovering how to climb over the gate to the stairs. Also, thank you, Michaela, for your smile at six o’clock in the mornings!

Finally, but not least of all, I want to give a big round of applause to my husband, Bjarni, for encouraging me to go on when I felt like giving up, for caring for the kids during weekends, when I have been away on conferences or when they have been ill, and for sharing the household chores. You are a one in a million; a hero!

This work has been financially supported by NUTEK, the Swedish National Board for Industrial and Technical Development.

Contents

Chapter 1

Introduction

In industry, today, a lot of resources are poured into programming large systems. A lot of the work involves ensuring that the produced code is correct and safe. There is a lot to be gained by aiding the programmer in writing more correct programs to start off with. Such support can be supplied by designing a *domain-specific language*, or DSL, that captures the vocabulary and the rules of the problem domain [War94, TMC97]. Taking the step from general-purpose languages to DSLs means that the same advantages as were achieved when programmers moved from *assembly languages* to *high-level languages* are obtained once again. The implementation details of the domain concepts are dealt with in the language implementation rather than in the application programs, thus, increasing the level of abstraction and decreasing the *semantic gap* [Bos96]. And, just as one can use a high-level programming language without having any knowledge of details of the physical machine like registers and memory location, one can program in a DSL without having to consider the low-level details of the underlying implementation. Instead, the programmer can concentrate on the problem at hand. In short, DSLs can reduce the amount of effort needed to program, debug, and maintain applications within the problem domain.

There are several different approaches to modelling DSLs. A DSL can be designed from scratch which often results in a very specialized, high-level language [Ben86]. Such languages often lack the abstraction mechanisms of general-purpose languages, but contain constructs which are very close, both in syntax and semantics, to the concepts of the problem domain. Another common approach is to extend a general-purpose language with domain-specific syntax by using syntactic macros. In this way the surface syntax of the language is closer to the concepts of the problem domain. A third approach is to use a language which is well suited to modelling different problem domains [Hud96, Kos96, KH97]. The abstraction mechanisms available in object-oriented languages make them very suitable for such domain-specific modelling. This was noted already in the definition of the object-oriented programming language Simula [DMN68].

In the late sixties Sammet deemed *user-defined languages* (or DSLs) to be “the most likely and promising future development” [Sam69, p. 727] within the area of programming languages. This has not been the case, partly due to the introduction

of object-oriented languages which have been successfully used for modelling problem domains. Other major factors which have limited the use and acceptance of the DSL technique include the huge cost of developing and maintaining a new language, the problem of compatibility and reusability of software across different platforms, and that the needed technology and tools for compiler construction have not been available [Nil96b, DK97].

In conclusion, in many programming situations it is desirable to use DSLs but not all companies have the resources, both financial and technical, needed to design and implement their own languages. In order to make the use of such languages more feasible there is a need for dedicated environments supporting the programmer, as well as tools for rapid development of such environments.

Goals

The goal of the research presented in this thesis was to develop techniques allowing the construction of easy-to-use tools for designing and implementing domain-specific languages, as well as programming environments for DSLs. In particular, we were interested in providing support for *rapid language prototyping* with which the development costs for DSLs can be decreased, making the approach more widely available. The compiler construction tools available today can be used to implement a language but do not support the language design process as such.

Tools and environments for supporting the programmer working with DSLs are very useful since one can expect the application programmer to be more of an application expert than a programming expert. Furthermore, DSLs are often used on an irregular basis, and the language may often need to be revised as new features and requirements are added to the system. This makes it hard for the programmer to stay acquainted with the details of the language.

Method

This research is a continuation of the work done within the Mjølner/Orm project [MHM+90, Mag93], a project concerned with building interactive programming environments for object-oriented languages based on techniques for structure-oriented editing [Min90], grammar interpretation [MH93] and incremental static-semantic analysis [Hed92a]. The technique of grammar interpretation was applied to parsing and static-semantic analysis by studying previously developed techniques for generating recursive-descent parsers [Hed89b] and for evaluating object-oriented AGs based on the implementation techniques for virtual functions in object-oriented languages [Hed89a]. The resulting techniques were implemented in APPLAB (Application Language Laboratory) which is a further development of the grammar editor of the Mjølner/Orm environment [MHM+90, MM94]. The system has been used as a test bed for experimenting with the developed techniques. A number of case studies have been performed in APPLAB with the purpose of evaluating how well these techniques support the DSL approach.

Also, by experimenting with extensions of object-oriented AGs and studying other AG versions, like higher-order AGs [VSK89], techniques for supporting the reuse and extension of language specifications have been sketched.

Results

The main contributions of this thesis are grammar-interpreting techniques for parsing and static-semantic analysis within a structure-oriented environment, and techniques for supporting reuse of language specifications. Combined with previously developed grammar-interpreting support for structure-oriented editing these techniques support rapid language prototyping.

The presented parsing technique is based on recursive-descent parsing and a generic parser based on this technique has been integrated into the APPLAB system. Object-oriented AGs are used to describe static-semantic rules of a language. The presented implementation techniques for a generic demand-driven attribute evaluator have been used in APPLAB to support user-defined advanced editing features like semantic editing and semantic error markers. The semantic editing features of the Mjølner/Orm system were obtained in APPLAB by allowing these features to be specified by the language designer.

The language-sensitive programming environment provided by structure-oriented editing, semantic editing and language-sensitive text editing is, by utilizing the techniques of grammar interpretation, also available at the language-design level which further enhances the language-design process. This aids the language designer in constructing a syntactically and static semantically correct language specification.

The *object-oriented modular grammars* presented in this thesis support the reuse and extension of existing language specifications. The presented *multi-layered grammars* support the development of multi-layered languages where a language level is implemented in terms of the routines of a lower level.

Through a case study on robot programming the presented techniques were applied to a realistic and challenging application domain. The rapid language prototyping proved valuable, as well as the semantic editing features. The techniques for object-oriented modular grammars and multi-layered grammars are not yet implemented but are expected to provide valuable support for the development of multi-layered languages.

1.1 About the Thesis

Outline

The rest of this thesis is organized as follows:

- Chapter 2: Domain-Specific Languages
  The concept of domain-specific languages is discussed and a number of problems and requirements on support for such languages are presented.
- Chapter 3: APPLAB - A Language Laboratory
  A general description of the APPLAB system is given, as well as an overview on the design of a new language in the system.

- Chapter 4: A Grammar-Interpreting Static-Semantic Analyser
  A technique for implementing generic demand-driven attribute evaluators for object-oriented AGs is presented.
- Chapter 5: A Grammar-Interpreting Parser
  A technique for implementing generic recursive-descent parsers in a structure-oriented environment is presented.
- Chapter 6: Case Study - Robot Programming
  A case study using APPLAB to support the development of a DSL for robot programming is presented.
- Chapter 7: Grammar Modularization
  Techniques for supporting the reuse of language specifications in a structure-oriented environment are presented.
- Chapter 8: Evaluation
  An evaluation of APPLAB and, thus, the techniques presented in this thesis is performed against the requirements of DSLs. Some measurements on time and memory consumption of the system are also presented.
- Chapter 9: Conclusions and Future Work
  The contributions of this thesis are summarized and future work is identified.

Publications and Contributions

Chapters 3, 5, 6 and 7 present contributions which have previously been published as follows.

The general description of the APPLAB system found in Chapter 3, in combination with a brief description of the robot case study discussed in Chapter 6, is presented in

> E. Bjarnason, G. Hedin and K. Nilsson. “APPLAB-An Application Language Laboratory”. Technical report LU-CS-TR:97-188, Department of Computer Science, Lund University, Sweden, August 1997.

The APPLAB system has also been described in the position paper

> E. Bjarnason. APPLAB - A Laboratory for Application Languages. In *Proceedings of NWPER'96 (Nordic Workshop on Programming Environment Research)*, L.Bendix, K. Nörmark, and K. Österbye (eds), Technical Report R-96-2019, Aalborg University, Denmark, May 1996.

and in the user guide

> E. Bjarnason. APPLAB User's Guide (Version 1.2) Technical Report LU-CS-IR:96-01, Department of Computer Science, Lund University, Sweden, April 1996.

The parser, described in Chapter 5, is based on an object-oriented recursive-descent parsing algorithm developed by Görel Hedin [Hed89b]. I adapted and implemented the algorithm in the grammar-interpreting environment of APPLAB. This implementation work is also presented in

> E. Bjarnason and G. Hedin. A Grammar-Interpreting Parser in a Language Design Laboratory. In *Proceedings of the Poster Session of CC'96 (International Conference on Compiler Construction).* P. Fritzson (ed.), LiTH-IDA-R-96-12, Dept. of Computer Science, Linköping University, Sweden, April 1996.

The static-semantic analyser, presented in Chapter 4, is based on an algorithm for demand-driven attribute evaluation of OOAGs developed by Görel Hedin [Hed89a]. Similarly to the parser, I adapted and implemented the algorithm to work in the grammar-interpreting environment of APPLAB.

The robot case study discussed in Chapter 6 was joint work with Klas Nilsson at the Department of Automatic Control, Lund University. Based on the principles presented in [NBL98], he developed and implemented the embedded execution environment used to execute robot programs developed in APPLAB, while I made the necessary adaptations to APPLAB and implemented the RAPID language. A summary of the case study has been presented in

> E. Bjarnason and K. Nilsson. Languages for Embedded Systems. In *Preprints of SNART 97 Konferens om Realtidssystem*, Lund, Sweden, August 1997.

The techniques for supporting the reuse of language specifications proposed in Chapter 7 were developed together with Görel Hedin. These initial ideas have been presented in the following position paper

> E. Bjarnason. Tool Support for Framework-Specific Language Extensions. In *Proceedings of LSDF'97 (Workshop on Language Support for Design Patterns and Frameworks held in conjunction with ECOOP'97)*, Jyväskylä, Finland, June 1997.

A shorter version of the same paper will be published in Lecture Notes of Computer Science 1357, Springer Verlag, (ECOOP'97 Workshop Reader.)

Chapter 2

Domain-Specific Languages

In this chapter we discuss the concept of domain-specific languages, i.e. languages that are especially well suited for programming a certain set of problems. After an introduction including the benefits of using this technique, different types of domain-specific programming are identified and exemplified. Then, we look at problems which arise when using domain-specific languages and how these can be solved. The chapter ends by drawing some conclusions.

2.1 Introduction

The language determines our expressiveness. Consider our natural language and a small child learning to speak. At first she can only express simple needs like 'water' and 'dolly', but as her language expands and is refined, the child can express what she wants in more detail. For example, "I want to give my favourite doll a glass of water". Moving on to natural languages in general, there are different sets of vocabulary within one language. One for domestic purposes, one for medical purposes, one for technical discussions, and so on. In order to be able to express ourselves, and to be understood by others, we need a (common) language which is well suited to the subject we wish to discuss. Similarly, when programming, it is important that the programming language is suitable for the current set of problems. That is, it should be expressive within the problem domain. There are general-purpose programming languages (corresponding to everyday English), like C, Java, Pascal etc., that serve as a basis for programming in general. But, when dealing with problems in a specific domain, it is often cumbersome to have to express those domain concepts in terms of a general-purpose language (comparable to explaining a medical condition without using any medical terms).

The abstraction mechanisms of a (general-purpose) base language can be used to create a library or a framework containing the domain-specific concepts, thus, introducing a domain-specific 'vocabulary'. The degree to which the syntax and the semantics of the domain can be reflected in this way depends on the strength of the abstraction mechanisms of the base language used. The aspects which can not be described directly in terms of the base language will then be conventions which the application programmer is expected to know about and follow. This requires that

the application programmer to some extent acquaints himself with the inner workings of the library or framework. Failure to correctly use the library can lead to serious errors which are often left undetected until run time. The expressiveness of the programmer would be greatly enhanced by a language which allows him to express the problem at hand, in a simple way, in terms which correspond well to the vocabulary and reasoning of the problem domain. That is to say, the *semantic gap* between the problem domain and the formal language used to describe it, should be as small as possible. Using such a language reduces the programming time needed to produce an application, as well as increases the quality of the produced code. We call such a formal (programming or specification) language a *domain-specific language*, or DSL. The technique of using a special language to develop a system, or a set of systems, has been called *Language-Oriented Programming* [War94], or *SDRR (Software Design for Reliability and Reuse)* [BBH+94, Kie95].

Taking the step from general-purpose languages to DSLs is comparable to moving from assembler to high-level languages. When high-level languages were first introduced, implementing a language was something left to the experts. Since then, compiler technology has come a long way and there are now plenty of compiler-compiler tools which aid in the implementation of new languages. However, these tools do not support the actual language-design process. To further support the design and implementation of well-designed DSLs, there is a need for further research in this area. Due to the advances in compiler technology, now the time seems to be ripe to do so, and over the past few year a number of workshops have been held in this area [Kam93, SEPL95, SEPL96, ALEL96, DALEL96, DSL97b, LSDF97], as well as the first conference on domain-specific languages [DSL97].

There are a number of different approaches to domain-specific programming including object-oriented frameworks, little languages, and extended languages. After having further looked at the benefits of DSLs we will attempt at giving an overview of these different approaches. In this thesis we are mainly concerned with textual languages, but, of course, for domains which are well suited for graphical representation graphical languages are more suitable.

2.2 Benefits of DSLs

> *The first characteristic of a* [domain-specific] *programming language is that the user can write a program without knowing much - if anything - about* [the underlying implementation layer, or,] *the physical characteristics of the machine on which the program is to be run.*
>
> [Sam69, p. 9]

The main idea of DSLs is the same as for programming languages in general, as is captured in the quote above. A language should allow the programmer to program an underlying (complex) layer without requiring detailed, or any, knowledge about how it works. In the DSL case this means that the problem domain should be made available to the programmer, in such a way that application programs can be written without considering the implementation of the domain concepts. The DSL approach includes a range of techniques for domain-specific programming. Even

though the techniques vary, they have a number of common traits and benefits due to the support of a formal language, and to the benefits of separating the concerns, i.e. the domain-specific concepts and the actual application.

Reduced Programming Time

In general, the more domain-specific support a DSL provides, the easier it is to learn and use. This support is especially useful for someone who is more of a domain expert than a programming expert. Since the implementation details of the domain-specific constructs are dealt with by the language, the programmer can focus on the logic of the application program and does not need to worry about the details of the underlying implementation. The language, thus, aids the user, especially non-programming experts, in expressing the problem at hand in terms of the programming language.

Increased Quality

The programs written using a DSL are usually shorter and, if domain-specific syntax is used, closer to the domain at hand. Less code generally means less errors and this is also true for these applications [DK97]. Furthermore, when working in a language that enforces domain-specific syntax and static-semantic rules, a number of errors that would otherwise remain undetected until run-time are reported by the compiler. Thus, the number of errors in application programs can be expected to be less when using a DSL.

2.3 Different Types of DSLs

We have chosen a very general definition of DSLs, so as to include a range of different techniques for handling domain-specific programming. Our definition of DSLs includes

- using the abstraction mechanisms of an existing language to create a set of domain-specific entities, e.g., a framework or a library,
- little languages limited to expressing problems within the domain, and,
- extending a general-purpose language with domain-specific syntax and/or static semantics.

We will now take a closer look at these different types of DSLs. Their advantages and drawbacks vary due to the set of requirements for a programming project, e.g., on efficiency, amount of resources, security demands, and skills in language design and compiler technology.

2.3.1 Libraries and Frameworks

The use of libraries and frameworks together with a general-purpose language is one very common way of coping with domain-specific programming. A set of domain-specific entities are then introduced into a language by utilizing the abstraction mechanisms of that existing language. This approach results in a *domain-specific embedded language, DSEL* [Hud96], or a *light-weight DSEL*, a term introduced by Hudak in the keynote address of DSL'97 [Hud97]. The technique is used in many different types of programming languages, e.g., procedural languages (resulting in libraries), functional languages [Hud96, Kam96], LISP [San78], object-oriented languages (resulting in object-oriented frameworks). In the rest of this chapter the term 'library' will be used to denote such a set of domain-specific entities. Isolating the implementation of the domain concepts in this way makes it easier to retarget the set of applications using the library. It is also simpler to extend and maintain than a complete language and its implementation. And, if the source code of libraries is available they can be adapted as new needs arise [GWB+96]. The application programs using the library will still contain non domain-specific details, like the number and order of arguments, the exact name of the desired routine (which often can not be very 'domain-specific' due to the large amount of routines in a system). Also, there are a number of conventions attached to the correct use of a library. Some of these conventions can be expressed by using the abstraction mechanisms of the language used for implementing the library. The more abstraction mechanisms the implementation language contains, the more precisely the syntax and semantics of the domain entities can be expressed, reducing the number and complexity of the (formally unexpressed) conventions [Hed97a].

Object-Oriented Frameworks

The object-oriented paradigm is very suitable for modelling domain entities and, thus, for introducing domain-specific language constructs within the language. This was also stated as one of the main points of the object-oriented language Simula, where the following can be read in the preface of the language definition [DMN68]:

> *A main characteristic of SIMULA is that it is easily structured towards specialized problem areas, and hence can be used as a basis for Special Application Languages.*

Object-oriented frameworks [Deu89, Joh88] have a lot in common with DSLs [Kos96]. They both consist of constructs for the entities of the domain and can be used for programming applications within that domain. In [RJ97], the design of DSLs is described as an evolutionary process starting with the development of a framework, which usually starts off as a white-box framework [Joh88] and then gradually evolves into a black-box framework, as more domain-specific details are added. The 'blacker' a framework becomes the more it takes on the role of a DSL,

as more and more domain-specific details are merged into the framework, as it is refined and maintained.

2.3.2 Building a Completely New Language

In [Bos96] two different approaches to the design of DSLs are identified: the *revolutionary* and the *evolutionary* approach. With the revolutionary approach the general-purpose language is discarded and the language designer starts from scratch. This usually results in a *little language* [Ben86], i.e., a language exclusively containing the necessary constructs for covering the domain.

Little Languages

A little language is a very high-level language with syntax and semantics close to the problem domain. Such languages are very expressive in the domain for which they are designed, allowing the programmer to state a problem more precisely in domain-specific language terms. These languages are small, and often do not contain any abstraction mechanisms like classes, procedures and types. This makes the language easier to implement, but limits its use to the domain for which it is designed. The advantages of domain-specific libraries, i.e., maintainability of the domain, portability, also apply to these languages, even if they require knowledge of compiler technology in order to design, implement and maintain the language.

Scripting languages are a well known example of little languages. Other examples include PIC [Ker82], CHEM and SCATTER [Ben86], and GRAP [BK86]. PIC is a language for drawing graphical figures. It is a fairly 'big' little language but still qualifies as such since it lacks several features found in general-purpose languages. The other examples given above, are all implemented as preprocessors for the PIC language. CHEM is used for drawing chemical structure diagrams, SCATTER is used to make scatter plots from x, y data, while GRAP is a language for generating graphical displays of data to be included in documents. These languages illustrate the fact that once "a task becomes repetitive or complicated, it is often profitable to design a language for it" [BK86, page 787]. Implementing these languages has been done by using standard tools like lex, yacc and AWK [AKW79]. Examples of development times being, for GRAP - "one long evening", and for CHEM - a week for design and implementation of the first version. They have proved to be of great practical use, saving both time and effort, and allowing tasks, that otherwise would have needed special programming expertise, to be performed by the domain experts themselves.

2.3.3 Domain-Specific Language Extensions

Even when programming within a specific domain and requiring special domain concepts there is often a need for the abstraction mechanisms found in general-purpose languages. For instance, robot programming languages started out a set of robot-specific instructions, but abstractions found in modern programming language have been found to be very useful in the more advanced applications of today

[Cra89]. Using the evolutionary approach for designing a DSL results in an extended general-purpose language, or a *heavy-weight DSEL* [Hud97] (compare to 'light-weight DSEL' in Section 2.3.1). The abstraction mechanisms and language constructs of an existing language can then be reused, and the desired domain-specific constructs needed for more efficient programming of the domain can be added. This saves having to (re)implement large parts of the language. The language designer can then concentrate on the domain-specific language extension. An example of extending a base language for real-time programming is found in Chapter 7, on page 86.

For a programmer who is already acquainted with the base language, it is not such a big step to start using a set of domain-specific constructs in that language. The language extensions are often relatively small and, if well designed, they are easily recognizable as parts of the problem domain. The actual implementation of the domain-specific entities can been done in a library or framework which is then incorporated into the language implementation. With this approach the shortcomings of the library or framework in fully reflecting the syntax and semantics of the domain can be alleviated by introducing language support for the domain entities. This is highly desirable since failure to adhere to the conventions which describe how the library is to be used leads to an increased cost in debugging the source code. And, in some application areas, undetected errors may lead to serious repercussions. For instance, if the program is to be installed in an aeroplane, or if it controls a robot, an undetected error may lead to damaging equipment or even injuring people. These conventions can be captured by extending different aspects of the base language [HB96]; the underlying object model, the static semantics, or the syntax.

Extensible Object Models

Extensible object models are a way of extending an object-oriented language by allowing more refined message passing between objects. It is a way of specifying the semantics of the object relationships which can be used for describing the semantics of domain entities. Extended object models have, for example, been used for dealing with real-time issues and database features. The *composition filters object model* [AWB+94, BTA96] is an extended object model that uses input and output composition filters which affect the received and sent messages of an object

LAYOM [Bos95a, Bos96] is an extensible object model which supports the extension of the object model by encapsulating an object with layers which filter the messages passing to and from the object. The primary principle of LAYOM is that "an entity in analysis and design models also is represented as an entity in the language model" [Bos95a, p. 120], which is what is sought after in a DSL. The extended features are made available in the language by introducing a new language construct.

The *FLO object model* [DBP95, Duc97] uses *connectors* for defining the semantics of the interaction between objects. The connectors are specified by a set of interacting rules which can be freely combined at the language level. This is allows for a more flexible use of the extensibility of the model compared to LAYOM. This flexibility is a result of the connectors not being as tightly connected to the objects.

Semantic Extension

In many cases, errors resulting from failure to adhere to conventions, can be detected at an early stage by adding static-semantic rules, or constraints, to a language. This can be done by extending the static semantics of the language with additional rules ensuring the correct usage of the domain concepts. A number of different kinds of conventions, like coding conventions, conventions connected to the use of libraries, frameworks, and design patterns [Gam95], can be captured in this way. Missing language constructs can also be simulated using this technique. In [Hed97a] such a technique is presented which is based on *attribute extension*. The base (attribute) grammar is then allowed to be extended with checks to ensure that the conventions are adhered to. Another attribute grammar based technique for extending the static semantics of a language is the use of *semantic macros* [Mad89]. They are similar to syntactic macros, but also consider the static semantics of the program and can be used to extend the static semantics, as well as the syntax, of a language.

Syntactic Extension

The syntax of a language greatly affects the amount of effort needed to write and read programs using that language, since programs are written by humans who think in terms relative to the specific problem domain at hand. Having the language reflect these concepts with an intuitive and natural syntax assists the programmer in correctly and precisely expressing a program that solves the problem. It also aids the understanding of written code since it is not cluttered with a lot of low-level details. In short, the syntax of a language can decrease the semantic gap between the problem at hand and the resulting source code of the application.

A language can be supplied with a domain-specific syntax by using *syntactic macros* [Lea66]. They are frequently used and are a convenient way of making the surface syntax more domain-specific. But, several disadvantages have been pointed out, e.g., the lack of consideration of the scoping rules of the language which may result in unwanted name clashes. *Extensible grammars* [CMA94] are an alternative to syntactic macros where the syntax of a language is extended by syntax-directed patterns. The extended grammar recognizes and respects the scoping structure of programs. Both syntax extensions and restrictions can be supported with this technique.

2.4 Problems and Solutions

The use of DSLs reaps great benefits, but also poses problems. A number of relevant research topics in this area are identified in [Nil96b], in the introduction of [ALEL96] and in the summary of [DALEL96]. In 'Little Languages, Little Maintenance?' [DK97], the question of DSL maintainability is covered in depth. We will now attempt to identify some of the problems and propose a few solutions by taking a closer look at the programming level and the language-design level of DSLs.

2.4.1 The Programming Level

The main goal for domain-specific programming is to reduce the time and effort needed to produce a correct application. This entails making it easier to read and write applications; aiding in maintenance and future development, as well as reducing the number of errors left undetected until run time. The introduction of one, or possibly several, new domain-specific languages does, however, pose some problems according to the following.

Problem 1: Unfamiliarity with the DSL

One can expect the application programmer to be unfamiliar with the actual syntax and semantics of the DSL. The reasons being irregular usage and language changes, which can be expected to be more frequent for a DSL than for a standardized general-purpose language. Providing interactive language-sensitive editing support which aids the programmer in using the correct syntax and static semantics would be very useful and relieve the programmer of having to remember the exact details of the current version of the language.

Problem 2: The User is Not a Programming Expert

The DSL programmer may very well be a domain expert, and as such be familiar with the problem domain and recognize its syntax and semantics in the DSL. This kind of domain programmer may have some programming experience, but is not necessarily a programming expert. A well designed DSL will supply language constructs that are familiar to the domain expert, but the actual programming can be further supported by an interactive language-sensitive editor which aids the domain experts in correctly using the formal language.

Problem 3: Lack of Debugging Tools for DSLs

One of the main benefits of using a DSL is increased quality, i.e., less number of errors in the produced program code. Even so, some debugging is bound to be needed. This may pose a problem, if no debugging tools are available, or the ones available use a base language to present the source code instead of the actual DSL code written by the application programmer. Thus, there is a need for debugging tools in terms of the DSL source code [DALEL96]. KHEPERA [FNP97] is a system which supports source-code debugging of DSLs by keeping track of the transformations performed on the abstract syntax tree of the program in order to produce the executable code.

Problem 4: Maintenance of a Program Using a Changing Language

The problem of maintaining an application is split into maintaining the DSL and maintaining the application [DK97]. If a domain-specific syntax is used, the appli-

cation program is usually easier to read and understand even for others than the original programmer, thus making it easier to maintain. On the other hand, it may be difficult to find someone who is fluent in the particular DSL used in the application. And, the documentation of the language may be poor or even non-existent.

Another aspect of DSL maintenance is that the language can be expected to change and evolve. The existing programs then need to be kept compatible with the current version of the language. A transformation mechanism, similar to the ones in Muir [Nor87, Win87] and TransformGen [GKL94], is needed. In both systems, a transformation template, or table, is created as the grammar is modified, and in some cases, for example, when deleting a production, the transformation needs to be manually refined in order to transform program documents according to the intentions of the language designer.

Problem 5: Unstable Language Implementation

When working with DSLs the language implementation is a potential source of errors. Undetected language implementation errors may result in very cryptic error messages and behaviour in the application program, comparable to errors in compilers. Since the application programmer usually is not familiar with the language implementation, these errors are very frustrating and hard to locate. DSL support for the language-design level is needed to ensure a correct language design and implementation.

2.4.2 The Language-Design Level

As has been mentioned, it is extremely important that the language implementation is correct since that is the platform on which the applications are to be built. In order to ensure a correct and stable design and implementation, the same support as is supplied at the DSL programming level should also be available for the language-design level. The language specifications can be seen as a program expressed in a (meta-level) DSL. By supplying DSL programming support for this level the resulting language implementation will be easier to read, write, maintain, and contain less errors. The language designer could be aided by, for example, warnings of missing grammar items, unparsable grammars, circular grammars, and ambiguity. We mentioned that good language design is hard. It would be very beneficial if the language designer could be supported in translating domain entities into ‘good’ constructs [Kam96]. A suggested approach is to first design and implement an object-oriented framework for the problem domain, and once the semantics of the domain are clarified move on to adding a (textual or graphical) language level [DALEL96, RJ97].

Problem 6: Need for Collaborative Design

The design of a language is very important since bad design can lead to severe difficulties. For example, if the language fails to cover the whole problem domain, this will limit the use of the language, and a DSL with a ‘bad’ syntax results in pro-

grams which are hard to understand. A combination of extensive knowledge of the domain and of language design is needed to obtain a 'good' language design. Since the person best acquainted with the problem domain is usually not an expert in language design, there is a need for both language and domain expertise to work in collaboration, interactively evaluating and changing the design as the project progresses. This requires an environment that supports such collaboration and, which also supports rapid prototyping of languages, giving the domain expert a chance to give important feedback on the language design at an early stage.

Problem 7: Limited Resources

The design and implementation of a DSL is typically not the main objective in system or product development. The building of the language may only employ a small part of the project resources. Also, in a smaller company the technical skills in language design and compiler technology may be very limited. Handcoding a compiler for a DSL is, thus, usually not desirable, nor possible. The time needed to design, implement, and maintain a DSL also has to be measured against the advantages, in software quality and reduced programming time, gained by using a DSL [DK97]. By applying the DSL technique and providing language-sensitive support for language development [DALEL96] these same advantages can be obtained for the language-design level. The language development phase could be further rationalized by allowing reuse of existing language designs and implementations. For example, through language extension, and through modular language specifications which allow for the construction of a language by combining a number of existing modules, or language building blocks [DALEL96].

Compiler-construction tools allow for a fast and easy implementation of a language. Yacc and lex are examples of well known, and much used, compiler-compilers. Eli [GHL+92, Kas96] is another example of a compiler-compiler which also support the construction of languages by combining reusable language components. The reuse of language components is also supported by TaLE [JKN95, KM95], a language implementation framework with a graphical user interface.

Problem 8: Incremental Language Design

DSLs are often of an experimental character, i.e., they are expanded and revised time and time again as the domain develops, as opposed to general-purpose languages which are rarely changed. Designing a language is a process where new constructs are tried and tested several times before they are 'right'. The language designer would be greatly enhanced by being supported in incrementally and iteratively designing a language in an integrated fashion [Nil96b]. Discrepancies of the language, as well as specification errors, could then be detected at an early stage in the design process by supplying immediate feedback of the changes of the language specification. Since DSLs are relatively often expanded and revised there is also a need for version control of the different revisions of the language.

Problem 9: Language Maintenance

As the problem domain evolves and new requirements arise the language needs to be revised. Changing the language implementation requires skill in language design and implementation [DK97]. At best only the language implementation is affected, but the language itself may also change. Existing programs will then need to be updated to the new language version, as was mentioned in Problem 4.

Problem 10: Portability

Programs written in a standardized general-purpose language can be used on almost any machine. This is not immediately true for programs written in a DSL. Depending on the language implementation, DSL programs may need to be ported in order to run on different platforms. Mapping the DSL to a standardized general-purpose language, or to code for a virtual machine which is implemented in a standardized general-purpose language, solves this problem.

Problem 11: Optimization

There are two different aspects concerning the optimization of DSL programs. A DSL is a high-level language which encapsulates the entities of the lower level. Full control of these entities is not available to the programmer, who can not perform, otherwise possible, optimizations. This may be a problem within some problem domains, for example, when dealing with hard real-time constraints for a system.

On the other hand, the DSL designer may be able to make optimizations based on the knowledge of the domain [Hud96]. The resulting executable DSL programs are then more efficient than programs expressed in a general-purpose language.

Problem 12: A Multi-Layered System Results in a Hierarchy of DSLs

Complex systems are often built in several layers where each layer covers a different problem domain with its own set of requirements, concepts, and language constructs [SG96]. Every layer uses its underlying layer, and provides routines for use in the next higher layer. Applying the DSL technique to such a system results in a DSL for the problem domain of each layer. Such a language is implemented in terms of the DSL of the underlying layer, and is in turn used in the implementation of the DSL of the next highest layer, resulting in a hierarchy of languages. Industrial robot programming [Nil96a] is an example of such a multi-layered system described in Chapter 6. Multi-layered grammars (see Chapter 7) can be used to describe and implement such hierarchies of languages.

2.4.3 Desired Support for DSLs

Language development requires a high degree of skill and knowledge in language design and compiler technology. In order to make the DSL technique more widely available, support for the language design and implementation phase, and for DSL programming is needed. In the previous sections we have identified a number of aspects which need to be dealt with in order to make the use of DSLs easier and safer. For the purpose of later evaluating our language laboratory, APPLAB, as a DSL tool these requirements are listed in Table 2.1. The list is not meant to be exhaustive. There are certainly additional items to add.

The Programming Level	*Problems*
1. Interactive language-sensitive programming environment	1, 2
2. Source code debugging	3
3. Version control of language specifications and programs	8
4. Transformation of programs to new language version	8, 9
The Language-Design Level	
5. DSL programming support for the language-design level	5, 7, 8
6. Rapid prototyping of languages	6, 8
7. Collaborative language design	6
8. Building a language from existing language blocks	7
9. Language extension	7
10. Multi-level languages	12
11. Translating a problem domain into a ‘good’ DSL	6
12. Portability	10

Table 2.1: Requirements for DSL support

2.5 Conclusions

A lot is to be gained by using DSLs for programming specific problem domains. The amount of programming effort required is reduced, both initially, and when debugging and maintaining applications. A number of serious errors can be statically detected by introducing static-semantic rules in a DSL. This greatly reduces the amount of work required to ensure that a program is safe to execute.

On the other hand, effort needs to be put into designing, implementing and maintaining a DSL. This is no simple task. It requires a lot of skill and technical know-how. It is a trade-off between the benefits of using a DSL and the amount of

work needed to design, implement and maintain such a language. In order to simplify the language design and implementation process, additional support of the DSL method is needed. By applying the DSL technique to the language-design level the same advantages, (i.e., reduced 'programming' time, and increased quality), are also achieved for language development. The language designer would be further assisted by an environment that supports the design process in an integrated, iterative and incremental way. Such language design and compiler technology support would make the DSL technique available to a wider audience.

A list of requirements for DSL support has been presented. Existing technology can be used to implement some of the requirements, while additional research is needed for others. In Chapter 8 the APPLAB system is evaluated against this list.

Chapter 3

APPLAB -

A Language Laboratory

APPLAB is an interactive environment for the design of domain-specific languages, developed as part of this thesis work. This chapter gives an overview of the system, while the two following chapters contain more detailed descriptions of the static-semantic and text editing facilities of APPLAB.

A number of different DSLs have been implemented using APPLAB. For example, a robot programming language has been designed and implemented, and APPLAB is used as a programming interface to an industrial robot. The meta grammars used to describe the grammars which the language designer uses to define a language in APPLAB, are also examples of DSLs. The robot case study is described in Chapter 6.

3.1 Introduction

Designing a new language is an iterative process where language constructs are designed and tested in a prototypical fashion. In particular, DSLs may need to evolve rapidly due to new requirements that emerge as the domain develops. Compiler-compiler tools are useful when implementing a fully designed language, but while designing a new language, or changing a language, it is desirable to work in an environment that supports the language design process in an integrated and incremental way. We call such an environment a *language laboratory*.

We have developed and used a language laboratory called APPLAB (*APPlication language LABoratory*) [Bja96] which is based on the principle of *immediate computation* [RT87], meaning that the necessary computations are performed immediately, and automatically, by the system. Immediate computation as such can be found in systems like electronic spreadsheets, WYSIWIG word processors, and language-based editors. In APPLAB, immediate computation is used both to support the interactive development of a DSL specification, and the interactive language-based editing of programs in the changing DSL. The language designer

can switch freely between editing the language specification and editing a program in the (changing) new language. The effects of changing the language specification are immediately seen in the edited program without requiring the user to issue any special updating commands.

APPLAB is based on an earlier language laboratory developed as part of the Mjølner/Orm environment [MHM+90, MM94]. Other systems with similar aims of interactive support for language design include DOSE [FJS86], Muir [Win87], the ASF+SDF Meta Environment [Kli91], and TaLE [JKN95, KM95].

The rest of the chapter is organized as follows. Sections 3.2 and 3.3 give an overview of APPLAB and describe its architecture. This is followed by a discussion in Section 3.4 of how the principle of immediate computation is applied in the system. In order to give the reader an idea of what it is like to use the system, an example of designing a language in APPLAB is given in Section 3.5. A few case studies performed using the system are described in Section 3.6. Related systems are discussed in Section 3.7. The current status and future work of APPLAB are identified in Section 3.8, and, finally, the chapter is concluded in Section 3.9.

3.2 The APPLAB System

APPLAB is a language laboratory based on language-sensitive editing, object-oriented attribute grammars, and the principle of immediate computation. The editing environment provides structure-oriented editing, semantic editing (explained later), incremental parsing, static-semantic checking, and code generation.

3.2.1 Grammar Interpretation

The most distinguishing characteristic of the APPLAB system is that the same editing environment is provided for simultaneous editing of both the language specification (i.e., the grammar) and programs written in the specified (changing) language. The immediate updating of the program editor, as a result of changes to the language specification, is accomplished by *grammar interpretation* [Min90], where generic editing tools interpret the grammar and other data structures. This distinguishes APPLAB from traditional compiler-compiler systems such as Yacc [Joh79], the Synthesizer Generator [RT89], and Eli [GHL+92], which are based on *generation*. That is, compilers and editors are built by compiling and linking source code generated from language specifications.

The editing style provided by APPLAB is particularly well suited both for the design and use of DSLs. While designing a new language, APPLAB allows the language designer to work in an experimental fashion. The grammar-interpreting features provide an instant feedback on new grammar rules, making it possible to try out new language constructs interactively as they are defined. APPLAB also aids the user in writing programs with correct syntax and with notification of static-semantic errors. This is very useful for an application programmer since DSLs are often changed and expanded, and are often used on an infrequent basis by domain experts with less computer-programming experience.

3.2.2 Editing

The primary editing style supported by APPLAB is *structure-oriented editing*. The user can expand placeholders in the program by selecting language constructs from a menu of all syntactically legal constructs at the current editing focus.

The system also supports *text editing* by allowing any syntactic unit (corresponding to a subtree of the syntax tree) to be edited as text and subsequently parsed. The parser is implemented by a grammar-interpreting technique [BH96] and is kept up to date as the language specification is changed.

An additional, editing style supported in APPLAB is *semantic editing* [Hed92b]. By allowing the definition and usage of menus based on the static-semantic information of a program the programmer can be aided in locating the desired and correct name directly in the program editor. Such a menu is called a *names-menu*. It may contain, e.g., all declared names, all used but undeclared names, or all type-correct names at a certain point in the program. Choosing a name from such a menu means that it is inserted into the program at the current editing focus.

Another kind of advanced editing support provided by the APPLAB system is *static-semantic error reporting*. The system notifies the programmer of static-semantic errors by marking the source of these errors in the program-editing window. The user can then choose to ignore the error or correct it. An example of static-semantic error reporting in APPLAB, is shown in Figure 3.1. The dotted lines around parts of the program indicate static-semantic errors. Upon requesting an explanation of such an error a message is displayed, provided one is defined.

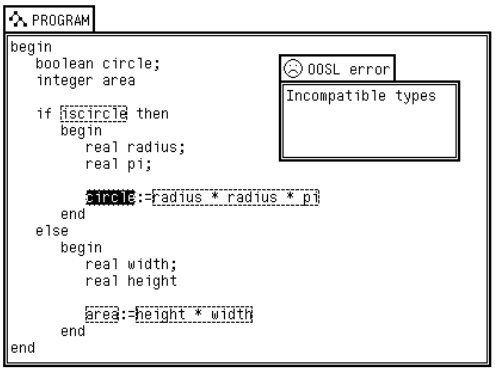

Figure 3.1 An example of static-semantic error reporting in APPLAB.

3.2.3 Static Semantics and Code Generation

Static-semantic checking and code generation can be specified using *OOSL* (Object-Oriented Specification Language), an object-oriented attribute grammar formalism [Hed92a]. In the current version of APPLAB, attribute evaluation is done by a *demand-driven attribute evaluator*. That is, the attributes are not stored in the syntax tree, but are computed whenever their values are demanded, for example, when the user explicitly asks for the value of a particular attribute of a

syntax node, or when the semantic editor asks for attributes to build the menu of visible identifiers. The demand evaluation mechanism is simple, but general in that it can evaluate attributes in any non-circular attribute grammar, and has been of great practical use. We have used it to implement semantic-editing support, demand-driven static-semantic checking (type checking), and code generation (generating C-code from DSLs) without efficiency problems. For example, generation of C-code for a 100-line robot program took 1.7 seconds.

3.2.4 Grammar Aspects

A language is specified in APPLAB in a grammar document containing several *grammar aspects*. Each aspect is displayed in a window as shown in Figure 3.2.

- The ABSTRACT aspect defines the abstract context-free syntax for the language, using a structured BNF notation, i.e., each production is either an AND-rule or an OR-rule.
- The CONCRETE aspect defines the concrete syntax, i.e., keywords and indentation defining the textual representation of an abstract syntax tree.
- The PARSE aspect defines additional rules necessary for parsing text into the abstract syntax trees described by the ABSTRACT aspect, for example, rules for operator precedence and associativity which resolve the ambiguities of the ABSTRACT aspect.
- The OOSL aspect defines an attribute grammar which can be used for, e.g., static-semantic checking and code generation. In one language definition, there may be several OOSL aspects used for different purposes.

The design choice to split the grammar representation into several aspects is partly due to historical reasons, and the specification currently contains certain redundancies. For example, the abstract syntax in the ABSTRACT aspect actually has to be repeated in the OOSL aspect, but in a slightly different form.

In addition to the grammar aspects, a grammar document may contain PROGRAM windows which contain example programs following the (changing) language specification. It is also possible to have programs in separate documents which follow the specification in a given grammar document. In fact, the different grammar aspects in a grammar document follow the language specifications of meta grammars given in other grammar documents.

Each grammar aspect or program is represented internally as an abstract syntax tree (AST). All the internal tools in APPLAB (the different editing facilities and the attribute evaluator) are implemented as generic tools which work on a program AST, interpreting grammar ASTs to provide language-specific behaviour. For example, the structure-oriented editor interprets the ABSTRACT and CONCRETE aspects to support structure-oriented editing for programs. Since the grammar aspects are represented in the same way as programs, namely by ASTs, all the internal tools work in the same way for the grammar aspects. For example, the structure-oriented editor supports editing of the ABSTRACT aspect by interpreting a meta grammar defining the language used for the ABSTRACT aspect.

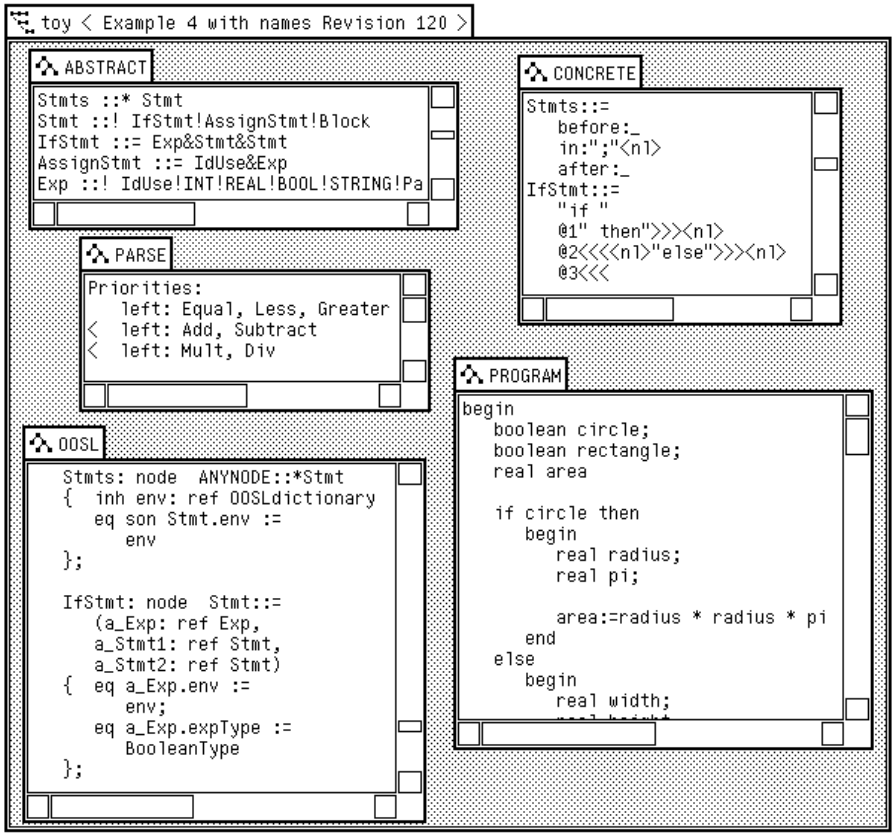

Figure 3.2 A grammar document 'toy', containing different grammar aspects and an example program.

3.2.5 Implementation Status

APPLAB is implemented in Simula [Sta87] and runs on SUN workstations. The compiled system consists of 3 MBytes of which 1.8 MBytes were 'inherited' from the grammar editor of the Mjølner/Orm system. It is an 'industrial prototype', i.e., it is has sufficient functionality for evaluation and for use in real-sized case studies, but it is not sufficiently robust for production usage. The system is available free of charge for research and evaluation purposes. The largest case study done with APPLAB is the implementation of a robot programming language (212 production rules). In addition, we have performed a number of smaller case studies (see Section 3.6), and all the formalisms for the different grammar aspects are themselves DSLs developed in APPLAB. The study of robot programming is discussed in more detail in Chapter 6.

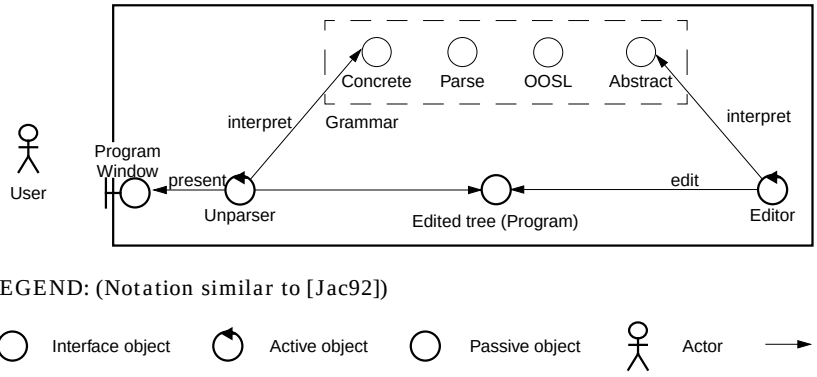

Figure 3.3 Base architecture of APPLAB, program editing.

3.3 The APPLAB Architecture

3.3.1 The Base Architecture

The base architecture of APPLAB is shown in Figure 3.3. The main components are the Editor, the Unparser, the Edited (Syntax) Tree, and the Grammar. The Editor is responsible for changing the Edited Tree (`edit`), guided by the Abstract aspect of the Grammar (`interpret`). The Unparser administers the presentation of the Syntax Tree on the screen (`present`), guided by the Concrete aspect of the Grammar (`interpret`).

Each of the grammar aspects is itself represented by a syntax tree, and is edited and unparsed according to a meta-grammar. Thus, there are actually five instances of the base architecture as shown in Figure 3.4. Each instance of the Editor, and the Unparser, interprets the base Grammar of the edited grammar or program.

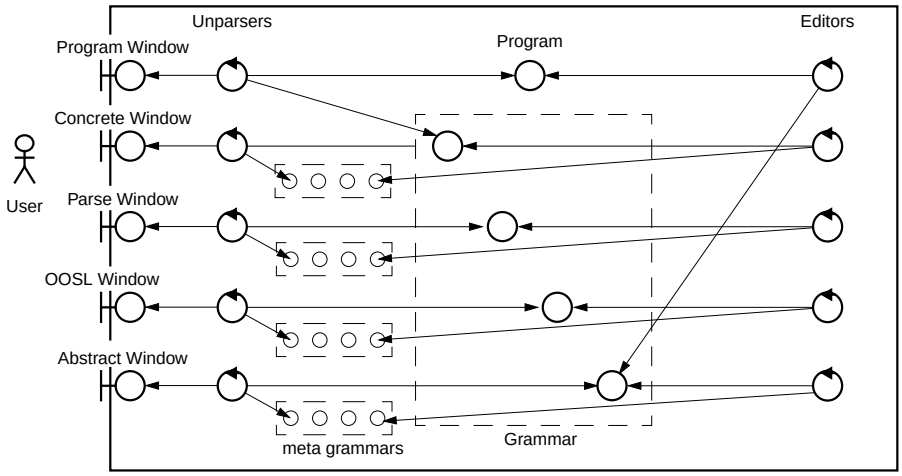

Figure 3.4 Base architecture including meta grammars.

Editing grammars can therefore be done in the same way as editing programs. The grammar editor is also structure-oriented with text-editing facilities and, if static-semantic rules have been defined for the grammar (in the meta grammar), supports the user in defining static-semantically correct grammars, for example, that all used names are defined.

3.3.2 The Parser Component

The base architecture is augmented by the parser component, GRIP, as shown in Figure 3.5. When given a text string by the User (`parse text`), GRIP parses it into a parse tree guided by the Abstract, Concrete and Parse aspects of the Grammar (`interpret`). If the parse was successful the resulting parse tree is passed to the Editor to be inserted into the Edited Tree (`insert`), and the Unparser is called to update the screen (`update`). Upon encountering errors an error window appears presenting the located errors (`report`). The User is then prompted to re-edit the text in the Text Editing Window.

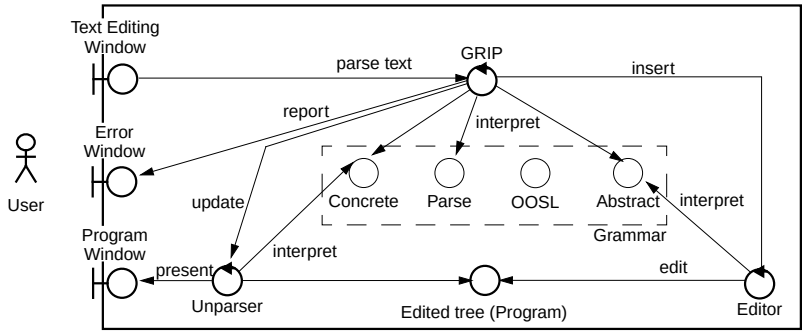

Figure 3.5 Base architecture of APPLAB including the parser component, GRIP

GRIP consists of the following internal components (see Figure 3.6): a Controller which coordinates different actions, a Parser which performs the actual parsing, a Lexer which tokenizes the input text, a Grammar Graph which is a representation equivalent to the grammar but in a form more efficient for interpretation during parsing, and a Grammar Checker which checks that the grammar is possible to parse from and which constructs the Grammar Graph from the grammar. See Chapter 5 for further details on the parser component.

3.3.3 The Demand-Attribute Evaluator

The static-semantic rules and code generation of a language, defined in the OOSL aspect of a base grammar, are applied by calculating the values of the attributes of the syntax nodes. The Demand Attribute Evaluator performs this evaluation by interacting with the basic components of APPLAB, as shown in Figure 3.7. The attributes are evaluated guided by the OOSL aspect of the base grammar and the currently edited syntax tree (`interpret`). The Demand Attribute Evaluator can

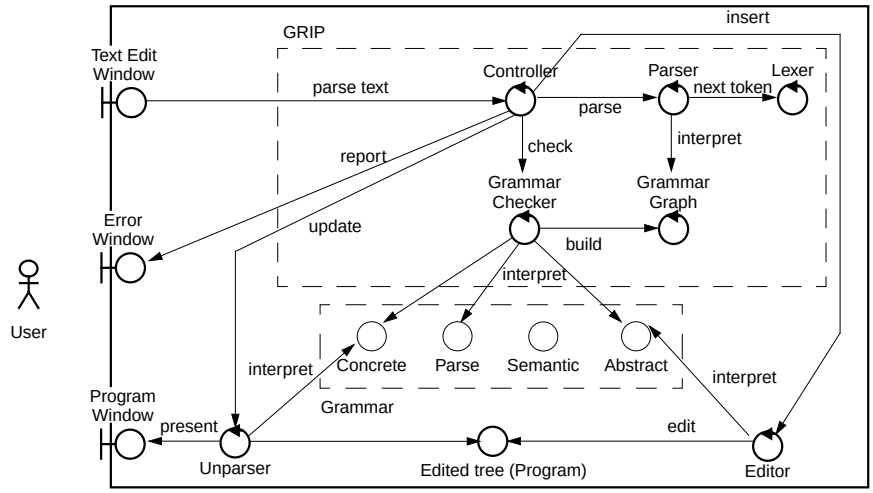

Figure 3.6 The parser component, GRIP

be invoked both by the editor and by the user (`evaluate`).In the latter case the resulting value of the attribute is displayed in a window (`present`). The semantic editing features of the Editor are supplied by evaluating the static-semantic attributes of the grammar, for example, the `error`-attribute. The Editor then uses the evaluated attribute to supply the user with static-semantic information. For example, upon evaluating an `error`-attribute to true the Editor requests that the Unparser sets an error marker at the position of the static-semantic error in the Edited Tree (`mark`).

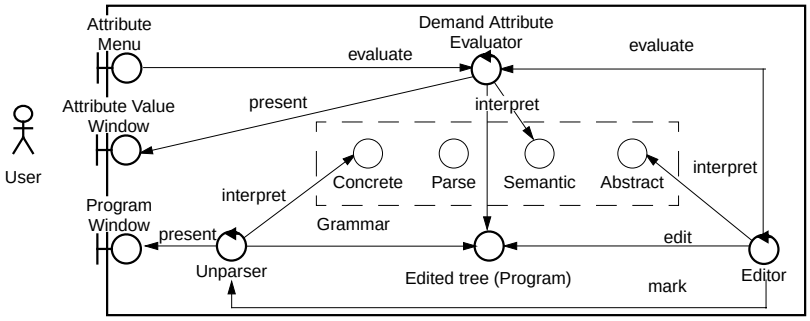

Figure 3.7 Base architecture of APPLAB including the demand attribute evaluator.

The Demand Attribute Evaluator consists of the following internal components (see Figure 3.8): a Controller which coordinates different actions, an Evaluator which computes the current value of an attribute, an Evaluation Structure which is a compiled representation of the OOSL aspect of the grammar allowing a more efficient evaluation of the declared attributes, a Compiler which interprets and checks the OOSL aspect of the grammar and which constructs the corresponding Evaluation Structure. See Chapter 4 for further details on the static-semantic component of APPLAB.

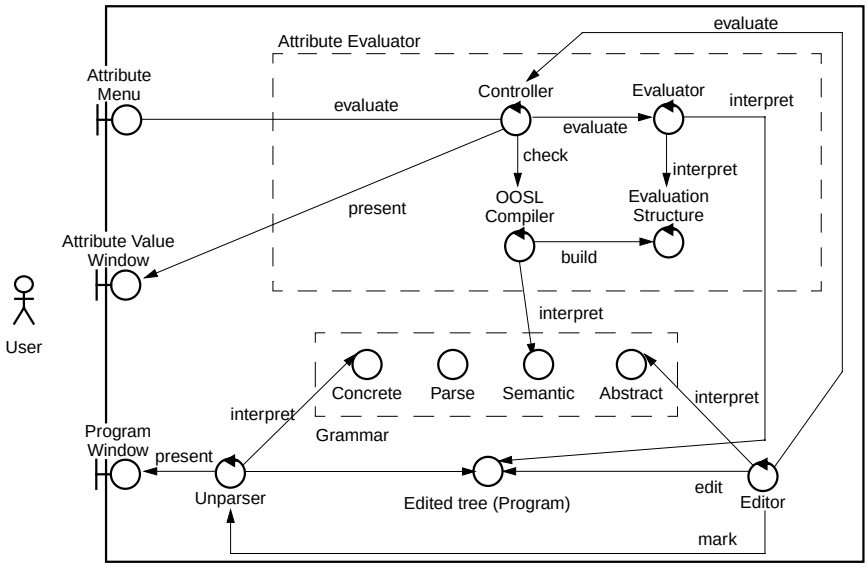

Figure 3.8 The attribute evaluator

3.4 Immediate Computation

The APPLAB system uses *immediate computation* as its prevailing design philosophy. This means that the system automatically performs all the necessary computations needed to keep the internal structures of the system up to date, without requiring that the user issues any explicit updating commands. This is comparable to changing the data in a spreadsheet. The values of the affected cells are immediately, and automatically, recalculated by the system. A similar example from APPLAB is the effect of adding a new language construct. Assume that the language contains constructs like '`while`' and '`if`', and that the language designer adds an additional construct '`repeat`' by adding grammar rules in the ABSTRACT and CONCRETE aspects. Programs can then immediately be edited using the new language construct; the '`repeat`' construct will appear in the structure-editor menus along with the previously existing '`while`' and '`if`' constructs, and when text editing, the system can automatically recognize also '`repeat`' statements.

Whereas functionality based on grammar interpretation are evaluated immediately as mentioned, other computations are delayed until the computed data is actually needed. For example, the structure-editing menus and the parser are updated when the user invokes them. These late and event driven computations (called demand driven in the sequel), like in other window-based applications, only result in a very short delay, normally unnoticable by the user.

We will now take a closer look at how the different components of the system utilize a combination of demand-driven and immediate computation.

3.4.1 Structure-Oriented Editing

The structure-oriented editor interprets the ABSTRACT aspect in order to build menus for structure editing. When the ABSTRACT aspect is changed by the user, these menus need to be rebuilt. This is done, as mentioned above, when the user wants to display a structure-oriented editing menu in a PROGRAM window. For a grammar with 100 productions, this takes 0.4 seconds on a Sun Ultra 1.

When the system carries out an editing command, i.e., the construction of a program structure as a response to a structure-editing command, the ABSTRACT aspect is interpreted directly, and the time for this construction is thus not affected by changes to the language specification.

The AST for a program is displayed as text by interpreting the CONCRETE aspect. When the CONCRETE aspect is changed, the programs need to be redisplayed. Currently, this is not done for each incremental change of the CONCRETE aspect, partly because it might disturb the user if other parts of the display are updated while his/her attention is focused on the CONCRETE aspect, and partly because it would result in the contents of the PROGRAM windows to be in several intermediate uninteresting states. Instead, a PROGRAM window is redisplayed the next time the user sets the editing focus in that window (by clicking in it). For a 100 line program, such a redisplay takes 0.2 seconds.

Changes to the ABSTRACT aspect may cause existing PROGRAM ASTs to become inconsistent with the specification. Currently, we only have an ad-hoc transformation mechanism which can handle some simple non-information-losing transformations, and takes 0.3 seconds for a 100 line program. Other inconsistencies remain in the program ASTs and although the system can handle most of them without crashing, this may cause problems later on in the programming process. This is clearly not satisfactory, and needs to be dealt with in a more systematic way in future versions of APPLAB. A mechanism is needed for transforming a program AST into a consistent version based on a user-supplied specification, by for example using techniques like those in the TransformGen system [GKL94] or Muir [Nor87].

3.4.2 Text Editing

During parsing, APPLAB interprets the current grammar represented by a data structure generated from information in the ABSTRACT, CONCRETE, and PARSE aspects. This is done to achieve a higher degree of efficiency during the actual parsing than if the grammar aspect ASTs had been interpreted directly. The parsing technique and data structures are discussed in more detail in Chapter 5. The generation of the data structure is demand driven. That is, when the user invokes the text-editing facilities the system checks to see that the data structure used by the parser is up to date, and if not, it is recomputed to conform to the current version of the grammar. This takes 0.3 seconds for a grammar with 100 production rules.

3.4.3 Attribute Evaluation

The demand-driven attribute evaluator of the APPLAB system interprets a data structure generated from information in the OOSL aspect, as described in more detail in Chapter 4. This data structure needs to be regenerated when the OOSL aspect has been changed. The regeneration is handled similarly to the text-editing case. When the user asks for the value of an attribute, invokes a semantic-editing command, or asks for code to be generated based on an OOSL grammar, the system checks to see that the OOSL data structure is up to date, and if not, it is recomputed to conform to the current version of the OOSL aspect. This takes 15 seconds for a 100 production OOSL grammar.

The advanced editing support described in Section 3.2.2 has been implemented by introducing special attributes. The contents of a names-menu is specified in a `names` attribute while the static-semantic error reporting is supported by attributes called `error` and `errorMsg`. The reporting of static-semantic errors is currently not done by using immediate computation since the attribute evaluator only works on demand. By implementing an incremental evaluator for Door AGs [Hed92a] which can be described in OOSL, one could obtain immediate computation of static-semantic error messages, as the program is changed. As reported in [Hed94] this updating can (for most programs) be done fast enough so as not to be noticed by an interactive user. Using a Door AG evaluator would imply a small delay for computing evaluation plans after changes to the OOSL aspect. The use of stored attributes would also imply that an attributed program AST could become inconsistent when the OOSL aspect for a Door AG is changed. The easiest solution would be to simply recompute the complete program attribution according to the new OOSL version.

3.5 Designing a New Language in APPLAB

When designing a new language in APPLAB the language designer can work in an iterative fashion, working on a sub-set of the language and gradually extend it to implement the whole language. Once the abstract syntax of a language sub-set, however small, has been defined, it is possible to work on the different aspects of the language, like the concrete syntax, the text-editing facilities, the static semantics and the code generation. This can be done in any order, and in an iterative fashion, all the time watching the resulting changes in a PROGRAM window.

3.5.1 Defining the Abstract and Concrete Syntax

When designing a new language in the APPLAB system one starts by defining the abstract and the concrete syntax. Once the syntax of a new language construct is specified, such a construct can be created and inspected in a PROGRAM window. If the language designer is not satisfied with, for example, the concrete syntax it can be edited and the resulting change is immediately shown in the PROGRAM window. Figure 3.9 shows the start of a new language. The abstract rule for `Stmt` and the concrete rule for `IfStmt` are highlighted. The focus of the PROGRAM win-

dow is set on a `Stmt`-construct, and when choosing **Expand** in the menu of the PROGRAM window another (sub-)menu appears, containing exactly all the available kinds of `Stmt`:s. The syntax defined for the `IfStmt` in the concrete aspect is used both in the expand-menu and in the PROGRAM window. If no concrete rule has been defined for a language construct a default syntax is used, as for the `Real` and `Boolean` constructs used in the variable declarations of the PROGRAM window.

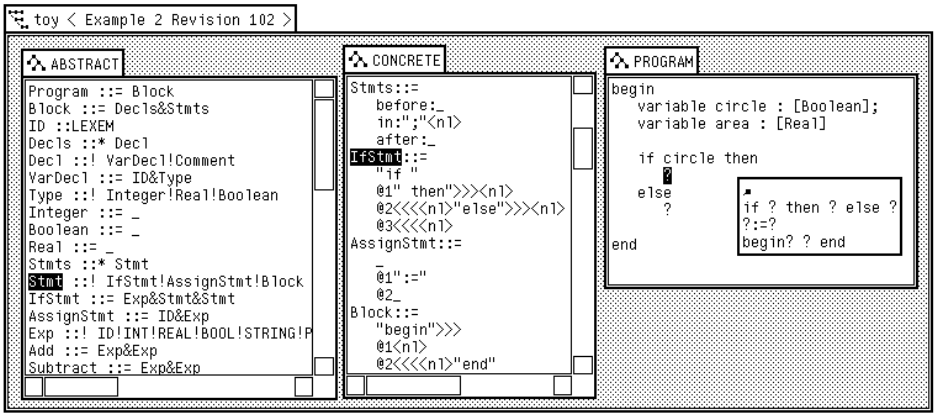

Figure 3.9 The ABSTRACT and CONCRETE aspects of a new language describe the language shown in the PROGRAM window, and in its Expand-menu.

3.5.2 Adding Support for Text Editing

Once the abstract and concrete grammar aspects have been specified the structure-oriented editor for the new language is complete. Some small, very simple languages can be edited as text without any further specifications. But, for most languages further specifications are needed since the structure-oriented editor allows an abstract grammar to be ambiguous. For example, the two (different) syntax trees:

- `add(a,subtract(b,c))`, and,
- `subtract(add(a,b),c)`

are both displayed using the same textual representation "`a+b-c`". Further rules, apart from the ones specified in the abstract and concrete aspect, are needed in order to resolve such ambiguities. This is done in a PARSE window where the associativity and precedence of binary operators are specified. Certain lexical information, like comments and strings, can also be specified by the language designer.

Figure 3.10 shows our example language with a PARSE window added. The precedence of the binary operators of the language are given, where rules on lower lines have precedence over rules on upper lines (indicated by <). For example, in our language `Mult` is given higher precedence than `Equal`. In the figure the user has chosen to text edit the highlighted expression. Extra parentheses have been

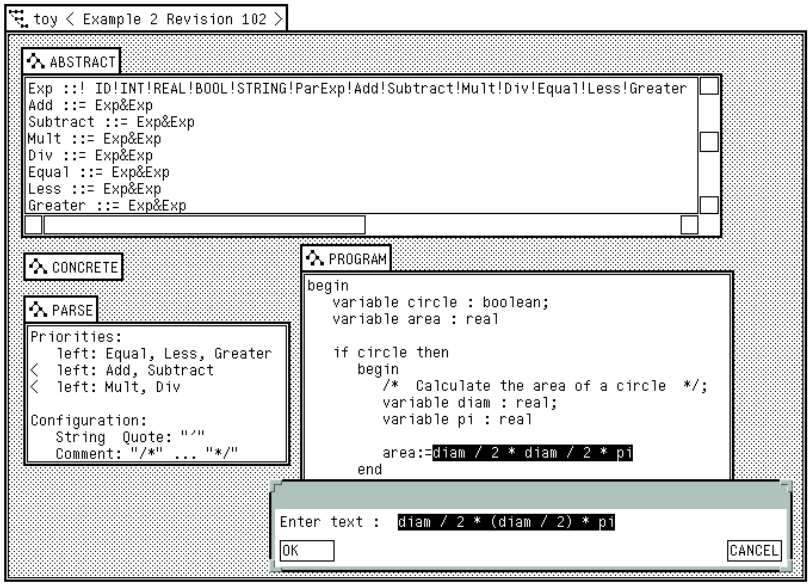

Figure 3.10 The PARSE aspect is used to specify the disambiguating information needed to correctly text edit a language.

added in the text-editing window, according to the precedence of the operators, so as to ensure an equivalent parse tree if the user returns an unedited text.

The lexical syntax recognized by the parser component of APPLAB is currently not completely user-defined. The keywords and operators are defined in the concrete aspect, while identifiers and numbers are pre-defined. The start- and end-tokens of strings and comments can be specified in the PARSE window. In our example language, strings are enclosed within the character ‘, while comments are surrounded by the tokens /* and */.

If any errors are encountered while text editing, an error window is presented, as in Figure 3.11, reporting the positions and types of errors, and the user is prompted to correct the error, or cancel the text editing operation. No changes are made in the actual PROGRAM document until a text editing operation is successfully completed. Errors in the language specification are also presented in the error

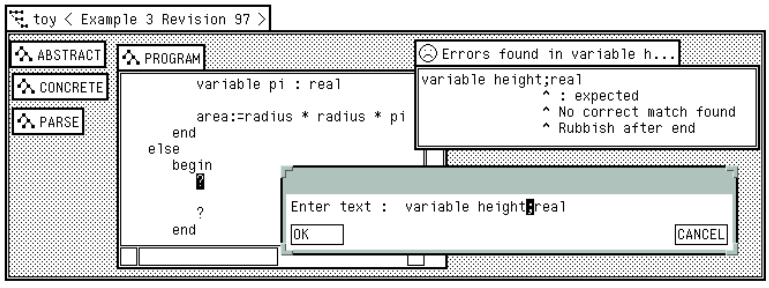

Figure 3.11 Language-sensitive text editing. The detected errors are reported and the user prompted to re-edit the text, or cancel the text-editing operation.

window. For example, if the concrete syntax has not been defined for a language construct, or if an undefined grammar rule is used.

3.5.3 Adding Static Semantics

Static-semantic features can be added to the structure-oriented editor for a language by adding an OOSL window containing an object-oriented attribute grammar expressing the static-semantic rules of the implemented language. Attributes and equations defining those attributes can be declared for each grammar rule of the ABSTRACT window. In particular, a few special attributes can be used to trigger static-semantic actions in the structure-oriented editor. The attribute `names` is an example of such an attribute. A `names`-attribute can be defined to specify the contents of a names-menu, a context-sensitive menu from which the application programmer can select a string (a name) to be inserted into the program. Figure 3.12 shows part of the OOSL aspect of our example language and the names-menu specified for the identifier part of an `assign`-statement. The names-menu contains all available declared identifiers of the program according to the scope-rules of the language. In the OOSL window, `AssignStmt` is specified as a kind of `Stmt` which has two parts, an `IdUse`- and an `Exp`-node. All different kinds of `Stmt`:s have an `env`-attribute containing all names of the current scope. The value of this attribute is, in `AssignStmt`, passed to the `names`-attribute of the `IdUse`-node, thus defining the names-menu of the left-hand side of an `assign`-statement to contain all the names of the current scope. The result is seen in the names-menu in the PROGRAM window.

While implementing and debugging the OOSL aspect of a language the user can evaluate the attributes of the programs by selecting the desired attribute from a sub-menu in the PROGRAM window. This sub-menu contains all the available attributes of the current editor focus.

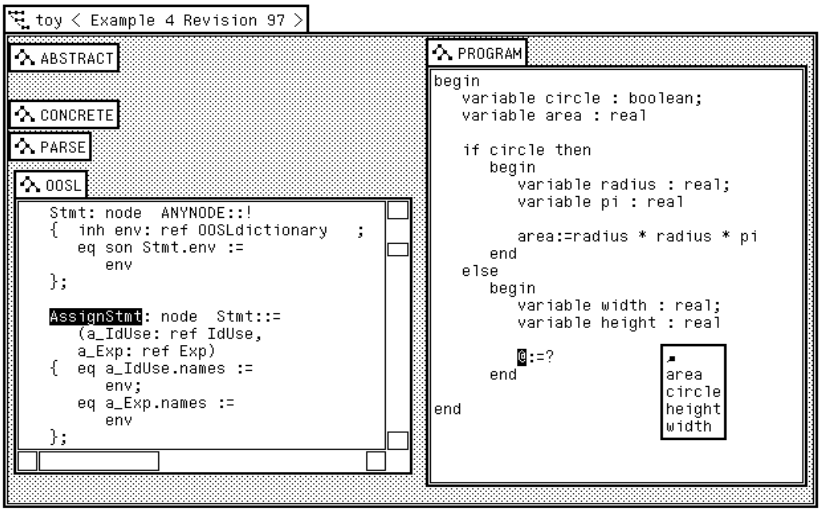

Figure 3.12 The names-menu of the current editor-focus.

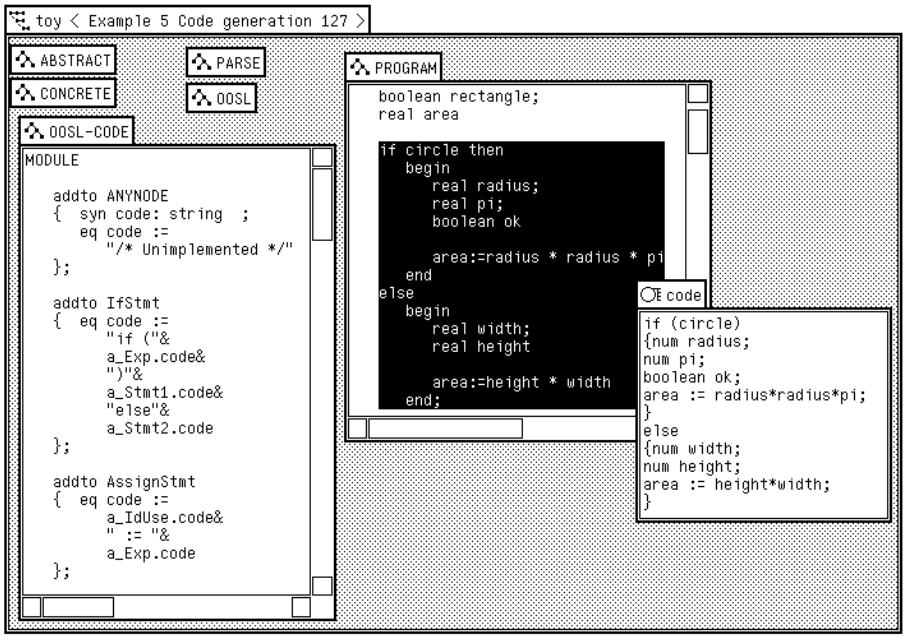

Figure 3.13 Code generation (in terms of C code) is specified in the OOSL-CODE window. The code window contains the generated code for the selected part of PROGRAM.

3.5.4 Specifying the Code Generation

The code generation of a language can be specified using the same mechanisms as for the static semantics. That is, by declaring an attribute for the generated code and equations defining its contents in an OOSL window. This could be done in the same OOSL window as used for the static-semantic rules but in our example we have chosen to use a separate window called OOSL-CODE for this purpose. In Figure 3.13 the translation of an *if*-statement to (unformatted) C-code is shown in the OOSL-CODE window. The attribute code contains the generated C-code and each language construct has an equation describing how to generate the corresponding C-code. Of course, the code generation could produce any other type of (textual) code, it is not limited to handling C-code.

3.5.5 Designing a Complete Language

By iterating over the different language aspects for the desired language constructs, all the time inspecting the resulting changes to the syntax (abstract and concrete), static semantics and code generation, a new language evolves, aspect by aspect, and construct by construct. As the language specification grows there is a need for debugging support. The currently available support is provided mainly by the parser component and the OOSL evaluator which detect grammatical errors,

like inconsistencies between grammar windows, circular attribute grammars, and ambiguous grammars. Also, it is possible to debug the OOSL grammar by evaluating the different attributes of any construct of a PROGRAM window, checking the resulting values of the declared attributes. The currently available debugging support is mostly a result of ad hoc development. In the future, it is desirable to design and implement debugging tools especially aimed at language design and implementation.

3.6 Case Studies Using APPLAB

In addition to the large case study on robot programming (see Chapter 6), APPLAB has been used in a number of smaller case studies including two master thesis projects. One such project carried out at SAAB Military Aircraft included integrating APPLAB with the Verilog CASE environment which supports graphical development of real-time systems [Ekb96]. APPLAB is used to view and edit the finite state machine specifications, thus integrating structure-based editing into a graphical environment. In another master thesis project [Nyb97], APPLAB was used for allowing the same robot program to be edited in two different languages. The project involved translating a program from RAPID to Karel, and back again (described in [Nil96a, pp 41-44].

Another very interesting case study is to look at the grammars used to describe the languages in APPLAB. They are in actual fact themselves described by (editable) meta grammars. We will now take closer look at these grammars.

3.6.1 Meta Grammars

The grammars the language designer uses to define a new language are examples of DSLs. They are usually small languages with 10-20 grammar rules, but can also be bigger. These grammars are described in the system by (meta-) grammars and can be edited just like any other language. For example, the concrete syntax of a grammar can be changed by editing its meta-grammar. Due to the generic nature of APPLAB's internal tools the same language-sensitive editing support as is provided for programs, e.g., text editing and semantic editing, can also be provided for grammars. Such support has been added to the abstract meta grammar. The reasons for selecting the abstract meta grammar were that it is a 'simple' grammar and that it is the primary grammar aspect; all other aspects have to conform to it.

The Abstract Meta Grammar

Text editing was supported right away by the generic parser. No additional specifications needed to be made since the abstract meta grammar is not ambiguous and does not contain any strings or comments.

Semantic editing was supplied by defining two different names menus. One containing all defined nonterminals and one containing all used, but undefined nonterminals. Interactive static-semantic error reporting was also implemented for the abstract meta-grammar by checking that each used nonterminal is defined.

The two defined names-menus were of great assistance, even when working with small languages containing 5-10 production rules. They save a lot of typing and allow the language designer to instantly see if there are any additional nonterminals which need defining. Any static-semantic errors that occur when a grammar rule is deleted or manually edited are caught and reported by the interactive static-semantic error reporting.

In the future, it is desirable to provide advanced editing support also for other meta grammars used in the APPLAB system. Text editing can be supplied for all meta grammars using the current version of the system. But, the advanced editing features based on static-semantic rules require non-local access to the abstract grammar aspect since the other aspects must conform to the abstract grammar defined for the edited language.

3.7 Related Systems

Relation to Mjølner/Orm

APPLAB is the result of further development of an earlier language laboratory which is part of the Mjølner/Orm environment [MHM+90, MM94]. The grammar-interpreting structure-oriented editor [Min90], as well as the version control [Gus90] of the Orm system were inherited by APPLAB. The new contributions for the APPLAB system are

- Integrated grammar-interpreting static semantics.

  Orm supports incremental static semantics of object-oriented languages based on an early version of Door AGs [Hed94]. The static-semantic support of APPLAB is more powerful in that the static-semantic component is generic (which it is not in Orm), but less powerful in that the evaluator works on demand and does not support object-oriented languages. The grammar-interpreting features of APPLAB allow for rapid prototyping of the static semantics of a language and of the advanced editing features. This is not the case for the Orm system which requires the use of external tools in order to obtain a static-semantic component for a certain language. The object-oriented AGs used in APPLAB support block-structured languages. The incremental static semantics of the Orm system is more efficient than the demand-driven attribute evaluation performed by APPLAB. For example, the static-semantic error reporting which is done on demand in APPLAB (described in Section 3.2.2) is performed continuously in the Orm system, immediately notifying the user of occurring static-semantic errors. The static-semantic support of APPLAB is further described in Chapter 4.

- Grammar-interpreting parsing.

  Text editing of expressions has been handcoded for Simula in the Orm system. In APPLAB text editing of any language structure (of a program or grammar) is supported by a grammar-interpreting parser component. This

also supports the import of programs in textual form into the APPLAB system. The parser component of APPLAB is further described in Chapter 5.

- Grammar-interpreting semantic editing.

  The Orm system contains handcoded hierarchical names-menus for Simula. The names-menus of APPLAB (described in Section 3.2.2) are nonhierarchical, but implemented by using the grammar-interpreting static semantics. The contents of the names-menus is, thus, specified by a grammar and not handcoded, as in Orm.

Other Systems

Other systems which also aim at interactive support for language design include DOSE [FJS86], Muir [Win87, Nor87], the ASF/SDF Meta Environment [Kli91], and TaLE [JKN95, KM95]. Of these, DOSE and Muir were research prototypes which are no longer available.

DOSE, Muir, and the ASF/SDF Meta Environment are all similar to APPLAB in that they provide generic structure-oriented and/or text editors working for both language specifications and programs, with immediate construction of the program-editing environment as the language specification changes. None of these environments, however, support semantic editing. The support for semantic checking and code generation is also different in these environments: DOSE used an approach based on action routines and Muir only had an experimental semantic component based on attribute grammars. The ASF/SDF Meta Environment contains semantic support based on algebraic specifications. APPLAB, thus, differs from the other environments in its support of semantic editing, static-semantic checking, and code generation based on object-oriented attribute grammars.

Neither DOSE, nor the ASF/SDF Meta Environment has (to our knowledge) version control for grammars and programs, or support for transformation of programs after changes to the grammar, or techniques for dealing with inconsistencies. The Muir system had an advanced systematic technique for handling transformations of programs after changes to the language specification, but only an experimental rudimentary version-handling mechanism where production rules and syntax nodes could be marked by a version number. APPLAB has full version control for grammar and program documents, keeping track of older versions of documents, and bindings between program and grammar versions (and between grammar and meta grammar versions), but only very limited support for program transformations.

The TaLE system is also a system supporting the interactive development of languages, but in quite a different way compared to the systems mentioned above. Rather than providing structure-oriented editing environments for various language specification formalisms, TaLE is a highly specialized editor, partly based on graphical representations of the language constructs. An object-oriented representation of the language constructs is used to provide reuse, and built-in support for standard programming language features on the lexical, syntactic, and semantic level is supplied. However, the interactivity is provided only at the language-spec-

ification level; similar to traditional compiler-compilers, TaLE generates source code for parsers and compilers.

3.8 Current Status and Future Work

Currently, new languages can be defined and edited in APPLAB, as well as programs expressed in those languages. The structure-oriented editor has been augmented to also allow textual editing. This was done by adding *a Grammar Interpreting Parser*, GRIP (see Chapter 5) to the system. The addition of a parser component also makes it possible to import programs into APPLAB from other systems via a textual representation.

The support for static semantics is, at present, supplied by a *demand-driven attribute evaluator* for standard attribute grammars. In the future it is desirable to implement *incremental* attribute evaluation. Door AGs allow for efficient incremental evaluation of attributes with non-local dependencies, and is especially suited for object-oriented name analysis. We wish to support full, incremental, Door AGs in the APPLAB system. Once this has been implemented, more advanced editing features can be added. For example, the marking, and unmarking, of static-semantic errors can be done automatically in the editor, instead of letting the programmer request the current status of the error markers. The names-menu facility, as described in Section 3.5.3, on page 34, is implemented in the current version of the system. In order to allow more advanced names-menus which also support referencing into objects, additional predefined data types, like reference and class types, need to be implemented in OOSL.

APPLAB has been used in a number of smaller case studies. A more extensive case study on robot programming is presented in Chapter 6. The system was then integrated into the Robotics lab at the Department of Automatic Control, Lund University, and used as a programming interface to an ABB Irb-6 robot. An extended version of ABB's robot programming language RAPID [ABB94] has been implemented in the APPLAB system, and simple programs have been written and executed on the robot.

An interesting aspect of the grammar interpreting features of the APPLAB system, is that the meta grammars themselves (the grammar used to describe, e.g., the abstract and concrete syntax) can be edited, and designed to offer the same (text-editing and static-semantic) support to the grammar developer as to the ordinary programmer.

3.9 Summary

This chapter has given an overview of APPLAB, a highly interactive language laboratory, providing a language-based environment for integrated development of both a DSL and example programs in the experimental changing DSL. APPLAB is implemented using immediate computation as a main design principle, allowing changes to a language specification to be immediately tried out on example programs. We have argued that this is particularly useful in the development of DSLs where a lot of experimentation may be needed with the various aspects of the lan-

guage specification; abstract syntax, concrete syntax, and the specification of various static-semantic aspects.

We will now continue by describing the static semantics and the parser component of the APPLAB system in more detail. Then, a case study on the development and integrated support for industrial robots will be presented.

Chapter 4

A Grammar-Interpreting Static-Semantic Analyser

The static-semantic component of the APPLAB system is described in this chapter. After an introduction, the object-oriented attribute grammars used to define static-semantic rules are discussed. The demand-driven attribute evaluation of these grammars is described as well as a framework for predefined attribute types. The chapter is concluded with a report on the current status and future work.

4.1 Introduction

The static-semantic information of a program can be used for, e.g., name analysis, type checking, detection of static-semantic errors, and code generation. Advanced editing support can be provided by making the static-semantic information available to the program editor. *Semantic editing* allows the programmer to edit a program according to its static-semantic rules. For example, a static-semantically correct name can be selected from a *names-menu* containing all the declared and type-correct names at the current editing focus. The editing of function calls, with the correct number and type of arguments, can also be supported by semantic editing. Another kind of advanced editing support is *interactive static-semantic error reporting* which checks the edited program for errors, which are then reported. Advanced editing support promotes the writing of static-semantically correct programs at an early stage in the program development process.

Over the last decades the techniques used for producing tools used for program development have evolved. Four 'generations' of techniques can be distinguished: handcoding, generation from imperative language specifications, generation from declarative specifications, and interpretation of declarative specifications. The four generations, and examples of programming environments and conventional systems (e.g., compilers) for the different generations are shown in Table 4.1.

The first compilers and programming environments were handcoded for a specific language. The Cornell Program Synthesizer [TR81], for example, is a

programming environment handcoded for a dialect of PL/I (PL/CS), and also for Pascal. The system provides syntax-directed editing of PL/CS programs. The incremental static-semantic analyser contained in the system is used for highlighting occurrences of static-semantic errors in a program.

The next generation of software tools was based on generation. That is, a programming environment (or compiler component) is generated from a language specification. An example is the Gandalf system [Not85] which generates syntax-directed editors. Semantic rules can be specified in Gandalf using *action routines* which are activated before and after certain editing events like insertion and deletion. The parser generator Yacc [Joh79] also takes specifications containing action routines and produces a parser for the specified language.

The third generation of software tools are also generative, but are based on declarative specifications. The Synthesizer Generator [RT89] generates environments similar to the Cornell Program Synthesizers from language specifications based on attribute grammars. This is comparable to Eli [GHL+92, Kas96] which generates traditional compilers for which the static semantics of the language is described using attribute grammars. FNC-2 [JBP90, JP91] is system which generates incremental attribute evaluators from AGs. Integration of the FNC-2 system with Centaur [BCD+88] resulted in the programming environment generator Minotaur [AP94].

	Programming Environments	Conventional systems	Characteristics
1	Cornell Program Synthesizer	Handcoded compilers	Hand-coded for a specific language.
2	Gandalf	Yacc	Generation. Imperative specifications (e.g., action routines)
3	Synthesizer Generator Minotaur	Eli, FNC-2	Generation. Declarative specification (e.g., AGs).
4	Mjølner/Orm, ASF/SDF, APPLAB		Interpretive systems. Declarative specifications.

Table 4.1: The Evolution of Programming Tools

The fourth generation of software tools supply a language-specific programming environment by interpreting declarative language specifications. This approach results in systems suitable for rapid language prototyping, i.e. systems which give immediate feedback on changes made to the language specifications. These techniques can also be applied to the language specifications, providing the same editing support for specifications as for programs. Examples of such systems include the Mjølner/Orm Environment [KLLM93, MHM+90], the ASF/SDF Meta Environment [Kli91] and APPLAB. All three of these systems support an interactive language-sensitive editor by interpreting specifications of the syntax and the

static semantics of a language. The Mjølner/Orm environment also provides advanced editing support based on the static-semantic rules of a language. Interactive static-semantic error reporting is supported by interpreting a language specification, while semantic editing support is provided for the object-oriented language Simula with handcoded names-menus.

APPLAB is a programming environment of the fourth generation, i.e. the system interprets a declarative language specification. Semantic editing support is provided, both for programs and for grammars, by interpreting an object-oriented attribute grammar. This is more flexible than the handcoded names-menus of the Mjølner/Orm system. The supported subset of standard AGs contains enough functionality to allow the declaration of static-semantic rules for block-structured languages. The static-semantic analyser is implemented using demand-driven attribute evaluation, a simple but general evaluation mechanism which allows for an interactive notification of static-semantic errors.

After a description of the object-oriented attribute grammars used in APPLAB, the implementation of the demand-driven attribute evaluator and a number of predefined attribute types is presented. The use and implementation of the advanced editing facilities of APPLAB are discussed before the chapter is concluded by a report on the current status and future work of the static-semantic analyser.

4.2 Object-Oriented Attribute Grammars

An object-oriented version of AGs, OOSL (Object-Oriented Specification Language) [Hed92a], is used to express the static-semantic aspect of a language in the APPLAB system. Figure 4.1 contains part of an OOSL grammar expressing the static-semantic rules for type checking an assignment statement of a strongly typed language. Each nonterminal and production is represented by a *node class*, e.g., `Stmt`, `AssignStmt`, `Exp`, `IdUse` and `IfStmt`. Nonterminals are represented as *alternation node classes* (::!), e.g., `Stmt` and `Exp`. Productions are represented as *construction node classes* (::=), e.g., `AssignStmt`, `IfStmt` and `IdUse`. A construction node class may have a number of class parameters declaring the number and types of son node for the node class. For example, two sons are declared for `AssignStmt`, i.e. `IdUse` and `Exp`. A construction node class specializes a node class by using the inheritance mechanisms of the object-oriented paradigm, for example, `AssignStmt` is a specialization of `Stmt` since it inherits from `Stmt`. This kind of inheritance will be called *oo inheritance* in the rest of this chapter in order not to confuse it with the inherited attributes of AGs. The nodes of an abstract syntax tree consist of instances of the construction node classes and, for 'empty' nodes which mark unexpanded parts of a program, alternation node classes.

```
ANYNODE: node ::!
{  syn error: boolean;                                          (1)
   eq error := false;                                           (2)
   inh env: ref OOSLdictionary;                                 (3)
   eq son ANYNODE.env := env                                    (4)
};

Stmt: node ANYNODE::!;

AssignStmt: node  Stmt::=
   (a_IdUse: ref IdUse, a_Exp: ref Exp)
{   eq a_IdUse.expType := a_Exp.type;                           (5)
   eq a_Exp.expType := a_IdUse.type                             (6)
};

Exp: node ANYNODE::!
{  inh expType : ref OOSLtype; (* Expected type *)              (7)
   syn type : ref OOSLtype lazy;(* Actual type *)               (8)
   eq type := Unknown;                                          (9)
};

IdUse: node Exp::= (a_ID: ref ID)
{  loc declaredType: ref OOSLtype lazy;                        (10)
   eq declaredType := env.locate( a_ID.lex );                  (11)
   eq type := if declaredType = none                           (12)
              then Unknown
              else declaredType;
   eq error := declaredType = none or                          (13)
                 not (type=expType or type=Unknown or
                      expType=Unknown);
};

IfStmt: node  Stmt::=
   (a_Exp: ref Exp, Stmt1: ref Stmt, Stmt2: ref Stmt)
{   eq a_Exp.expType := BooleanType;                           (14)
   eq error := a_Exp.error or                                  (15)
               Stmt1.error or Stmt2.error;
};
```

Figure 4.1 Part of a static-semantic grammar expressed in OOSL.

Each node class may have any number of *attributes* and *equations* which define the values of attributes. For example, the node class ANYNODE contains the attributes error and env, and the equations (2) and (4). Both attributes and equations are oo inherited by any node class which specializes ANYNODE, e.g., Exp. The equations defining the attributes can be seen as virtual functions which can be overridden by equations in an oo inheriting node class. This is very useful for defining default equations for attributes which can then be redefined in a subclass. For example, the error attribute is defined by equation (2) in the ANYNODE node class, and is redefined in equation (13) of the node class IdUse (which oo-inherits from ANYNODE via Exp).

There are three different kinds of attributes: *local*(loc), *synthesized*(syn) and *inherited*(inh). Local attributes, like declaredType (10), are used and defined within the declaring node class. Synthesized attributes are local attributes which can be accessed by the father node. For example, the attribute type (8) of Exp is a synthesized attribute which is used in equation (5) of AssignStmt to access the actual type of the expression. Inherited attributes are defined by an equation in the

father node, and are often used to propagate declarative information through the syntax tree. A *collective equation* propagates a value to an inherited attribute of all sons of a given type. For example, in equation (4) the (inherited) attribute `env` is propagated to all nodes of type `ANYNODE`, including nodes which are a specialization of `ANYNODE`.

4.3 Demand-Driven Attribute Evaluation

The static-semantic analyser is implemented as a *demand-driven attribute evaluator*. The values of the static-semantic attributes are not stored in the abstract syntax tree, but are evaluated on demand. A demand-driven attribute can be seen as a *semantic function* of a node, (as is noted by e.g., Engelfriet [Eng84]). When invoking the semantic function of an attribute its value is calculated and returned. Using a purely demand-driven algorithm can lead to performing the same attribute evaluation several times. An alternative approach, which avoids such unnecessary computations, is *lazy demand-driven evaluation*, first described by Jalili [Jal83]. Each attribute evaluation is then only performed once; the first time the attribute is accessed. The resulting value is stored in the tree and is used for subsequent attribute accesses. This saves time but requires additional space. APPLAB supports a combination of the two approaches similar to the attribute evaluator of the Mjølner/Orm system [Hed89a] which supports both *lazy* (or stored) *attributes* and *demand attributes*. Optimal performance can be obtained by configuring the attribute evaluation wisely; lazy for attributes which are accessed several times in an evaluation and for which the evaluation is time consuming.

The static-semantic evaluator of the APPLAB system was implemented based on the techniques for demand-driven and lazy evaluation of object-oriented attribute grammars described by Hedin in [Hed92a]. An overview of the architecture is shown in Figure 4.2. The `User` (i.e. the programmer or the editor) asks the `Controller` to evaluate an attribute (`evaluate attribute`). The `Controller` instructs the `Compiler` to check that the `Evaluation Structure` is up to date (`check`). The `Evaluator` is then invoked to evaluate the desired attribute (`eval-`

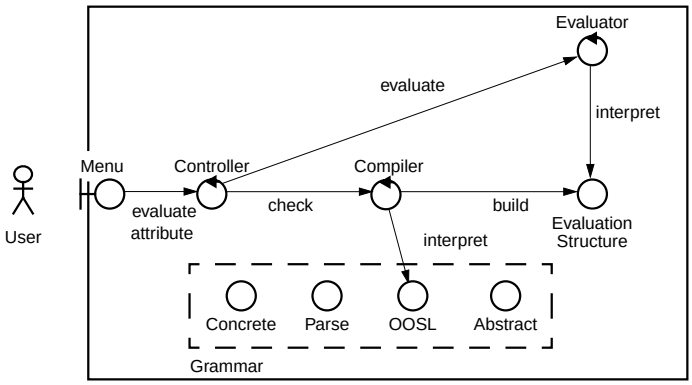

Figure 4.2 An overview of the architecture of the static-semantic analyser. (Notation explained on page 26.)

uate). The Evaluator calculates the current value of the required attribute by interpreting (interpret) the Evaluation Structure. The Compiler generates (build) this Evaluation Structure from the OOSL grammar (interpret) by binding the declared attributes to a representation of their definitions, thus allowing for a more efficient attribute evaluation.

4.3.1 Prototypes and Attribute Tables

A number of attributes and equations are defined for the different node classes. Each attribute is represented by a *semantic function* defined by an equation. The semantic functions are similar to virtual functions of object-oriented languages, that is they can be redefined in a subclass and the actual binding is done at runtime. The vtable technique, used for implementing virtual functions in languages like Simula [DMN68] and C++, can then also be used for demand attributes. The implementation in APPLAB is based on an adaptation of these techniques for object-oriented AGs presented by Hedin in [Hed89a].

In the Evaluation Structure, shown in context in Figure 4.2, and in more detail in Figure 4.3, the attributes are organized into *tables of attributes*. Each node class is represented by a Prototype containing a reference to its (possible) superclass, a Table of Attributes and a Son Table containing a Table of Inherited Attributes for each son of the node class. The position of the semantic function for an attribute is determined by the order in which the attribute is declared in the node class.

The Evaluation Structure is constructed by the OOSL Compiler by interpreting (interpret) the OOSL grammar and creating a Prototype for each node class. A position is reserved (Add) in the Table of Attributes for each attribute

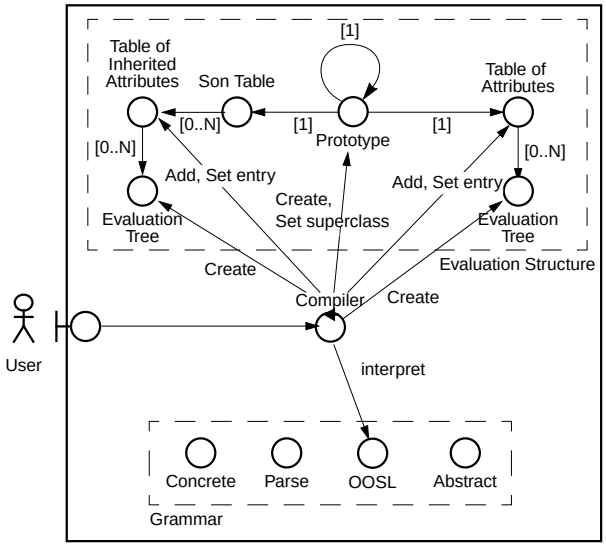

Figure 4.3 The OOSL compiler is responsible for keeping the Evaluation Structure up to date. ([n] indicates that an object refers to n other objects of the given kind.)

declaration of the current node class. Each attribute of a node class is given an offset into the table of attributes. This offset is then used when accessing the semantic function for the attribute. Upon compiling an equation the semantic function is translated (`create`) into an `Evaluation Tree`. The structure of this tree is equivalent to the structure of the expression defining the semantic function. The only difference being that any attribute accesses (by name) are replaced by the corresponding location of their semantic function. (See Section 4.3.2 for a more detailed description of the evaluation trees.) Once the `Evaluation Tree` is complete it is stored (`Set entry`) in the `Table of Attributes`, at the position of the defined attribute and is later evaluated by the static-semantic analyser when calculating the attribute's current value.

The evaluation structure for the OOSL grammar of Figure 4.1 is shown in Figure 4.4. The attributes `error` and `env` declared in `ANYNODE` are found in the table

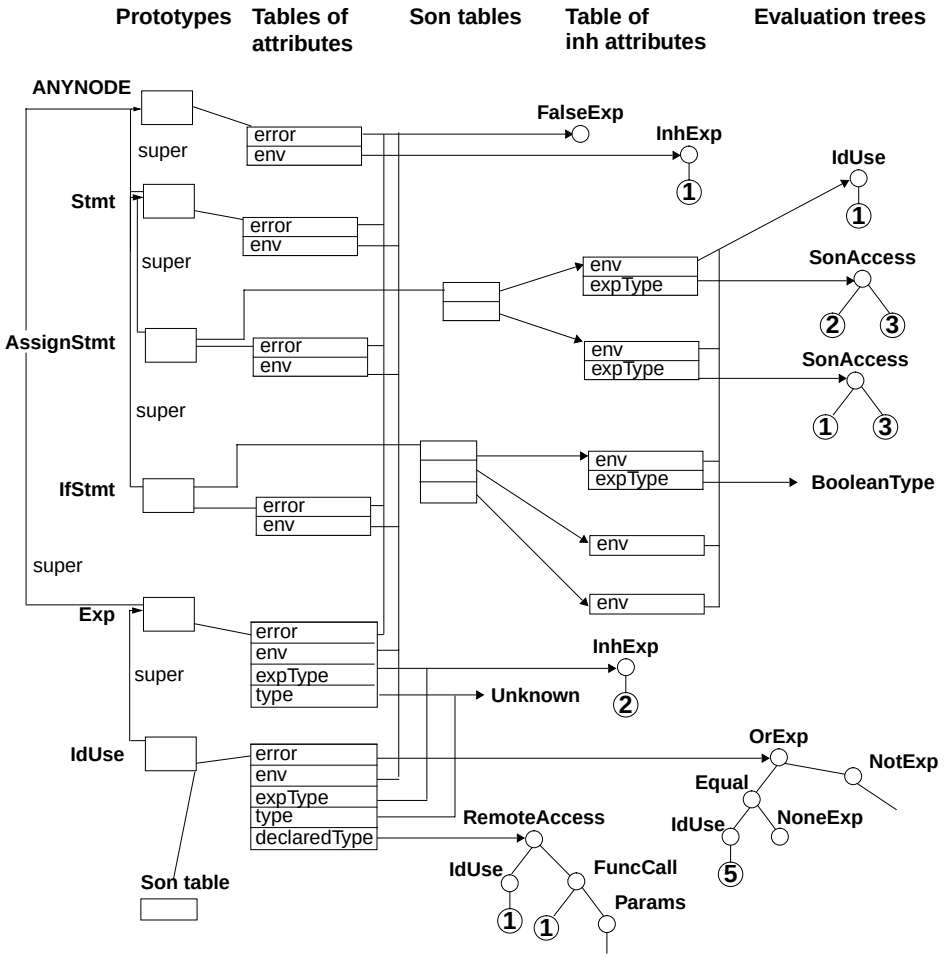

Figure 4.4 Part of the evaluation structure for the OOSL grammar of Figure 4.1.

of attributes of the prototype for ANYNODE. The semantic function for error, defined by equation (2), is represented by the evaluation tree FalseExp. Since Stmt is a specialization of ANYNODE both the attributes error and env are oo-inherited and, thus, also found in the table of attributes for Stmt. Note that the evaluation tree for both attributes is shared by ANYNODE and Stmt since no overriding equations are declared in Stmt. This is not the case for IdUse. Since equation (13) overrides the error attribute a different evaluation tree is stored for the error attribute in the table of attributes for IdUse.

Inherited Attributes

The analogy of virtual functions works well for local and synthesized demand attributes since their semantic functions are located in the same node class hierarchy as the declarations of the attributes, as is also the case for virtual functions. Inherited attributes do not directly follow this analogy since the semantic function of an inherited attribute is to be found, not in the declaring node class but in the node class of its father which is not known until 'run-time'. A different technique from the one presented in [Hed89a] is used in APPLAB. The semantic functions of inherited attributes are stored in a Son Table in the node classes of their prospective fathers. This table contains a Table of Inherited Attributes for each son. For list nodes which can consist of any number of sons, only the first entry of the Son Table is used since all sons are of the same type.

Looking at the example grammar on page 44, we see that equations (5) and (6) of AssignStmt and equation (14) of IfStmt all define different semantic functions for the inherited attribute expType (7) of Exp. The evaluation structure, shown in Figure 4.4, shows that these three equations result in three different semantic functions, each stored at a different location in the evaluation structure. The son table of IfStmt contains three entries, one for each son, i.e. a_Exp, Stmt1 and Stmt2. The table of inherited attributes at the first position in the son table contains the semantic functions for the inherited attributes of Exp, i.e. env and expType. The semantic function BooleanType has been stored for expType as a result of equation (14). Similarly equations (5) and (6) of AssignStmt define the semantic functions for expType, found in the tables of inherited attributes located at the first and second entry of the son table for AssignStmt.

When evaluating the *i*th inherited attribute of a node which is the *n*th son of its father the Evaluator locates the prototype of the father node, and accesses the *n*th entry of the Son Table which is the table of Inherited Attributes for the *n*th son. The *i*th entry of that table contains the semantic function for the requested inherited attribute. As an example, assume we wish to evaluate the attribute expType of an Exp node located in an IfStmt. The father node (an IfStmt) is then requested to evaluate the second inherited attribute of its first son. The correct semantic function (BooleanType) can then be located by accessing the first entry of the son table of the IfStmt, and the second entry of the accessed table of inherited attributes.

4.3.2 The Evaluation Trees

A semantic function for an attribute is represented by an evaluation tree. The structure of this tree corresponds to the semantic function. For example, the expression

```
12+3*7
```

would be represented as the tree

```
add(int_const(12), mult(int_const(3), int_const(7)))
```

where `add`, `mult` and `int_const` are nodes of the evaluation tree. All semantic functions are represented in this way, by trees consisting of evaluation nodes. All types of expressions allowed for defining the semantic function of an attribute in the OOSL language have a corresponding type of evaluation node. For example, attributes used in expressions are represented by `IdUse`-nodes containing the offset into the table of attributes which corresponds to the attribute. Thus, access to the attribute `declaredType` of the `IdUse` node in Figure 4.1, on page 44, is represented by

```
IdUse(5)
```

since `declaredType` is the fifth attribute declared for the `IdUse` node class and its semantic function is consequently found at the fifth position in the table of attributes. Similarly, the access of a synthesized attribute of a son node is represented by a `SonAccess` node containing the son's number and the offset of the required attribute.

As was discussed in Section 4.3.1, the semantic functions of an inherited attribute are located in the current father node. In order to access an inherited attribute it is represented by an evaluation node for inherited attributes (`InhExp`). For example, the inherited attribute `expType` of `Exp` in Figure 4.4 is represented by the evaluation tree `InhExp(2)`, where the number 2 indicates that it is the second inherited attribute of `Exp` (the first being the attribute `env`).

Implementation

The evaluation nodes are implemented as classes with a virtual function *eval* which takes an abstract syntax node and a representation of the evaluation (or, run-time) environment as its parameters. The syntax node is used for traversing the tree and for accessing the correct prototypes. The environment variable contains a stack used for, e.g., passing actual parameters and objects currently being accessed, for iterating over the nodes of a list, and for temporary variables. For each class the function `eval` has been implemented to evaluate the current object at the given node in the program tree. For example, the evaluation node `IdUse` is implemented in Simula by the class `IdUse` in the following way:

```
EvaluationNode class IdUse(offset); integer offset;
begin
   ref (Wrapper) procedure eval(targetNode, env);
      ref (ASTnode) targetNode;
      ref (RTEnvironment) env;
   begin
      ref (EvalNode) semanticFunction;

      semanticFunction :-
         targetNode.myPrototype.TableOfAttributes(offset);
      eval :- semanticFunction.eval(targetNode, env);
   end;
end;
```

The attribute's position in the table of attributes is denoted by the class parameter `offset`. When `eval` is invoked for an `IdUse`-object the semantic function for the requested attribute is located in the prototype of the current abstract syntax node, and evaluated by calling its virtual function `eval`.

Attribute values are handled in a uniform way by 'wrapping' them in Wrapper objects. There is one kind of Wrapper for each data type, i.e. IntWrapper, StringWrapper, BoolWrapper and so on. Virtual functions of the Wrapper class are used for performing basic arithmetic and logical operations on the wrapped values. For example, boolean values are represented by an object of the following class:

```
Wrapper class BoolWrapper(v); boolean v;
begin
   ref (Wrapper) procedure AndExp(wr); ref (BoolWrapper) wr;
      OOSLAnd :- new BoolWrapper(v and wr.v);
   .......
end;
```

The object-oriented implementation of the evaluation trees and attribute values has proved to be very useful. Extending the OOSL notation with new operations and additional simple data types results in extending the existing class hierarchies of the evaluation nodes and wrapper objects, and adding new virtual functions to the Wrapper class. Since the class hierarchy for the evaluation nodes correspond to the expressions specified in the grammar for the OOSL notation it is relatively easy to maintain the implementation and check that it corresponds to the current OOSL version.

4.3.3 Lazy Attributes

An attribute can be defined as 'lazy' in the OOSL grammar aspect. The first time such an attribute is evaluated the resulting value is stored in the program tree to be re-used at later accesses. This saves time for an attribute which is often accessed but takes up memory space, mostly for the attributes which are lazy but the implementation itself also requires some additional memory. Each entry in the table of attributes is extended to also contain a flag indicating whether or not the attribute is lazy. The first time a lazy attribute is evaluated the resulting value is stored in the program tree at the current abstract syntax node. When evaluating a lazy attribute the demand-driven attribute evaluator first tries to find the requested attribute in the current syntax node. If a value is stored for the attribute this value

is used, otherwise the semantic function of the attribute is evaluated and the resulting value stored.

4.4 Framework for Predefined Attribute Types

OOSL allows attribute values to contain simple data types like boolean, integer and string. Reference types and simple classes are also supported. In order to express static-semantic rules like declaration-usage dependencies, more efficiently, structured data types have been implemented in APPLAB. A few predefined attribute types like sets, dictionaries and types have been added to the OOSL grammar. They are implemented as *applicative classes*, that is, classes "whose objects are used to represent values" ([Hed92a, page 67]). The object must therefore never be changed and when testing for equality a special function of the class is to be used, not simple object identity where the reference pointers are compared.

The predefined attribute types each have a class interface description in OOSL, shown in Figure 4.5. The class `OOSLset` represents a set of strings. Simple data types, like integer and real, can be represented (for putting into symbol tables) by creating objects of the class `OOSLtype`. A symbol table containing identifiers and their declared types can then be created by using the class `OOSLdictionary` which contains the names of identifiers and a type-object representing their declared type. The function `match` of the `OOSLdictionary` class returns the set of all the identifiers of the current `OOSLdictionary` which correspond to a given type. This can be used for locating the set of all type-correct identifiers.

```
OOSLset: class
   (* applicative: A set of strings *)
{  empty: func boolean;
   contains: func boolean (item: string);
   add: func ref OOSLset (item: string);
   union: func ref OOSLset (s: ref OOSLset );
   equal: func boolean (s: ref OOSLset )
};

OOSLtype: class;
   (* applicative: A class used for representing types *)

OOSLdictionary: class
   (* applicative: Dictionary mapping string to OOSLtype. *)
{  locate: func ref OOSLtype(item: string);
   add: func ref OOSLdictionary
        (item: string; type: ref OOSLtype);
   union: func ref OOSLdictionary
        (d: ref OOSLdictionary );
   match: func ref OOSLset
      (* Returns all strings which match the given type t.*)
         (t: ref OOSLtype)
};
```

Figure 4.5 The declarations of the predefined attribute types in OOSL.

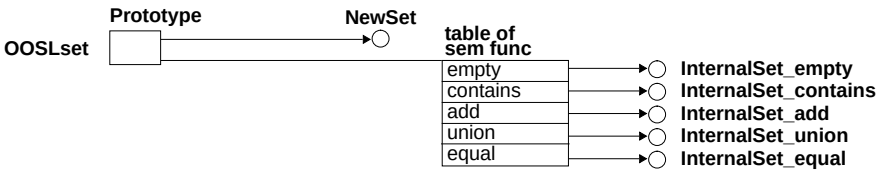

Figure 4.6 The prototype for the predefined attribute type OOSLset.

Implementation

The internal structures of the predefined attribute types are built explicitly (coded in Simula) and connected to their class interface descriptions in OOSL. These internal structures are organized into a prototype similar to the ones used for attributes and identical to the ones used for classes defined in OOSL. This means that the compiler can treat the usage of predefined attribute types the same as using any other class declared in OOSL. New attribute types are added by defining a new class interface in OOSL, implementing a number of (sub)classes using the base classes of the framework, and (in the code for the OOSL compiler) creating and connecting a prototype containing these classes to the OOSL interface description.

The prototype of the `OOSLset` class is shown in Figure 4.6. Each function of the `OOSLset` is represented by an entry in the table of semantic functions (similar to a table of attributes). The evaluation nodes stored at each entry are specializations of the general class `EvaluationNode`. The prototype also contains a reference to an evaluation node (`NewSet`) which is used for evaluating `new`-expression which create new `OOSLSet` objects. The virtual function `eval` of these evaluation nodes, i.e. `NewSet`, `InternalSet_empty`, `InternalSet_contains` etc., perform the required evaluations on internal classes implementing `OOSLset`:s. These OOSLset objects are contained in `SetWrapper`:s, a specialization of the `Wrapper` class. The code for the evaluation classes for `OOSLSet` is as follows:

```
EvaluationNode class NewSet;
begin
   ref (Wrapper) procedure Eval(targetNode, env);
      ref (ASTnode) targetNode; ref (RTEnvironment) env;
   begin
      Eval :- new SetWrapper;
   end;
end;

EvaluationNode class InternalSet_empty;
begin
   ref (Wrapper) procedure Eval(targetNode, env);
      ref (ASTnode) targetNode; ref (RTEnvironment) env;
   begin
      ref (SetWrapper) wr;

      wr :- env.getInternalData;! Fetches the current OOSLSet;
      Eval :- wr.setEmpty;
   end;
end;
EvaluationNode class InternalSet_contains; begin ... end; ....
```

Just as for operations and simple data types, the modelling of predefined attribute types, using classes and virtual functions, has resulted in a system which is relatively easy to maintain and extend with additional structured data types. Also, no additional considerations need to be taken when compiling and evaluating attributes of these types. This is due to the fact that the predefined attribute types are accessed via their interface description in OOSL and implemented using the same class hierarchy as simple OOSL data types and expressions.

4.5 Current Status and Future Work

The static-semantic analyser currently used in APPLAB supports demand-driven attribute evaluation of an object-oriented version of standard AGs, OOSL. The attribute evaluator was implemented using an existing technique based on the vtable technique used for object-oriented languages. This technique was adapted to the grammar-interpreting approach, and the representation of the semantic functions for inherited attributes was improved (by introducing son tables).

The static-semantic analyser has been used to implement advanced editing support, i.e., names-menus and interactive static-semantic error reporting, as well as code generation for the RAPID language (described in Chapter 6.) The interpretive nature of the static-semantic analyser allows the language designer to interactively specify the static-semantic rules determining (the code generation and) the behaviour of the names-menus and the static-semantic error reporting. This also means that advanced editing support can be provided for language specifications.

The currently supported version of OOSL has been extended with a number of predefined attribute types, e.g., list, set, dictionary, in order to support simple block-structured languages. A framework for implementing and adding these attribute types in a systematic way has been developed.

The names-menus of the current implementation are non-hierarchical. In the future we also wish to allow hierarchical names-menus which support access into reference types. The current interactive implementation of static-semantic error reporting is due to the inefficiency of the demand-driven attribute evaluator. Checking for static-semantic errors after each editing operation would be too slow using the current evaluator. Due to this and (mainly) to the fact that we wish to support the specification of object-oriented language, we intend to use Door AGs [Hed92a] in the future (also mentioned in Section 3.8). The object-oriented grammar notation OOSL (Object-Oriented Specification Language) [Hed92a], currently used can also be used to express Door AGs [Hed92a] which allow for efficient incremental attribute evaluation using a technique that scales up well. A problem with the current implementation of the static-semantic analyser is the amount of preprocessing required to construct the evaluation structure from the OOSL grammar aspect. This could also be alleviated by supporting incremental attribute evaluation for Door AGs. The construction of the evaluation structure could then be described in the meta grammar for OOSL. The evaluation structure for a language would then be incrementally updated as the language designer edits the OOSL grammar aspect.

Chapter 5

A Grammar-Interpreting Parser

This chapter describes the dynamic parser component of APPLAB. Parsing is done according to the current state of the grammar. To achieve this dynamic behaviour, the parser is based on grammar interpretation and uses an object-oriented recursive-descent parsing algorithm.

5.1 Introduction

This chapter describes the dynamic parser GRIP (GRammar-Interpreting Parser) implemented as part of APPLAB. The GRIP parser is used both to allows textual editing as an alternative to structure-oriented editing, and to supports document exchange with other environments via textual representation. Analogous to the structure editor, GRIP uses grammar interpretation to supply dynamic language-specific parsing. Other systems which contain dynamic parsers include DOSE [FJS86] and GIPE [Kli91]. (See section 5.4 for a comparison.)

The requirements on a language-laboratory parser differ from those in a traditional text-based environment:

- *Parsing speed*. The speed of the parser is not as important in a language laboratory as in text-based environments: During editing, very small text-fragments are parsed at a time (often only part of a line), and as long as this can be done in a fraction of a second (so it is unnoticeable to an interactive user), the speed is acceptable. For parsing larger documents produced by other environments, the parsing speed will show, but since this is a comparatively uncommon operation in the language laboratory, a moderately fast parser may be acceptable.
- *Parser generation speed*. The parser generation speed is very important. It should take at most a few seconds to produce a new parser when the grammar is changed, in order to allow editing/parsing according to the new grammar to proceed immediately after the change.
- *Parser grammar*. The context-free part of a language specification is, typically, centered around an abstract grammar, augmented by a pretty-printing

specification describing how to unparse an abstract syntax tree as text. Preferably, the parser should use these existing specifications as far as possible. In particular, we want to avoid having to specify a separate parsing grammar in order to satisfy a particular parsing algorithm.

For GRIP we have chosen an object-oriented recursive-descent parsing algorithm which interprets the grammar while parsing. The object-oriented algorithm supports parsing directly into the structure described by the abstract grammar. To resolve parsing ambiguities, we have extended the grammar specification with precedence- and associativity rules. To achieve reasonable speed during the grammar-interpreting parsing, a graph representation of the grammar is constructed. This graph construction corresponds to the parser generation, and takes only a couple of seconds.

The rest of this chapter is organized as follows. Section 5.2 describes editing in APPLAB from the user's point of view. Section 5.3 describes the implementation of APPLAB and the interpretive parser in particular. Section 5.4 discusses related work, and section 5.5 concludes the chapter.

5.2 Text- and Structure Editing in APPLAB

APPLAB supports simultaneous editing of a grammar and an example program following the specified grammar. (The grammar formalisms are specified in meta grammars.) The edited structures are represented internally as abstract syntax trees (ASTs) and are presented on the screen as text by unparsing the ASTs. Any subtree can be selected by the user and edited either structurally or as text. In structure editing, constructs are inserted into the AST by selecting them from a menu which contains all syntactically legal alternatives at the point of insertion. In text editing, the textual representation of the selected subtree is edited by the user, and then parsed into a new subtree which replaces the old selection.

5.2.1 Overview

Figure 5.1 contains a snapshot of an APPLAB session. The grammar currently edited describes RAPID [ABB94], a robot programming language. The windows marked ABSTRACT, CONCRETE, and PARSE contain different *aspects* of the context-free part of the language specification (the context-sensitive parts are not discussed in this chapter). The ABSTRACT aspect describes the abstract syntax, i.e. the syntactic structure of the language. The CONCRETE aspect describes the concrete syntax, i.e. the screen layout of different constructs of the language. The PARSE aspect is used to define additional rules needed for parsing. The PROGRAM window contains an example program following the RAPID grammar. Changing, e.g., the concrete syntax in the CONCRETE window causes the PROGRAM window to be updated to show the new syntax instead of the old one. In the same way, a new language construct, added to the ABSTRACT and CONCRETE windows, is immediately accessible in the PROGRAM, both for structure and text editing.

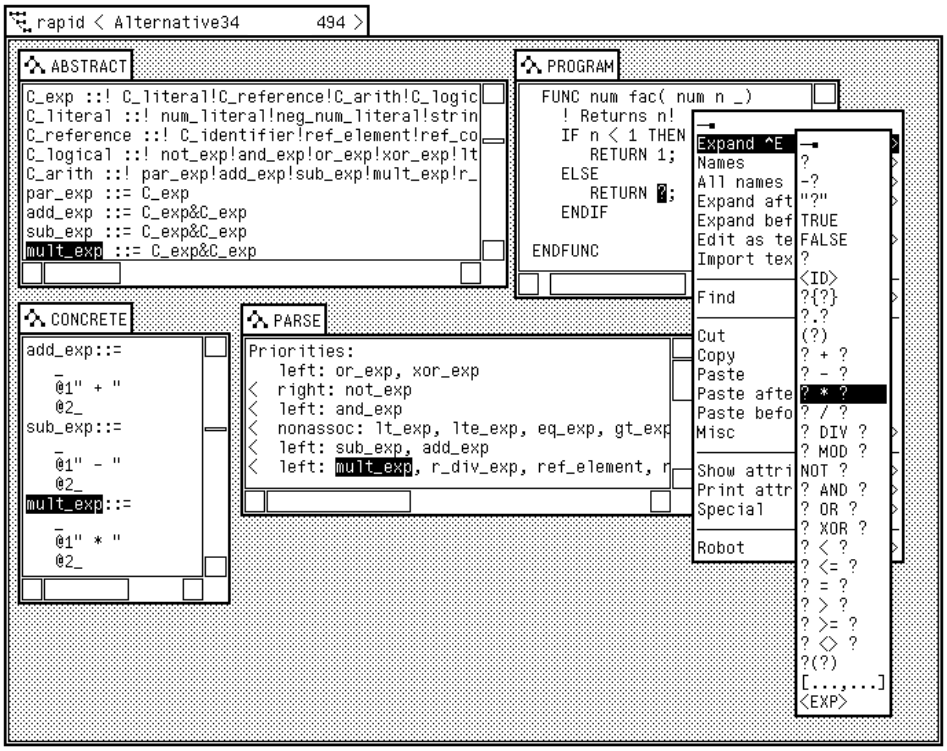

Figure 5.1 Structure Editing in APPLAB

Figure 5.1 shows the different grammar aspects for expressions, and for `mult_exp` in particular. The ABSTRACT aspect gives the BNF AND/OR structure of the grammar. An OR rule corresponds to the declaration of a class hierarchy, where the left-hand side, e.g., `C_exp`[1], is a superclass of the symbols on the right-hand side, e.g., `C_arith`. Similarly, `C_arith` is a superclass of `mult_exp`. An AND rule corresponds to composition where a syntax node object of the left-hand-side class has components according to the right-hand side. For example, a `mult_exp` node has two components (son-nodes) of class `C_exp`.

The CONCRETE aspect gives the concrete text representation for each of the AND rules. For example, when displaying a `mult_exp` node, the token "*" will be shown between the texts representing the two son nodes (referred to as @1 and @2).

The PARSE aspect shows the precedence and associativity rules for different expressions, in order to disambiguate the textual representations for different syntax trees. In the figure, the `mult_exp` has been given higher precedence than `add_exp` (rules on lower lines have precedence over rules on upper lines). This

1.The prefix "C_" used in some rules is used as a primitive structuring devise for the editor menus. It will be removed in future releases of APPLAB.

means that the string 'a+b*c' will be parsed as 'a+(b*c)'. Switching the precedence levels for `mult_exp` and `add_exp` would mean parsing the same string as '(a+b)*c'.

Text Editing

When choosing to edit a structure as text the user is given the text representation of the currently selected subtree as the initial text and may edit it character by character. In some situations text editing may be preferred to structure editing, e.g., when editing complex expressions, and when changing an existing construct. For example, changing an existing `while-stmt` to a `for-stmt` with the same body. Text editing such constructs is easy, while doing the same editing using the structure editing technique would mean a series of cut-and-paste operations.

If errors are encountered during parsing they are reported to the user in an error window which displays an error message and the position of the error in the edited text. GRIP tries to repair the errors as far as possible. For example, if an error causes the parser to fail in parsing the body of a `while-stmt`, the parse still results in a `while-stmt`, but possibly with an empty body structure.

Changing the Grammar

When the grammar is changed, subsequent text editing in the PROGRAM will be parsed according to the new grammar state. The user does not need to give any explicit command for updating the parser. Instead, the parser will be regenerated automatically the next time the user edits a subtree as text.

Some changes to the grammar may cause the existing program in PROGRAM to become inconsistent with the current grammar. If only the CONCRETE aspect is changed, the PROGRAM will be automatically reunparsed to reflect the new grammar state. If rules in the ABSTRACT aspect are changed, e.g., adding or deleting components in an AND rule, this will cause existing nodes of these rules in the PROGRAM to become inconsistent with the grammar. Currently, the inconsistent nodes are simply marked in the PROGRAM, and the user may explicitly delete them to obtain a consistent program. Future versions of APPLAB may add more sophisticated techniques for updating inconsistent PROGRAM programs, allowing transformation rules to be stated using, e.g., the techniques suggested in [GKL94].

Grammar Errors

When editing the grammar, errors will often be introduced. For example, the misspelling of nonterminals, the use of a nonterminal on the right-hand side of a rule without it appearing on the left-hand side of any rule, and inconsistent aspects of the same grammar rule, e.g., different number of components in the ABSTRACT and CONCRETE aspects of a rule. These errors are reported during the parser generation phase.

More intricate errors can occur as a result of certain combinations of rules, e.g., indirect left-recursion and grammar ambiguities. Indirect left-recursion is detect-

ed and reported during parsing, see section 5.3.5. Grammar ambiguities are not detected, but rules appearing early in the grammar are given precedence over rules appearing later, thus disambiguating the grammar.

Since we can assume that errors in the grammar will be rather common while designing a language it is desirable to have an environment that is as tolerable as possible towards these errors. The environment must not crash as a result of a grammar error, and it should be as supportive and helpful as possible in debugging the language being designed. GRIP has been designed with this in mind and allows parsing even with an incomplete, incorrect grammar. If the erroneous rules are used during parsing, the grammar errors are reported as warnings, and parsing is done according to a repaired version of the grammar, as described in section 5.3.5.

5.3 Implementation

5.3.1 OO Recursive-Descent Parsing

Recalling the requirements on language-laboratory parsers listed in section 5.1, it is important that the parser generation speed is high, whereas the actual parsing speed is less crucial, and that the abstract grammar used for structure editing can be used as the basis also for parsing. To meet these demands, we have chosen to base GRIP on OORD, an Object-Oriented Recursive-Descent parsing method [Hed89b]. Similar to traditional recursive-descent parsing, OORD follows the grammar closely, and it is therefore simple and fast to generate the parser. The parsing method is termed *object-oriented* because the language specification is given in an object-oriented form, using classes and subclasses, rather than in the traditional form using nonterminals and productions. These classes are called *node classes*, because they classify the nodes occurring in the syntax trees. An important advantage of the OORD method is that it is particularly well suited for parsing directly into abstract syntax trees:

- Precedence and associativity, normally specified using production rules such as term, factor, etc., and common prefix factorization, can be specified using specialization (subclassing), thereby avoiding the generation of non-abstract nodes in the resulting parse tree.
- Left-recursion, as it normally appears for expressions, can be handled directly by the algorithm. In contrast, in traditional recursive-descent parsing one has to eliminate left-recursion from the grammar, thereby distorting the resulting parse tree.

Precedence and Associativity

The example below shows how precedence and associativity of arithmetic expressions can be defined in the OORD notation:

```
Exp: ();
Term: Exp ();           # Term is a specialized expr
Factor: Term ();        # Factor is a specialized term
Primary: Factor ();     # Primary is a specialized factor
Add: Term (Term '+' Factor);
                        #  Add is a specialized term with two
                        #  subcomponents - a term and a factor.
Sub: Term (Term '-' Factor);
Mul: Factor (Factor '*' Primary);
Div: Factor (Factor '/' Primary);
Id: Primary (LexId);
ParExp: Primary ( '(' Exp ')' );
```

As seen from the example, specialization is used to limit what kinds of expressions may occur as subcomponents to different expressions. For example, the specification for Add limits the left component to be a term and the right component to be a factor. Add is thus left-associative, since another instance of Add can only appear as the left subcomponent. The corresponding specification in a conventional nonterminal/production notation would be:

```
add: <Term> ::= <Term> '+' <Factor>
```

where add is the production name, and Term and Factor are nonterminals. An important difference between these notations is that they result in different parse trees: the OORD parse tree is like an abstract syntax tree in that it only contains instances of the most specialized classes: Add, Sub, Mul, Div, Id, and ParExp. In contrast, a parse tree from a conventional grammar contains additional internal nodes for the nonterminals Exp, Term, Factor, and Primary.

Left-Recursion

As seen from the conventional specification of add, the production is left-recursive (the first component 'Term' is the same nonterminal as the left-hand side 'Term'). In a standard recursive-descent parser, such left-recursion must be eliminated by rewriting the grammar rules in a way which distorts the parse tree and adds more nodes to it. In the OORD method, such rewriting is unnecessary; the parser can deal with the left-recursion and construct the abstract syntax tree directly.

Common Prefix Factorization

If two productions with the same left-hand side nonterminal have a common prefix, a standard recursive-descent parser would, in general, need an unbounded lookahead. The usual technique to avoid this, in order to use a lookahead of fix length, is to factor out the common prefix into a new nonterminal, thereby changing the structure of the parse tree and adding more nodes to it. Using the OORD notation,

the factorization can be done by specialization rather than by adding new nonterminals, and this way the resulting parse tree is unaffected by the factorization. The example below shows an example grammar where two variants of the if-statement start in the same way, i.e. they have a common prefix.

```
Stmt: ();
IfThenElseStmt: Stmt ('if' Exp 'then' Stmt 'else' Stmt);
IfThenStmt: Stmt ('if' Exp 'then' Stmt);
```

The common prefix can be factored out by adding a new class `IfThenStarting-Stmt` which contains the common prefix. The new class is inserted in the class hierarchy between the general class `Stmt` and the specialized `If`-classes:

```
Stmt: ();
IfThenStartingStmt: Stmt ('if' Exp 'then' Stmt);
IfThenElseStmt: IfThenStartingStmt ('else' Stmt);
IfThenStmt: IfThenStartingStmt ();
```

Since we have only changed the class hierarchy, and not the structure of instances of the concrete most specialized classes, this factorization does not affect the structure of the parse tree.

In the case of left-recursive classes, like Add and Sub above, the common prefix does not have to be factored out because the OORD method for handling the left-recursion will automatically take care of the common prefix as well.

5.3.2 Parser Generation

When the grammar has been changed, the parser needs to be re-generated. For GRIP, being an interpretive parser, this re-generation means updating only the graph structure interpreted by the parser. The core of the parser is generic and does not need to be updated. The following steps are carried out:

- Check the grammar to report possible errors in it.
- Construct a graph representation of the grammar according to the information in the ABSTRACT and CONCRETE grammar aspects.
- Transform the graph representation of the grammar to deal with precedence and associativity according to the information in the PARSE aspect.
- Transform the graph representation by doing common-prefix factorization.
- Generate a lexer.

The graph construction and transformation steps are done even if the grammar contains errors. It is then possible to parse successfully, at least as long as none of the erroneous parts of the grammar are involved. The transformation steps are described in sections 5.3.3 and 5.3.4. Section 5.3.5 discusses error handling and in section 5.3.6 the generation of the lexer is described.

The graph is an internal representation of the grammar, structured according to the OORD notation. There is one vertex per node class, and related vertices are connected to each other in order to allow efficient interpretation: each vertex has a

set of edges to its immediate subclasses, and to its component classes. Node classes which have components also have an edge to the corresponding rule in the CONCRETE grammar aspect, where the keywords occurring in the rule can be found. The grammar transformations which are described in sections 5.3.3 and 5.3.4, are done only on the graph representation of the grammar, and do thus not interfere with the structure-oriented editor which interprets the original grammar.

5.3.3 Transformation of Precedence and Associativity

The precedence and associativity specification given in the PARSE grammar aspect is used to transform the grammar graph to include this information by inserting additional node classes. As an example, consider arithmetic expressions where the specification in the ABSTRACT and CONCRETE grammar aspects correspond to the following (ambiguous) OORD grammar:

```
Exp: ();
AddExp: Exp (Exp '+' Exp);
SubExp: Exp (Exp '-' Exp);
MulExp: Exp (Exp '*' Exp);
DivExp: Exp (Exp '/' Exp);
ParExp: Exp ( '(' Exp ')' )
```

These rules are disambiguated by the following PARSE aspect specification:

```
< left: AddExp, SubExp      precedence level 1
< left: MulExp, DivExp      precedence level 2
```

The grammar graph is transformed to include these precedence and associativity rules by inserting node classes, yielding the following OORD grammar:

```
Exp: ();
Exp_1: Exp ();
Exp_2: Exp_1 ();
Exp_3: Exp_2 ();
AddExp: Exp_1 (Exp_1 '+' Exp_2);
SubExp: Exp_1 (Exp_1 '-' Exp_2);
MulExp: Exp_2(Exp_2, '*', Exp_3);
DivExp: Exp_2(Exp_2, '-', Exp_3);
ParExp: Exp_3 ( '(' Exp ')' );
```

We consider the transformation corresponding to a precedence level specification for subclasses to a base class *B*. Each of the listed subclasses should be a unary or binary operator with subcomponents qualified by *B*.

Precedence is handled by inserting n+1 new node classes P_k, k=1..n+1, where n is the number of precedence levels, and where super(P_1) = B and super(P_k) = P_{k-1} (for$k$>1). Node classes at precedence level k are moved in the class hierarchy by making them subclasses to P_k, and the qualifications of their components are also changed to P_k, or to the more specialized class P_{k+1}, depending on associativity, as described below. For instance, `MulExp` on level 2 is made subclass to `Exp_2`, and its component qualifications are changed to `Exp_2` and `Exp_3` respectively.

Associativity of binary operators is handled by specializing either or both of the subcomponents to P_{k+1}. For left-associative operators, the second component is

specialized. This restricts an additional instance of an operator on the same level to appear only to the left of the operator. For instance, in `MulExp`, the second component is specialized to `Exp_3`. An additional instance of `MulExp` or `DivExp` can thus only appear as the first component. Analogously for right-associative operators, the first component is specialized. For non-associative operators, both components are specialized (disallowing adjacent instances of the operator altogether). Unary prefix (or suffix) operators are treated like binary operators with an empty first (or second) subcomponent.

Subclasses to the base class *B* whose right-hand side neither starts nor ends with *B*, for example `ParExp`, should not be listed in the precedence level specification since they cannot take part in the grammar ambiguities addressed by precedence and associativity. These classes are made subclasses to P_{n+1} and the qualifications of their components are left unchanged. For instance, `ParExp` is made subclass to `Exp_3`, but its component is still qualified by `Exp`.

The transformation algorithm for precedence and associativity is given below. In the algorithm we assume that all classes listed in the precedence specification are direct subclasses of the base class. Some of our grammars, however, make use of intermediate classes, such as `C_arith`, as shown in figure 5.1. These intermediate classes merely have the role of grouping similar expressions in order to give more structure to the abstract grammar, and they are removed in the parser version of the grammar before doing the transformation.

```
Algorithm PrecAndAssocRewrite(B)
   where B is a base class with subclasses specified in n
   precedence levels.

   Let SUB_k be the subclasses to B listed at level k.
   Let NONLISTED be the subclasses to B not listed on any
   level.

   Add a new class P_1; super(P_1) := B;
   for k := 1 to n do
      Add a new class P_k+1; super(P_k+1) := P_k;
      for each class X in SUB_k do
         super(X) := P_k;
         Let s_1 and s_2 be the subcomponents of X
         case associativity of level k:
            left: qual(s_1) := P_k; qual(s_2) := P_k+1;
            right: qual(s_1) := P_k+1; qual(s_2) := P_k;
            non: qual(s_1) := P_k+1; qual(s_2) := P_k+1;
         esac;
      od;
   od;

   for each class X in NONLISTED do
      super(X) := P_n+1;
   od;
```

5.3.4 Transformation of Common Prefixes

Common prefixes of sibling classes (classes with the same superclass) are factored out by introducing an additional class containing the common prefix, as was described in section 5.3.1. The algorithm is given in detail below. In the case of left-

recursive classes, however, the OORD method handles the common prefix automatically, and no factorization is needed.

```
Algorithm CommonPrefixFactorize(C, SUB)
   where C is a class and SUB is a set of subclasses of C.

   if |SUB| > 1 then
      let p be the longest common prefix in SUB.
      I.e., p is the longest common prefix of the sequences
         RHS(X), X ∈SUB,
      where RHS(X) is the sequence of right-hand side
      symbols (keywords and nonterminals) of the class X.
      if |p| = 0 then
         Split SUB into subsets SUB_k starting with the same
         symbol. I.e.,
            ∀X ∈ SUB_k, head(RHS(X))=sym_k.
               sym_i ≠ sym_j, i ≠ j
         for each set SUB_k do
            CommonPrefixFactorize(C, SUB_k);
         od
      else
         Construct a new class N for the common prefix p
         and insert it between C and SUB in the class
         hierarchy, removing the prefix p from the classes
         in SUB.
      fi
   fi
```

5.3.5 Error Handling

When doing interpretive parsing there are two sources of errors: errors in the parsed text and errors in the grammar. Errors in the grammar are reported at parser generation time, but since APPLAB allows parsing even if there are errors in the grammar, these errors may be reported at parsing as well.

Errors in the Parsed Text

GRIP performs *Phrase-level Error Recovery* [ASU86], if the parsed text does not match the grammar. I.e., it performs local correction on the remaining input in the following manner. At a token mismatch, the parser first tries to skip the current token and continue parsing. If there still is a token mismatch, the parser inserts the expected token instead and continues parsing, possibly encountering more errors.

If there is more than one possible expected token, i.e., if the parser is parsing a node class with subclasses, and the current token matches neither of the FIRST symbols for the subclasses, the parser inserts a placeholder for the expected node class and continues parsing.

Errors in the Grammar

Most grammar errors, e.g., reference to undefined symbols, etc., are reported at parser generation time. If the grammar contains errors, the incomplete or incon-

sistent rules are repaired in the graph representation and parsing is allowed all the same. If a repaired rule is used during parsing, the parser reports a warning. This is useful in the experimental language laboratory setting where the grammar is often changed; one can continue to edit the program even if the grammar temporarily contains errors.

As described above, the OORD algorithm handles direct left-recursion which appears, e.g., for normal left-associative binary expressions. However, the algorithm does not handle indirect left-recursion. According to [DM82, p. 62], indirect left-recursion hardly ever appears in practical grammars. Nevertheless, endless recursion must be avoided, should the user happen to specify a grammar with indirect left-recursion, by mistake or on purpose. To handle this, a check is done while parsing. When this check detects an indirect left-recursion in the current parse, an error is reported, and the current parse is cancelled.

It would be possible to check indirect left-recursion at parser generation time as well, but by checking it at parsing time it is possible to parse strings which do not involve the indirect left-recursion.

5.3.6 The Lexer

The lexer uses a table of keywords constructed from the CONCRETE grammar aspect. The table is constructed during parser generation. Currently, the definitions of other lexical items are hand-coded to fit the RAPID language, including identifiers, numbers, strings, and comments (for the latter two, start- and end-tokens can be specified in the PARSE aspect). These definitions are fairly standard and are sufficient for many languages. However, future releases of APPLAB will include a fully grammar-interpretive lexer which allows complete specification of the lexical level. Analogously to the parser, a graph representation of the lexical specification will be constructed and interpreted during lexical analysis. This will allow full use of the interpreting parser for all languages defined in APPLAB.

5.4 Related Work

Editing programs in a structure-oriented editor is fundamentally different from using a traditional text editor since the editing is done in terms of language constructs rather than characters. Many voices have been raised to say that text editing should not be banned altogether from structure-oriented editors, e.g., [VG92, Wat82]. On the other hand, it has been argued that the structure-oriented approach has not yet been fully explored and evaluated, and that the usual arguments given in favour of text editing are not valid in structure-based environments [Min92]. Whatever viewpoint is taken towards text editing in structure-oriented editors, there is a need for parsing in such systems if one wishes to integrate it and exchange documents with other (text-based) systems.

Syntax-Directed Versus Syntax-Recognizing Editors

Among the existing language-based environments there are two fundamentally different approaches to combining text and structure-oriented editing: *syntax-directed* editors and *syntax-recognizing* editors. In the syntax-directed editors, the syntax tree of the document is presented to the user, and editing operations are performed directly on the tree. Adding text editing to such editors is done as in APPLAB, by letting the user select a subtree and edit it as text. The edited text cannot affect any parts of the syntax tree outside the structural selection. Other environments supporting this kind of editing include the Synthesizer Generator [RT89], Centaur [BCD+88], Gandalf [Not85], GIPE (ASF/SDF) [Kli91] and DOSE [FJKS85].

The selection can be parsed either *exhaustively*, i.e. parsing all the text of the selection, or *incrementally*, i.e. parsing only the changed parts of the text and reusing the old syntax subtree in order to construct the new subtree for the selection. Incremental parsing speeds up the parsing for large selections. It depends on how the environment is used if large selections are edited as text or not. It is common to edit only small structures, e.g., expressions, as text, whereas larger structures, e.g., procedures or control structures, are edited using structure editing. In this case an exhaustive parser is usually sufficiently fast. APPLAB, the Synthesizer Generator, and Centaur use exhaustive parsing. The Gandalf system uses a technique for incremental parsing where a series of tree transformations are performed as tokens are inserted and deleted [KK85]. DOSE uses a combination of techniques, where assignments and expressions are parsed using a method based on the incremental parsing algorithm used in the Gandalf system.

A different approach in combining text and structure is taken in *syntax-recognizing editors*, e.g., Pan [BGV90]. In these editors, the user edits the document as text and the system maintains an internal syntax tree which is hidden from the user. This type of system allows the user to continue working in the same way as in a traditional text editor, while the internal syntax tree can be taken advantage of in the same way as in the syntax-directed editors, e.g., for cross-referencing or type-checking. The text-editing in the syntax-recognizing editors thus corresponds to text editing in a syntax-directed editor where the selection is always the complete document. The 'selection' is thus usually quite large, and incremental parsing is therefore very useful in these systems. Since it is not feasible to perform incremental lexical and syntactic analysis for each character-edit operation, a design decision has to be made for such editors as to when the incremental lexical analysis and parsing is to be invoked. In Pan, the different levels of analysis maintain a summary of increments to be used by the next analysis level. I.e. the incremental LALR(1) parser revises the internal structure in response to increments produced by the lexical analyser.

Language Laboratory Parsers

When implementing a programming environment or structure editor for a specific language, the easiest way to produce a parser component is to generate one by using a general-purpose *parser generator* such as yacc[Joh79]. For example, both

Centaur and the Synthesizer Generator use yacc to generate the parser component. However, yacc and other parser generators produce program code as output (C in the case of yacc) which has to be compiled and linked. For a language laboratory, this approach is not suitable since it would take too long time to generate a new parser, definitely more than a few seconds as was required according to our list in section 5.1. Furthermore, with current standard compilation and linking technology, such a parser could not be an exchangeable part of the language laboratory process, but would need to run as a separate process, adding interprocess communication overhead during parsing.

For a language laboratory, the reasonable approach seems to be to use some kind of generic parser interpreting a data structure derived from the grammar. The language-laboratory environments DOSE, GIPE, and our APPLAB, all use some variation of this approach. There are a few different techniques, or combinations of techniques, that are used to generate the parser sufficiently fast:

- Incremental parser generation. I.e., update the parser data structure according to the grammar changes.
- Choose a top-down parsing algorithm, since they require less pre-processing than bottom-up algorithms.
- Lazy parser generation. I.e., generate the parser data structure piece by piece as it is needed during parsing.

For example, APPLAB uses a top-down parser, while GIPE uses a bottom-up parser and both incremental and lazy parser generation. DOSE uses an algorithm that combines top-down and bottom-up parsing. The different language-laboratory parsers are discussed in more detail below.

DOSE Parsing is performed in DOSE by a combination of parsing techniques. Expressions and assignment statements are parsed using a method based on the incremental expression parsing algorithm developed by Kaiser and Kant [KK85], while other language constructs are parsed top-down by recognizing the first keyword of a construct, like IF and WHILE. Just as in APPLAB the same language description that is used for the structure-editor is also used by the parser. While APPLAB presents the subtree to be text edited in a separate window, DOSE allows the user to edit text within its context on the screen.

GIPE The parser generator of the GIPE environment, IPG [HKR90, Rek92], is implemented as a lazy, incremental LR(0) parse table generator. The parse tables are incrementally updated by need while parsing input. The syntax specification of GIPE is similar to the one in APPLAB in that the abstract syntax is augmented with disambiguating precedence and associativity rules. The text editing facilities of GIPE are more smoothly integrated into the structure-oriented editor. While APPLAB lets the user text edit a selected part of a program in a separate window, GIPE allows text editing of the program fragment currently in focus, to be done within its context in the structure-oriented editor. Parsing of the edited text is performed when the user moves the focus to another part of the program. If any syntax errors are encountered the user is required to correct them before moving

the focus. In APPLAB, should any syntax errors be detected, they are reported and, as far as possible, corrected by the parser and the resulting AST is inserted into the edited program. If the user so wishes the latest text editing operation can be undone by re-inserting the old (unedited) AST. GIPE and APPLAB both allow the user to select any part of the edited program for text editing.

TaLE The TaLE environment [JKN95, KM95] is a specialized, partly graphical, editor for designing textual languages. It allows the user to edit, reuse and refine language features in a controlled fashion. TaLE is suited for rapid prototyping of application-specific languages but is not a language laboratory in our sense since it does not allow the user to freely switch between designing and testing the language. Instead, TaLE generates Eiffel-classes (which may be manually edited) corresponding to the language constructs which are then compiled and result in a traditional source-code compiler rather than a structure-based editor.

TaLE uses incremental parser generation where the generated Eiffel classes are incrementally updated as the user changes the language description. The parsing algorithm used in TaLE is a lazy recursive descent algorithm modified to fit object-oriented context-free grammars [KV92]. The parsers in TaLE and APPLAB have certain similarities. They are both recursive-descent parsers, and they both draw on the principles of object orientation in distributing the information needed for parsing into objects representing the nonterminals of the grammar. In TaLE, this is utilized to support incremental (and lazy) parser generation. For example, changing the production for a nonterminal only affects the corresponding object in the parser.

5.5 Conclusion

We have presented the dynamic parser GRIP and its integration in the language design laboratory APPLAB. GRIP is a generic parser which interprets a grammar graph in order to parse according to a given grammar. Any text selected in APPLAB's structure-oriented editor can be edited as text and parsed by GRIP. In case the grammar has been changed since the last text edit operation, a new grammar graph is generated automatically, causing GRIP to always use the current version of the grammar. The system has been tested on several languages, ranging from small meta-grammars of around 10 productions to grammars for real-sized programming languages with over 200 productions (the grammar for RAPID, which is a Pascal-like language with some special constructs for programming robots).

APPLAB and GRIP are implemented in the object-oriented language Simula [Sta87] and runs on SUN Sparc workstations. An evaluation of GRIP compared to the requirements listed in section 5.1 yields the following conclusions:

- *Parsing speed*. GRIP parses around 190 tokens/second on a SUN Sparc Ultra 1. This corresponds to around 1700 lines per minute. This is sufficient for the main requirement, namely that parsing of small texts, such as short expressions, should be unnoticeable by the user. However, faster parsing would certainly be desirable for document exchange. There are many details in the

implementation which could be optimized and we therefore have good hope of increasing the parsing speed substantially.

- *Parser generation speed*. A new parser is generated in 0.6 seconds for a real-sized language with over 200 productions. We find this well within the acceptable delay.
- *Parser grammar*. As required, GRIP runs without the need of a special parsing grammar. The abstract and concrete grammars augmented with disambiguating precedence and associativity rules is sufficient information to generate the parser.

The object-oriented recursive-descent (OORD) algorithm turned out to be very suitable for implementation of the language-laboratory parser. Being a recursive-descent algorithm it is straight-forward to implement. Furthermore, the possibility to express precedence, associativity, and common prefix factorization using specialization, allowed us to transform the original abstract grammar to a grammar fit for the OORD algorithm, yet without changing the structure of the resulting parse tree. That is, the tree resulting from parsing is also a derivation tree of the abstract grammar.

Chapter 6

Case Study - Robot Programming

APPLAB has been used in several case studies where robot programming is the topic of the most extensive one. It was chosen because of its challenging demands on programming on several different levels which would benefit by applying the DSL technique. The robot programming language RAPID was implemented in APPLAB which was then used as a front-end for operating an ABB Irb-6 industrial robot. The case study was carried out in cooperation with the Department of Automatic Control at Lund University, using information provided by ABB Robotics Products AB. After an introductory motivation and presentation of the area of robot programming, the experimental setup and APPLAB's role in it is described, followed by a discussion on DSL support at the end-user level. The chapter is concluded by experience and future work.

6.1 Motivation

The purpose of the case study was to explore the use of interactive language-based tools for DSL development and programming in the context of industrially relevant and demanding programming situations, and in the course of doing so, to evaluate APPLAB and to identify desired extensions to further support DSL development. In an industrial context, we may focus on programming and configuration of computers as such, but other types of programmable equipment are probably more interesting; such programming often has to be carried out by engineers that are not primarily programmers which implies a greater need for domain-specific support. Furthermore, the possible benefits in terms of time and money are substantial if engineering time can be saved and the utilization of the (expensive) equipment can be improved. Examples include telecom systems, process control systems, and manufacturing systems. Perhaps the most demanding, or diversified, type of programming is the programming of industrial robots. Including the system programming usually done by the robot manufacturer, some characteristics are:

- Programming ranges from hardware related programming of low-level control and sensing, to very high-level programming using abstract operations that are created on lower levels of the system.

- Simple robot tasks should be simple to teach (the robot) for the inexperienced user, and advanced operations should be possible to implement by the experienced application expert.
- Programming constructs that are suitable for high-level operations are not appropriate for low-level control due to the demands on execution efficiency.
- The robot task is not known at the time of the design of the robot control system, and task descriptions may change at run time. Even the type of application may be new to an available robot programming system.
- Concurrency and timing has to be dealt with.

Thus, programming of industrial robots includes a relevant set of programming issues. Each of these issues can also be found in other applications, but a robot programming setup provides a comprehensive research environment for the study of programming tools and languages.

6.2 Robot Programming

When programming industrial robots there are several different levels of programming involved [Nil96a]: the *motion control level*, the *application-specific level* and the *end-user level*. Figure 6.1 illustrates these different levels. At the motion control level, the *developer* of the industrial robot works at the robot manufacturing company which delivers the product together with a programming interface for the basic motion control of the robot. The *application expert* then uses this interface to implement an application-specific level which specializes the robot for a specific application, e.g., welding or gluing. At the moment this phase is most often done by the manufacturers themselves, since the design of the application packages requires detailed knowledge of the system design including the motion-control interface. The resulting specialized robot is delivered to a customer, or *end-user*, who wishes to use the robot to manufacture some product. Programming is then done at the end-user level, typically by using a dedicated robot programming language together with data from teach-in or CAD/CAM systems [Cra89].

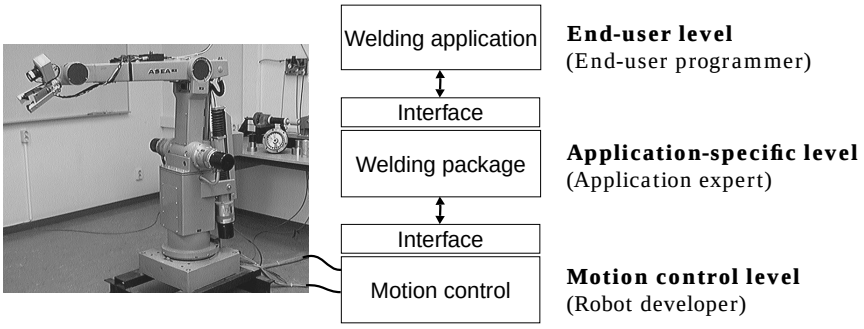

Figure 6.1 Different levels of robot programming. “Welding” exemplifies a standard, but demanding robot application.

In order to simplify the programming between the different levels, a DSL could be introduced for each level. In this chapter we discuss DSLs and the usage of the APPLAB tool to implement the end-user level. At first we used a transparent application-specific level which only reflects the motion control level, thus, making it available at the end-user level. We have also considered different language constructs needed for the DSL of the application-specific level.

ABB's robot programming language RAPID[1] [ABB94] is used at the end-user level, and since that is the level we were interested in studying we chose to use RAPID as our base language. It is a Pascal-like language especially designed for robot programming. The fact that each robot vendor has designed its own robot language is often criticized by the rest of the software engineering community (reinvention of computer programming). New languages are, however, designed to better support the robot programmer in typical manufacturing situations. In fact, the variety of different robot programming languages shows the need for DSLs. In the ABB case, the special properties of the RAPID language and its run-time system include:

- Backward execution. This is mainly useful for motion statements which then makes the robot move backwards along its programmed path.
- Persistent variables. A variable declared as persistent (`PERS`) remains after a power failure. The robot can automatically resume execution and continue its task.
- Numerical variables. Floating point and integer values are declared as the same type (`NUM`). Their internal representation is of no interest to the user of the robot.
- Modules. Dynamically loaded libraries which are only loaded into memory when needed.
- Placeholders. Non-defined parts of a program can be represented by placeholders. The program is then syntactically correct but upon attempting to execute a placeholder an execution error occurs. Yet undefined robot poses are also allowed in the language. During execution the user is prompted to define these robot poses via a teach-in procedure, in order to adopt to the actual position of a workpiece.

An embedded interpreter, developed within ABB Robotics, for the RAPID language is delivered together with new ABB robots today. Note that interpretation, instead of compilation, is not a problem concerning execution speed (the mechanical robot is slower anyway), but the interpretation has to take place on the target computer to ensure timely operation. When developing a language such as RAPID, or when extending it for special application areas like welding or gluing, appropriate language-design tools are, of course, highly desirable.

---

1. or ARL, *ABB Robot Language*, which was its original name within ABB Robotics.

6.3 Experimental Setup

We will now investigate how to support the design and implementation of the language of the end-user level. An overview of the experimental setup is given in Figure 6.2. The APPLAB system has been tailored to be used as a programming and control interface to the motion control system of an ABB Irb-6 robot. Functionality for communicating with the motion control system via an *embedded execution environment* on a target computer has been added, and a DSL based on the robot programming language RAPID has been implemented in APPLAB. The DSL includes the full syntax of RAPID extended with some instructions for controlling the robot arm.

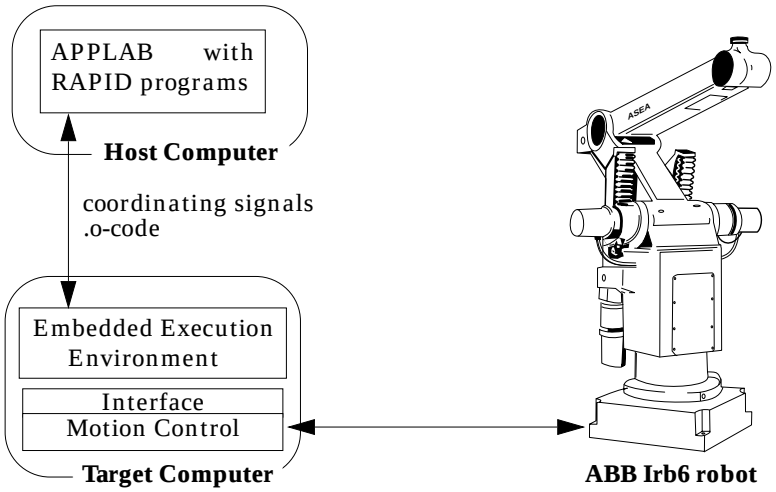

Figure 6.2 APPLAB runs on the host computer and communicates with the target computer which controls the robot.

6.3.1 Robot Hardware and Interfaces

The Irb-6 robot used in our setup has five degrees of freedom, which means that there are five motors moving the five joints of the robot. On each motor there is a sensor for reading its angular position. The *Motion Control* software running on the target computer controls the robot by reading the sensors of the robot, computing the necessary control signals to achieve the desired actions of the robot, and sending these signals to the robot's actuator hardware. The existing motion control software of the robot laboratory and its interface was used, see [Nil96a] for further details. The *Interface* of the motion control system consists of a number of library routines written in C and Modula-2. These routines provide, for example, access to the current position of the robot, and control of the robot arm and the gripper. The target computer is a VME-based Motorola 68040 board with a network interface connected to Sun workstations which are used for software development and for running software tools such as APPLAB.

6.3.2 The Embedded Execution Environment

It is highly desirable to be able to change and reload the robot program while the robot is working. This implies loading and binding the compiled application program to software already running in the target system, and also to deallocate the program when not used any more. This is dealt with in the embedded execution environment (see Figure 6.3) using a software technique with cross compiled 'plug-in' code-pieces [Nil96a] which are host controlled and managed over the network via the *load server*. The load server provides the *dynamic linker* with the necessary addresses for resolving all references in the code of the application program. Once the code is loaded into memory the *Interpreter* running on the target computer is used to execute the dynamically linked and loaded programs that control the robot, and to control it with a small hand-held terminal (*teach-in*). Basic *coordinating signals* like `load`, `run`, `step`, and `stop` are used for communicating with the embedded execution environment, and controlling the loading and running of APPLAB programs.

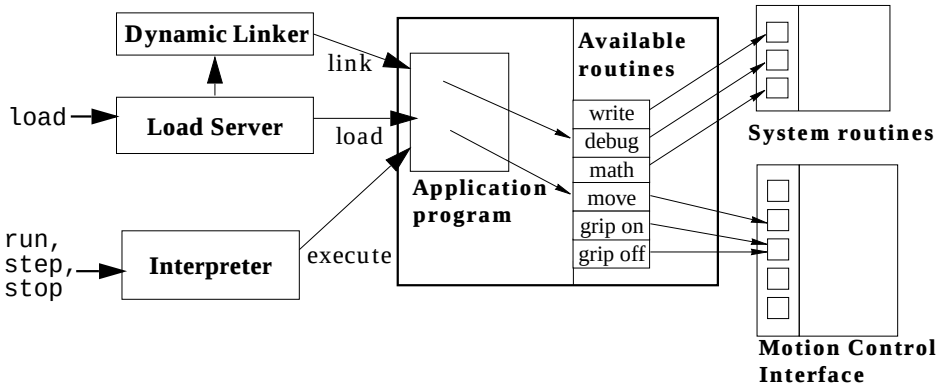

Figure 6.3 The embedded execution environment of the target computer.

6.3.3 The APPLAB-Robot Connection

Figure 6.2 shows how APPLAB communicates with the robot via the embedded execution environment; by coordinating signals and .o code. Functionality for sending and receiving the coordinating signals has been added to APPLAB. The .o code sent to the embedded execution environment may contain instructions accessing the routines of the motion control, as well as calls of a *hook-instruction* which temporarily halts the execution of the application program. This hook-instruction is used to implement step-wise execution of RAPID programs and is explained further in Section 6.4.1.

6.4 Implementing the End-User Level in APPLAB

An extended and modified version of RAPID was implemented in APPLAB. Instead of implementing full interpretation (backwards and forwards) and binary code generation, we decided to take the short-cut to map RAPID to C code and to only support the sequential execution model. A standard C compiler can then be used to generate executable code for the hardware actually used. This approach was taken since we are interested in studying the language level, and not the details of the run-time system.

The RAPID to C translation was conveniently done by using the attribute grammar support supplied by APPLAB. An OOSL aspect of the RAPID grammar was used to describe the mapping from RAPID to C. A `code` attribute and equations defining this attribute were declared for each grammar rule. The C-translation of a RAPID program can then be generated by evaluating the `code`-attribute of the root of the RAPID program tree. Figure 6.4 shows how a `MOVELIN`-statement in RAPID is translated to C, using the embedded execution environment by accessing a `MoveLinear`-function. This is later executed by a call to a routine in the basic motion control system. A more complete example is shown in Figure 6.5 which contains a snapshot of an APPLAB session. The code generation for the robot-specific instructions `move_linear` and `grip_on`, is shown in the OOSL window. The `PinTray` window shows part of a RAPID program which gets pin by pin out of a tray and puts them at some `goalPos`. Part of the C code generated for the `PinTray` program is shown in the `C-code` window. The calls to `NextInstruction` are used to allow step-wise execution of the program currently running on the robot.

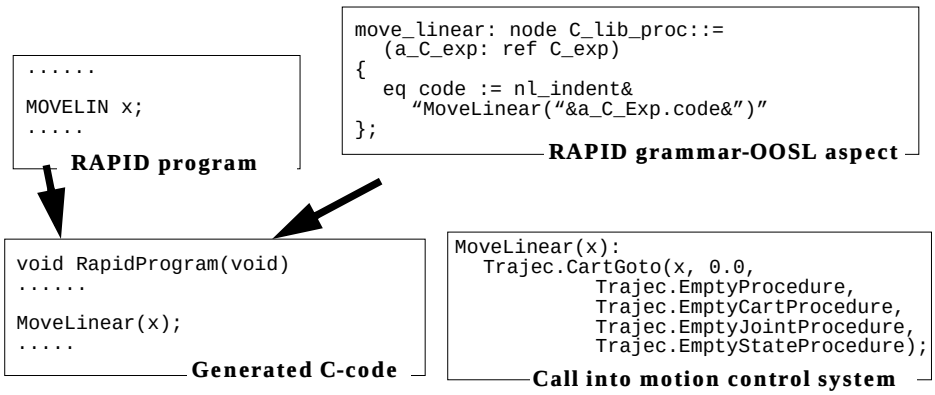

Figure 6.4 An example of accessing motion control routines in an end-user program. The "Empty" arguments of the `Trajec.CartGoto` call are dummies; these formal parameters are used for more advanced motions.

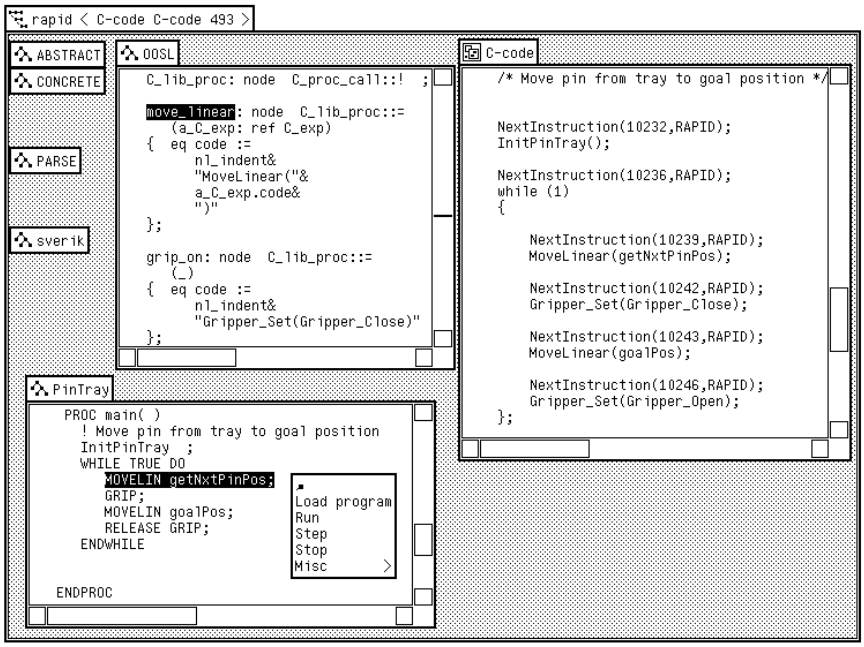

Figure 6.5 A screenshot from APPLAB showing the RAPID program `PinTray` and its corresponding C-code. The RAPID-to-C translation is defined in the OOSL window.

6.4.1 Execution of Application Programs

Functionality for receiving and sending the coordinating signals mentioned in Section 6.3.3 has been added to APPLAB. By using these signals the system communicates with the embedded execution environment. The end-user controls the execution of robot programs from APPLAB via a menu containing the commands **Load program**, **Run**, **Step** and **Stop** corresponding to the coordinating signals mentioned in Section 6.3.2. Upon issuing the **Load program** command APPLAB performs code generation for the current program, i.e. generates C code and compiles it. The cross-compiled code is then sent to the embedded execution environment along with a `Load`-signal. The application program is then dynamically linked and loaded into the memory of the target computer. When the end-user issues a **Run** or a **Step** command, the corresponding signal is sent by APPLAB to the interpreter of the embedded execution environment which launches the execution of the currently loaded program.

Debugging Support for RAPID Programs

Debugging support for programs written in APPLAB was achieved, as mentioned earlier, by invoking a *hook-instruction* for each instruction of the RAPID program. When the application program is executed, control is returned to the interpreter at

each such hook-instruction. The interpreter decides whether to continue executing, or to stop. While performing step-wise execution of an application program a `step`-signal indicating the current point of execution is sent to the invoking APPLAB session. This point is received from the invoked hook instruction. Upon receiving this signal APPLAB locates the instruction in the application program and highlights it to indicate the current point of execution. Execution of the robot program is resumed by the interpreter when the user issues a **step** or a **run** command which sends the corresponding signal to the embedded execution environment.

Generating the calls to this hook-instruction was straight forward. The OOSL grammar describing the code generation for RAPID was extended to also generate a call to the hook-instruction for each RAPID instruction, together with an identifier of the current position in the application program.

6.5 A DSL for the Application-Specific Level

The RAPID language is in itself a DSL for robot programming, but when considering the end-user level of an application like gluing or welding, the problem domain changes. Instead of general robot movements the user is concerned with application-specific actions like, for example, turning the welding equipment on and off. In standard RAPID this is achieved by using a module (library) containing the application-specific routines. The syntax of procedure calls in RAPID is such that they can be perceived as being part of the language (i.e. without parentheses). Even so, the problem of formally unexpressed conventions for correctly using the routines remains.

The Problem of Modal Settings

The definition of desired language extensions is a topic of its own, but one published [Nil96a] desired feature is to support the robot programmer concerning so called *modal settings* such as base coordinate systems, definition of end-effector geometry (what kind of equipment is attached to the robot arm), speed limits along programmed paths, or some application-specific settings of external equipment. Such settings are typically made via procedure calls that change some logical state of the system, or global variables that are manipulated directly. Since the programmer then explicitly has to restore the settings, this technique often gets hard to manage when robot programs grow in size and complexity. This is mainly a problem within so called on-line programming when the programmer may edit and execute individual statements in any order, not necessarily in sequence. This may affect the modal settings, and since the equipment is used during the programming, the settings of the equipment must agree with those made in the robot program. In robot systems today, this is handled manually by the programmer, and when not correctly done the result may be unexpected (and sometimes dangerous) robot behaviour. Without going into the details of industrial robot programming, let us illustrate how DSLs can help solve this type of problem.

As an example, assume that we have a robot language in which we can express linear motions between defined poses using a `MOVELIN` statement; a motion starts

from the current pose of the robot arm and the robot hand is moved in a straight line to the pose defined by the coordinate following the keyword `MOVELIN`. Then, as an attempt to handle the modal settings in a structured way, we want to extend the language in the following two ways:

- Possibility to express motion coordinates relative to an intermediate coordinate system which we call a `FRAME`. Programming a sequence of motions relative to a certain `FRAME` should be supported. Teach-in programming of such motions requires the frame to be properly initialized to ensure that the motion specification really is relative to the frame. Otherwise, when the frame is moved due to a relocation of its corresponding work-piece, the robot motions will not be properly adjusted, and an unexpected robot motion (back to the old location of the work-piece) will occur.
- Dedicated support for arc-welding applications. This is related to the previous item in that we need a DSL extension in terms of a 'local welding scope', or block, i.e., the system should ensure that welding is started and stopped properly when such a block of the robot program is entered and exited. This application-specific block has been named `ARCWELD`. An `ARCWELD` statement accepts welding parameters (such as voltages, currents, wire feed etc.) defining how to perform welding within its scope.

The original RAPID language, including its motion specification package, provides a fixed set of intermediate frames which can be used as optional arguments to procedures, but the frames are data and not supported by the language. A deeper analysis [Nil96a] shows that interactive programming and evaluation of individual statements, as done in the original ABB system, implies that the `FRAME` and `ARCWELD` features should be supported by the language. Further, note that `ARCWELD` is an application-specific construct, but a robot as such is a general-purpose machine. This means that this construct has to be added afterwards either as a language extension, or by utilizing an appropriate abstraction mechanisms of a general-purpose robot language. This would support the correct usage of modal settings as well as make it possible to correctly initialize the robot when the user explicitly sets the point of execution during teach-in or debugging.

Solution 1: Language Extension

Since extending an existing language is not a trivial operation, and because software engineers in general think of a language as something statically defined, current industrial robot languages do not (yet) provide this type of programming support for modal settings. But, when using a language laboratory like APPLAB language constructs like FRAME and ARCWELD can quickly and easily be added to an existing language. An example program using the desired language extensions `FRAME` and `ARCWELD` is shown in Figure 6.6. While executing different parts of the program the affected modal settings are handled by the language implementation, like calling `startweld` for an `ARCWELD` block. A similar solution for hierarchical editors (which are used in the ABB RAPID system) was presented in [Nil96a].

```
MOVELIN home;
FRAME base_plate
   ! Coordinate system implicitly set
   MOVELIN start_pose;
   ARCWELD \I:=10 \U:=60
      ! startweld 10, 10 implicitly called
      MOVELIN mid_pose;
      MOVELIN end_pose;
      ! stopweld implicitly called
   END;
   ! Coordinate system implicitly reset
END;
MOVELIN clean;
```

Figure 6.6 Proper management of modal settings can be ensured by implementing special language constructs which deal with the settings.

Solution 2: Inline Procedure Specialization

The *inline procedure specialization* of the BETA language [LMN93] solves the problem of application-specific blocks in an elegant way. A procedure in BETA can be specialized using the inheritance mechanism of OO languages. An *inner*-construct in the body of the procedure code is used to indicate where the code of the specialization is to be executed. Such procedure specializations may appear inline, avoiding the declaration of a new specialized procedure. Consider the following example, where FRAME and ARCWELD have been defined using an extension of RAPID that supports inline procedure specializations (in RAPID syntax):

```
PROC ARCWELD( \num I, \num U)
! Block for welding

   startweld I,U;
   INNER;
   stopweld;
ENDPROC
```

```
PROC FRAME( CoordSystem base )
! Block for setting a local frame

   VAR CoordSystem oldBase;

   oldBase:=currCoordSystem;
   setCoordSystem base;
   INNER;
   setCoordSystem oldBase;
ENDPROC
```

RAPID has here been extended with an `INNER` construct. In the `ARCWELD` procedure, `INNER` is used to encapsulate any specialization of `ARCWELD` with calls to `startweld` and `stopweld` which turns the welding equipment on and off in a correct fashion. The `FRAME` procedure sets the coordinate system to the value given as a parameter, and after executing possible specialization code resets the original coordinate system. The program of Figure 6.6 can be rewritten, by specializing the procedures ARCWELD and FRAME, resulting in the following code:

```
MOVELIN home;
FRAME base_plate
   MOVELIN start_pose;
   ARCWELD \I:=10 \U:=60
      MOVELIN mid_pose;
      MOVELIN end_pose;
   END; ! ARCWELD
END; ! FRAME
MOVELIN clean;
```

Note that the syntax is identical to Solution 1. The difference lies in the implementation of the application-specific blocks, FRAME and ARCWELD.

Comparison of the Solutions

Inline procedure specialization is the most general of the two proposed solutions. Once the needed abstraction mechanisms are implemented, any type of application-specific blocks can be created. Extending the language with additional constructs is simpler to implement, but results in more work in the long run since the language has to be extended for each desired type of application-specific block.

We are using APPLAB and the developed robot interface for rapid prototyping of both kinds of solutions. An attribute grammar is used to expressing the semantics of the extension. The extension of the syntax is trivial. Once the RAPID language is defined within APPLAB, language extensions such as introducing an ARCWELD or a FRAME construct can be implemented within a day, including testing on the real robot. Introducing the more general extension of inline procedure specialization could also be done with a reasonable amount of work in a future version of APPLAB which provides static-semantic support for object-oriented languages (which is needed for specifying 'subclassing' of procedures). The extension of a base language like RAPID will be further supported in future releases of APPLAB when the modular and multi-layered grammars presented in Chapter 7 are implemented.

6.6 Experience and Future Work

We have used APPLAB in a case study on the development of DSLs and integrated programming support for industrial robots. The programming of industrial robots is a complex task with the need for DSLs at several different levels. The study has confirmed the usefulness of the APPLAB system for DSL development, and also pointed out a number of interesting areas for future development of the system.

The use of a structure-oriented environment proved important because it relieves programmers of having to remember detailed syntax or semantics for an experimental, changing language; the system automatically provides guidance such as menus for structure-oriented and semantic editing. The language implementation was changed and updated hundreds of times. The support for incremental language design was then invaluable, as well as APPLAB's rapid language prototyping features. We expect this approach; a structure-oriented incremental language laboratory based on immediate computation, to be very suitable for the development and experimentation with extensions of the base robot language to different specialized robot application areas.

As the language specification grew we experienced a need both for semantic editing support of the language specifications, as well as the possibility to split the specification into several modules, as described in Chapter 7, which would make it easier to handle the complexity of the language implementation. As a result of this a small case study was performed on introducing such support for the abstract meta grammar (see Section 3.6.1).

So far, we have only looked at the end-user level and introduced a language for that level. If we continue by looking at the application-specific level and introduce a DSL for that level, support for *multi-layered grammars* will be needed in APPLAB. That is, support for implementing an extension to a base language in terms of programs written in that base language, which is also described in Chapter 7.

The motivation of this case study was to evaluate APPLAB's suitability as a tool for DSL development and programming, as well as identify desirable extensions, which turned out to be multi-layered languages, modular languages, and semantic editing support of language specifications. A more extensive evaluation of the system is presented in Chapter 8 where APPLAB is measured against the requirements we have set on DSL tools.

Chapter 7

Grammar Modularization

This chapter sketches a technique for supporting reuse of language specifications within an integrated interactive environment like APPLAB. The introduction is followed by a discussion of object-oriented modular grammars and multi-layered grammars, which are then further explained by an example. Related work within attribute grammars and systems supporting grammar reuse is discussed. The chapter is concluded with a summary and future work.

7.1 Introduction

Modularization is as important for language specifications as it is for programs. Dividing a specification into separate modules is a way of dealing with its complexity which increases rapidly as the different aspects of a language are defined. Modularization also supports *reuse*, *extension* and *adaptation* of existing specifications. Since most programming languages contain roughly the same basic building blocks, like statements, expressions and procedures, allowing the reuse of existing specifications reduces the amount of work needed to design a new language.

A language specification can be modularized in different ways. For example, a module may contain a complete description of part of a language, e.g., expressions, or it may contain a description of a certain aspect of a language like its concrete syntax or code generation. The most flexible approach is to allow a combination of both. That is, allow a language to be *extended* by adding a module containing descriptions either of additional language constructs, or of an additional aspect to the language. We have chosen to support both types of modularization by a combination of *modular* and *object-oriented grammars*; *object-oriented modular grammars*. A grammar module is then used to extend a grammar specification with new language constructs while the object-oriented features are used to adapt, i.e. override by inheritance, a certain aspect of the grammar.

An extension of this modularity is used in *multi-layered grammars* which support the construction of a new language *on top of* another one. This is of interest in a number of applications, for example, robot programming. A multi-layered grammar extends a base language with framework-specific language extensions, i.e. language constructs corresponding to the functionality of a framework. The imple-

mentation of these framework-specific language constructs is similar to syntactical macros [Lea66]. They are defined in terms of the base language and the routines of the framework. Expansion is handled internally by the system allowing the programmer to use the language extensions (or macros) without having to consider their implementation.

The current version of APPLAB does not contain support for modular or multi-layered grammars. This chapter outlines techniques for such grammars which we plan to implement in the APPLAB system.

7.2 Object-Oriented Modular Grammars

As a first step towards supporting a general modularization mechanism for grammar specifications, we have chosen to look at specifying a language by extending a base language. This corresponds to reusing one grammar module, i.e. the base language, when designing a new language. It is desirable to reuse as much as possible of the existing specification, as well as add new features and adapt the existing ones. By importing the base language, rather than copying its definition, changes made to the base language can be automatically incorporated into the extended language.

In order to make the reuse of a language specification practically useful, mechanisms for adapting and extending reused language specifications are required. We have used the inheritance mechanism of the object-oriented paradigm to achieve this. A new language construct can then inherit all the attributes and rules of an existing construct, thus, reusing the original specification. We plan to investigate additional techniques for adapting constructs in an imported module. The excluding of certain (imported) constructs has been discussed by Aksit et al. in [AMH90] and is used in TaLE [JKN95, KM95]. TaLE also allows for the adaptation of components of an inherited language construct, like, e.g., the declaration part of a procedure declaration. The exclusion and adaptation of language constructs is not directly supported by OO modular grammars, but can be simulated by defining static-semantic rules that makes them illegal.

Figure 7.1 shows the architecture of a simple OO modular grammar. Two additional statement constructs, `Loop` and `While`, are added in the `Loops` module by importing the `Stmts` module and then specifying the new constructs as subclasses of `Stmt`. The existing features of `Stmt` can be modified by reimplementing the corresponding rules in the specifications of the new language constructs and new features can be added by defining additional attributes. This allows for great flexibility. One can either reuse all or part of an existing language construct. Also, the inheritance mechanism incorporates the new constructs with the existing ones. For example, the `Or` construct specified in the `Bools` module, as a subclass of `Exp`, can occur (syntactically) at any place in a program where `Exp` is legal. Furthermore, new attributes can be added to a non-terminal by using the `addto` declaration of OOSL [Hed92a]. In our example, the non-terminal `Exp` is extended in the `Bools` module with an `error` attribute and default equations for that attribute. This means that `Add` also contains an `error` attribute in the resulting Toy language.

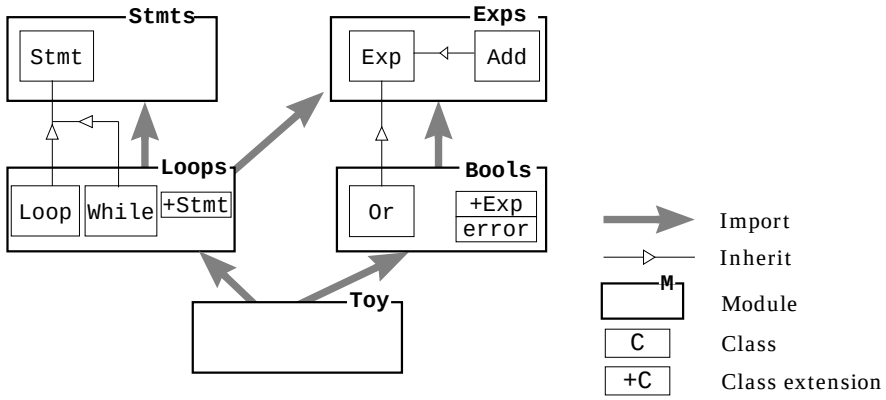

Figure 7.1 The architecture of a modular object-oriented grammar.

7.3 Multi-Layered Grammars

In a multi-level system, like robot programming (see Chapter 6), several program layers are used in programming an application. Implementing a DSL for each level is a way of capturing and enforcing the conventions of each such layer. The dependencies of the resulting language hierarchy could be expressed and supported by a *multi-layered grammar*. Such a grammar consists of a number of program and language layers. A program layer which implements some functionality, consists of a library, a framework or an application program. A language layer introduces new syntax and semantics, encapsulating the functionality and conventions of the program layer at the previous level.

Figure 7.2. shows an example of expanding the base language, G_{BL}, for a real-time framework, `RT Framework`. The framework is programmed in the base language, whereas the application program is programmed in the extended language G_{BL+RT}. The grammar for the extensions, G_{RT}, can access the framework to implement the code generation of the new language constructs.

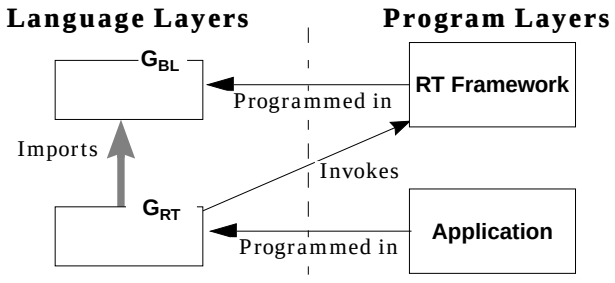

Figure 7.2 The architecture of a multi-layered grammar consisting of a base language, G_{BL}, a language extension for real-time programming, G_{RT}, a framework (RT Framework), and an application program expressed in the extended language, G_{BL+RT}.

A programmer using an extended language is primarily interested in seeing the new language constructs and their syntax. The system, on the other hand, needs to consider how the new constructs are implemented in terms of the base language and the framework, in order to correctly perform code generation and static-semantic checking. In a system whose internal representation of programs is based on abstract syntax trees (ASTs), *expansion trees* can be used for representing the implementation of the new language constructs. Similarly to macros which are not expanded until compile time, expansion trees are not constructed until an attributed syntax tree is evaluated. This can be done by using *Higher-Order Attribute Grammars* [VSK89] which allow a node in the tree to be defined by the value of an attribute. We want such nodes to be invisible to the user, but used by the system to perform attribute evaluation, and, thus, code generation and static-semantic checking. Since the structure of an expansion tree follows the base language the system can evaluate its attributes in the same way as for the other parts of the program tree.

7.4 An Example

Real-time programming is an example of programming for which an object-oriented framework, or a library, often is used. The framework then handles the concurrency and real-time aspects of the programming. Correct usage of the framework is crucial for the correctness of the program. For example, consider a monitor [Hoa85] which according to its definition is intended to be a language construct providing mutual exclusion. Part of the real-time framework is as follows:

```
class Monitor;
begin
   procedure Enter;
      (* Called first of all in each monitor procedure. *)
   procedure Exit;
      (* Called last in each monitor procedure.*)
end;

class Event(mon);
   ref Monitor mon;
begin
   procedure Await;
      (* The monitor must have been entered. *)
   procedure Cause;
      (* The monitor must have been entered. *)
end;
```

A monitor is defined as a class with the methods `Enter` and `Exit`. An example of a convention for the `Monitor` class is that the method `Enter` always should be called on entering, and the procedure `Exit` always should be called on exiting a method of the `Monitor` class. By designing a monitor language construct in a language extension, G_{RT}, this convention could be described and enforced by the language, supporting the programmer in correctly using the monitor concept.

The object-oriented framework for real-time programming is expressed in the language G_{BL}. Two example programs for a buffer are shown in Figure 7.3. Example 1 uses the framework directly in a base language, G_{BL}, while Example 2 uses

Example 1 (G_{BL})

```
begin
   Monitor class Buffer(size);
      integer size;

   begin
      ref (Event) NonFull, NonEmpty;
      character array
                thisBuffer(1..Size);
      integer inp, outp, cnt;

      character procedure get;
      begin
         Enter;
         while cnt=0 do NonEmpty.Await;
         get := thisBuffer(outp);
         outp := mod(outp,size) + 1;
         NonFull.Cause;
         Exit;
      end;

      procedure put(character ch);
      begin ..... end;

      .....

      NonFull :- new Event(this Buffer);
      NonEmpty:- new Event(this Buffer);
   end Buffer;
end
```

Example 2 (G_{BL+RT})

```
begin
   monitor Buffer(size);
      integer size;
   events NonFull, NonEmpty;
   begin

      character array
               thisBuffer(1..Size);
      integer inp, outp, cnt;

      character procedure get;
      begin

         while cnt=0 do await NonEmpty;
         get := thisBuffer(outp);
         outp := mod(outp,size) + 1;
         cause NonFull;

      end;

      procedure put(character ch);
      begin ..... end;

      .....



   end Buffer;
end
```

Figure 7.3 Two examples of a Buffer. Example 1 uses the framework (see page 86) directly while Example 2 uses it via a real-time language extension.

the extended language, G_{BL+RT}. The examples show the procedure `get` which returns the next item in the buffer, using a monitor to exclude other processes from accessing the buffer data. In Example 1, the `get` procedure contains calls to the monitor methods `Enter` and `Exit`, and the body of the Buffer class contains statements for creating the event variables. Note that Example 2, which uses the language extensions, does not contain any such (explicit) calls to the classes or methods of the framework. Instead, this is dealt with in the language implementation. Note, that the encapsulation of the body of the monitor procedure could have been expressed in a language like BETA [LMN93] that allows for the specialization of actions using an `inner` construct. This illustrates that the base language used to implement a library or framework determines to which extent the syntax and semantics of the domain entities can be described.

7.4.1 Using O-O Modular Grammars for Language Extension

Consider the language used above in Example 2, where a base languages is extended by adding a monitor construct. Using the OO modular grammars, the new construct `MonitorDecl` can be declared as a subclass of the existing `Decl` declaration. Part of the specification of the extended language, G_{RT}, is as follows:

```
MonitorDecl::=                                                    (1)
   Decl("monitor" Id "events" Events "begin" Block "end")
{  (* expansion definition:                                       (2)
        equations defining the expansion tree impl. the
        monitorDecl construct in terms of the framework.*)
   (* static semantics:                                          (3)
        equations which check that all variables are private
        (protected) to this Block. *)
   (* code generation:                                          (4)
        equations which compute the code to generate by
        using the expansion tree.*)
};

addto ProcDecl                                                    (5)
{  (* expansion definition:                                      (6)
        equations defining an expansion tree for
        declarations appearing within a MonitorDecl *)
   (* code generation:                                          (7)
        equations which compute the code to generate by
        using the possible expansion tree. *)
};
```

The object-oriented attribute grammar notation OOSL [Hed92a] is used in the example. The `addto` construct in (5) inserts additional rules and attribute declarations into the original declaration of `ProcDecl`. In (1) the abstract and concrete syntax are specified, introducing the new keywords `monitor` and `events`, and stating that a `MonitorDecl` declaration contains an `Id`, a list of `Events`, and a body (`Block`). Static-semantic rules that ensure that variables declared within a `MonitorDecl` are protected, that is, only accessible from within the `MonitorDecl` are specified in (3). The code generation for the `MonitorDecl` construct (4) involves generating calls to the framework using the defined expansion tree (2). This includes encapsulating any procedures declared within the `MonitorDecl` with calls to the framework. This is done by extending the code generation (7) of the `ProcDecl` definition (5) to include the defined expansion tree (6).

7.4.2 Implementing A Language Construct With Expansion Trees

Part of the AST for the program in Example 2 (see Figure 7.3) is shown in Figure 7.4. The monitor construct of the extended language, G_{BL+RT}, is represented by a `MonitorDecl` node. This node has an `Id` ("`Buffer`"), an `Events` node, a `Block` node containing the procedure `Get`, and an expansion tree. The expansion tree of `MonitorDecl` (specified in (2) above), maps the `Buffer` monitor construct to a `Buffer` class which is a subclass of the `monitor` class of the real-time framework. The `get` procedure of the buffer also has an expansion tree, (specified in (6) above), which encapsulates the body of the procedure with calls to the methods `Enter` and `Exit`. In order to do this the expansion tree refers back into the program AST, in this case the body of the `get` procedure. The calls to `Enter` and `Exit` will be bound to the corresponding methods of the `monitor` class of the framework via the `Buffer` class declared in the expansion node of `MonitorDecl`.

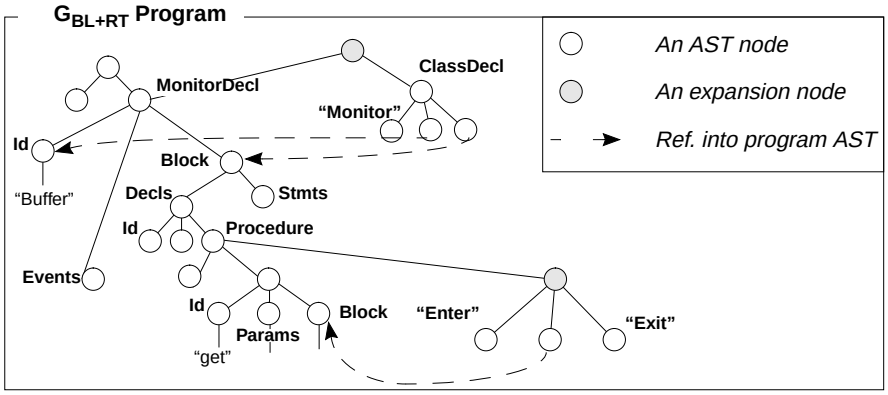

Figure 7.4 Part of the abstract syntax tree for a program (see Figure 7.3) expressed in the extended language, G_{BL+RT}.

7.5 Related Work

The problems of dealing with *complexity*, *extensibility* and *reusability* in language specifications, have been identified by several researchers, e.g., [AMH90, MM94, Bos95b]. A common approach in dealing with these problems is to apply the same concepts as are used for dealing with them in programming languages, to language specifications, and attribute grammars. In doing so, the same benefits as are gained for the programming level, are made available at the language implementation level. Four main approaches based on AGs are identified in [Paa95]; *attribution paradigms*, which allow for a more efficient specification of attribute values, *structured AGs*, which use ideas from block-structured programming languages for structuring a specification, *modular AGs*, with which a specification can be constructed by combining a number of modules, and *object-oriented AGs*, which incorporate the OO paradigm. Our approach is a combination of *modular* and *object-oriented AGs*.

Modular AGs cover a number of different types of modularization, varying in which part of the grammar is modularized; either the nonterminals and productions, which is used in the ASF/SDF meta environment [Kli91], or the attributes and equations for a certain aspect of the language, which is used in, e.g., MAGGIE [DC90], Eli [Kas96], the Synthesizer Generator [RT89] and in FNC-2 [JBP90, JP91]. Modularizing the nonterminals results in modules which fully implement part of a language. This is useful for language subsets which are identical in several languages, for example, expressions. Modularizing the different aspects of a language caters for easily adding new aspects to a language. The modular AGs (MAGs) used in MAGGIE consist of classes of attribute computations (or equations) which are merged, by the system, with a CFG resulting in a monolithic AG. A different approach to modularizing according to language aspects is to allow a module to consist of any number and combination of nonterminals, productions, attributes and equations. Since this allows for the most flexible packaging it is this type of modularity we wish to support in combination with the object-oriented approach.

We have used the inheritance mechanism of the object-oriented paradigm to support the reuse and adaptation of existing language constructs. In [AMH90], the term 'grammar inheritance' is used to indicate that a grammar (describing the syntax) inherits productions from a 'superclass' grammar, with the possibility to exclude certain constructs. Our approach is different in that we say that a class corresponds to a production which inherits attributes and semantic functions from a 'superclass', which is also a production. This is similar to the approach taken in Mjølner/Orm [Hed89a] and in OOAG [SK90]. This type of inheritance for language constructs allows for overriding and extension of attributes and rules, but not for exclusion of language constructs. Utilizing the polymorphism of the object-oriented paradigm means that no extra mechanisms need to be introduced into the grammar notation for supporting overriding and extension of attributes and semantic functions. The inheritance mechanism also supplies the 'glue' for combining the grammar modules.

TaLE [JKN95, KM95, HHK97] is an object-oriented framework for language implementation which also allows for the reuse of language concepts by inheritance. In addition to overriding and extending the semantics of an inherited concept TaLE also supports the adaptation of the components of that concept. Take the monitor example presented earlier in this chapter. Assuming the following class declaration

```
classDecl::= (superClass, "class", idUse, formalPars, classBody)
```

a monitor construct, could be specified as a subclass of `classDecl` as follows

```
monitorDecl ::= classDecl (monitorSuper, -, -, -, -)
```

The declaration of `superClass` in `classDecl` is then refined to `monitorSuper`, a subclass of `superClass` using the `Monitor` class of the real-time framework as the superclass. The remaining components of `classDecl` are left unchanged (denoted by -). This kind of adaptation (or refinement) which is not supported by out OO modular AGs, could be very useful for, e.g., adapting existing language specifications to a DSL.

In [Bos95b], a delegating parser is described which allows a grammar specification to be split into several grammar modules. The parser is then implemented as several separate, delegated, parser components which each parse according to a grammar module. This allows for separate generation (analogous to separate compilation) of a parser. APPLAB works by supplying language-specific behaviour by interpreting grammar specifications, and does not have a generation phase, and, thus, not the problem of having to update generated code after a change of the grammar specification.

Implementing a new language on top of an existing one can be done by using a preprocessor which transforms the source code into code of the base language. When supporting source-code debugging and reporting of run-time errors these transformations need to be reversed in order to present the affected source code and not the code actually executed. This is done in KHEPERA [FNP97] and in ASCENT [GCN92] which also supports the reuse of programming environment of the base language. We expect the multi-layered grammars outlined here to be a

suitable starting point for providing similar support for source-level debugging. The reverse mapping of execution points to source code could then be achieved by taking advantage of the fact that the expansion trees (which are used to generate the executable code) are connected to the corresponding nodes in the original program tree.

7.6 Summary and Future Work

Framework-specific language extensions which supply framework-specific syntax and enforce the conventions of the framework make it safer and easier to use object-oriented frameworks. An object-oriented grammar notation allows new language constructs to be added by subclassing existing grammar rules. The new constructs can either reuse or respecify the semantic properties of the existing language constructs. When working in a programming environment which represents programs as abstract syntax trees *expansion trees* can be used to implement new language constructs in terms of a framework. Static-semantic checking and code generation, as well as source code debugging, can then be supplied for the extended language. We are planning to add such support for the design and implementation of framework-specific language extensions to our language laboratory, APPLAB.

There are several interesting issues to look into concerning the use and implementation of framework-specific language extensions. For example, when a base language is changed this affects languages implemented as an extension of that base language, and programs expressed in the changed language. A mechanism is then needed for transforming the affected languages and programs into consistent versions according to the new version of the base language, by for example using techniques like those in the TransformGen system [GKL94]. It is also desirable to be able to allow multiple language extensions. That is, to combine several language extensions into one extended language. There may then be combinations of language constructs which interfere with each other. Can such clashes be avoided or resolved automatically?

A lot of work remains to be done in this area. Both in implementing the proposed techniques and in doing further research into the area. Due to its declarative nature APPLAB is a suitable platform for performing such research, and trying out new ideas in practice.

Chapter 8

Evaluation

This chapter discusses how well APPLAB, and the techniques used to implement the system, supports the DSL approach. The system is evaluated against the requirements for DSL support listed in Chapter 2. Measurements of APPLAB's performance in terms of time and memory consumption are also presented.

8.1 APPLAB as a DSL Tool

The main features which make APPLAB suitable for DSL programming and rapid language prototyping (which is appropriate for the design of DSLs) are:

- Language-sensitive editing
- Immediate feedback on changes
- Version control

In addition, the execution model used in the robot case study provided DSL programming support. The techniques for supporting reuse and extension of language specifications described in Chapter 7 are not implemented in the current version of the system. This feature can therefore not be evaluated in practice, but we judge that it will provide the system with a powerful mechanism for quickly and easily developing new DSLs. In Chapter 2, on page 18, a list of requirements for the DSL approach was given. APPLAB will now be evaluated against these requirements. Table 8.1 contains a summary of this evaluation.

The Programming Level		
R1. Interactive language-sensitive programming environment	Full	8.1.1
R2. Source code debugging	Indirect	8.1.4
R3. Version control of language specifications and programs	Full	8.1.3
R4. Transformation of programs to new language version	Part	8.1.3
The Language-Design Level		
R5. DSL programming support for the language-design level	Full	8.1.1
R6. Rapid prototyping of languages	Full	8.1.2
R7. Collaborative language design	Some	8.1.3
R8. Building a language from existing language blocks	No	8.1.5
R9. Language extension	No	8.1.5
R10. Multi-level languages	No	8.1.5
R11. Translating a problem domain into a 'good' DSL	No	8.1.5
R12. Portability	Indirect	8.1.4

Table 8.1: APPLAB's current DSL support

8.1.1 Language-Sensitive Editing

Requirement R1.('Interactive language-sensitive programming environment') is fully met by APPLAB. Programs are edited in a structure-oriented editor which supports semantic editing, as well as language-sensitive text editing. The editing is an interaction where the user issues editing commands like picking a language construct for the current editing focus from a menu containing all legal constructs. The selected construct is then inserted into the program document. Similarly, all legal names at the current editing focus can be accessed via a names-menu. These two types of menus are of special assistance to a user unfamiliar with the details of the current programming language. Also, during text editing, all syntactical errors are reported, allowing the programmer to correct them, before inserting the edited text into the program document.

The environment used for language design and implementation in APPLAB is also used for program development. Thus, requirement R5.('DSL programming support for the language-design level') is also fully supported by APPLAB since the same support is offered at the language-design level as for the programming level. That is, an interactive language-sensitive environment which aids the DSL designer in developing a language. For example, two different kinds of names-menus have been specified for the abstract grammar. The first type of menu contains all available productions (non-terminals). The second type of menu contains the

names of all non-terminals which need to be defined. These two names-menus are of great assistance in specifying an abstract grammar since the language designer can immediately see which non-terminals are left to specify and, when defining a new production rule, which non-terminals are available.

8.1.2 Immediate Feedback on Changes

Requirement R6.('Rapid prototyping of languages') is fully supported by APPLAB. Immediate feedback on changes made either to a language specification, or to a program is given in the interactive editing environment. Upon editing a language specification the resulting changes are immediately seen in the program window. This was very valuable in the development of the RAPID language implementation. The language specification was revised hundreds of times, at first to create a correct specification of the base language, and later in experimenting with different language extensions. The rapid language prototyping provided by the immediate computation mechanisms of APPLAB made it possible to quickly try out the new constructs while getting them 'right'.

The immediate feedback mechanism is also vital to the syntactic and semantic editing support. The structure-oriented and names menus always reflect the current status of the program and of the grammar.

8.1.3 Version Control

Requirement R3.('Version control of language specifications and programs') is fully met by APPLAB. The immediate updating of a program to follow a new language specification, as described in the previous section, applies only to example programs within a grammar document. For separate program documents, APPLAB supplies version control (inherited from the Orm environment), keeping track of which grammar version was used to construct the program document. This allows newer experimental versions of the DSLs to be developed while existing programs continue to use an older stable version. When a newer stable version has been established, the user can decide whether a program should continue to use the old version of the grammar, or if it should be updated to conform to the newer version. This scheme was very valuable in order to make sure that existing programs did not become inconsistent due to language changes, and to allow the user to control when updates to newer DSL versions were made.

Requirement R4.('Transformation of programs to new language version') is partly met by APPLAB. The limited support provided by the system to transform a program to adhere to a new grammar version (or a grammar aspect to adhere to a new meta-grammar version) proved invaluable in handling the numerous small grammar changes. On a couple of occasions when we actually updated the language description formalisms themselves (the meta grammars), we were able to update our DSL specifications in a controlled manner. Extending this transformation support is, however, a very important issue if a system like APPLAB is to come into production use.

Requirement R7.('Collaborative language design') is also only partly met by the system. The version system in APPLAB supports parallel development of variants of documents allowing, for example, two developers to work on different aspects of a grammar. However, because merging of variants is not yet supported, we could not make full use of this feature in our case study. For future releases of APPLAB we would like to include advanced fine-grained versioning and incremental merging of programs and grammars similar to the COOP/Orm system [MAM93, MA96].

8.1.4 Program Execution

Requirements R2.('Source code debugging') and R12.('Portability') are not directly supported by the system but both were achieved in the case study on robot programming. Portability of the application programs was obtained by generating C code for the robot programming language, RAPID. Any standard C compiler can then be used to generate binary code for the desired target machine. This approach allowed us to specify the code generation for RAPID with a minimal amount of effort, and produce executable code for the target computer.

Source code debugging was implemented by using an (external) component (the embedded execution environment) for administering the program execution. This component communicates with APPLAB via explicit signals and through adding calls to a hook instruction in the code generation of the application program. See Section 6.3 and Section 6.4 for further details.

8.1.5 Reuse of Language Specifications - Future Work

Requirements R8-10 are currently not met by APPLAB but will be supported in the future by the object-oriented modular grammars and the multi-layered grammars presented in Chapter 7. These grammars promote the reuse of language specifications, thus, enabling a faster development of new languages, requiring less effort than with today's system.

Requirement R8.('Building a language from existing language blocks') will be directly met by the object-oriented modular grammars which allow a language to be constructed by combining a number of (existing) grammar modules. This feature will also provide support for Requirement R9.('Language extension') by allowing a base language specification to be combined with additional grammar modules resulting in an extension of the base language.

The multi-layered grammars directly meet requirement R10.('Multi-level languages') as well as offering some support for requirement R11.('Translating a problem domain into a 'good' DSL'). Techniques for DSL design primarily require a method for identifying the domain entities and their corresponding language constructs. One approach which has been proposed [RJ97] is to first design an object-oriented framework which is then 'pushed into' a language implementation. The multi-layered grammars mentioned above, would provide support for the development of such a language.

8.2 Performance

8.2.1 Time Consumption

The times for various computations and updating of internal data structures used in APPLAB are summarized in Table 8.2. The times relate to the three different DSLs; the abstract meta grammar, a toy language and RAPID, and run on a Sun SPARC Ultra 1. The example program used for each language each consists of 100 lines of code.

Computation	*Languages*		
	Grammar *Abstract (8 prods)*	*Small DSL* *Toy (36 prods)*	*Large DSL* *RAPID (212 prods)*
Build structure-editing menus	0.01	0.05	0.8
Redisplay of unparsed text	0.2	0.1	0.2
Computation of data structure for parser	0.02	0.1	0.6
Computation of data structure for attribute evaluation	0.6	5.4	30.0
Code generation	-	-	1.7
Compute names menu	0.02[a]	0.1	-
Static-semantic rror marking	0.6	1.3	-

Table 8.2: APPLAB computation times (in seconds).

a.names menu of all declared nonterminals

Most of the computations are less than a second and do not disturb the interaction in any significant way, even for a full-sized language like RAPID. We have not tried to optimize the implementation, and we think it is possible to reduce these figures so that the updates become completely unnoticable to the user. The one update which gives some problems is the computation of the data structures for the attribute evaluator. This problem will be solved if we implement incremental attribute evaluation using Door AGs. We will then be able to compute these data structures incrementally, by describing the data structures themselves using a Door AG. (In contrast to standard AGs, Door AGs allow the definition and computation of data structures with objects and references, and not only simple attribute values.) Incremental evaluation of attributes will also allow the marking of static-semantic errors to be done continuously and not interactively, as in the current version.

8.2.2 Memory Consumption

The memory requirements for an APPLAB session vary depending on which language is used. The object code of the system itself requires 3 MBytes and additional memory is needed for grammars and programs, and internal structures for parsing and attribute evaluation as shown in Table 8.3. In addition, temporary memory (about 0.1 MB) is also needed to perform various computations.

Purpose	*Languages*		
	Abstract (8 prods)	*Toy (36 prods)*	*RAPID (212 prods)*
ABSTRACT	71	100	295
CONCRETE	73	121	417
PARSE	-	68	73
OOSL	432	653	1 585
PROGRAM (100 loc)	121	83	123
Parser structure	7	29	116
Evaluation structure	173	215	674
	876	1 269	3 283

Table 8.3: Memory usage (kByte)

8.2.3 Time and Memory Trade-Offs

Using the lazy feature of the attribute evaluation increases the amount of memory required, but decreases the time needed to perform the evaluation. It is up to the language designer to select which attributes should be handled using lazy evaluation, so as to achieve optimal performance both in time and in memory. Table 8.4 shows the time and memory requirements for accessing the two different kinds of names menus of an abstract grammar consisting of 100 production rules (or loc). The time required to generate the names-menu of all used, but undefined names is large (29 seconds) when using pure demand attribute evaluation. This is caused by re-evaluating the complete set of defined names for the grammar for each used name, in order to determine if the used name is defined. By letting the attribute containing the complete set of defined names for the grammar be a lazy attribute its value is only calculated once thus resulting in a faster access time, (0.72 s the first time the names menu is accessed and 0.02 s the following times.) The additional memory requirement (12.9 kByte) is small compared to the amount of memory needed for the evaluation structure (173 kByte, i.e. 7%). And, most importantly the response time using the purely demand driven evaluation (29 s) is

unacceptable in an interactive editor. In conclusion, choosing the lazy attributes wisely leads to a small increase in memory consumption compared to the amount of time gained.

Names menu contents	*Demand evaluation*	*Lazy evaluation*	
	Time (s)	*Time (s)*	*Memory (kB)*
Defined names	0.17	0.17	+18.2
		0.02	
Undefined names	29	0.72	+12.9
		0.02	

Table 8.4: Time and memory requirements for the first and following accesses of the names menus of an abstract grammar of 100 production rules (or loc).

8.2.4 Summary

The time and memory requirements for the APPLAB system are reasonable, even for larger programs and grammars. The memory requirement for running APPLAB with a full-sized programming language like RAPID is less than 7 MBytes, and most of the computations, like accessing structure-editing and names menus, are performed in less than a second. Thus, no annoying delays occur in the interaction with the user. The one computation which does cause concern is the generation of the evaluation structure used for attribute evaluation. As previously mentioned, this will be alleviated by implementing incremental attribute evaluation based on Door AGs, and by describing the evaluation structure itself with DoorAGs.

8.3 Conclusions and Future Work

Looking at Table 8.1, we see that a third of the listed requirements on support for DSLs are fully met by the current version APPLAB; the system provides an interactive language-sensitive environment both for language and program development, rapid language prototyping is supported, as well as version control of the different versions of a language. Another third of the requirements are partly or indirectly supported. The system provides some (limited) updating of programs to newer versions of the language, source code debugging is indirectly supported, as well as portability of DSL programs. The remaining requirements (which are not supported by the current version of APPLAB) all fall into the category of future work. A more complete support for the transformation of programs to a newer language version needs to be included in the system, as well as support

for merging different variants of a language development in order to support collaborative language design. The object-oriented modular grammars and the multi-layered grammars presented in Chapter7 provide the support needed to extend and reuse existing languages, as well develop multi-layered languages, which are covered by the four remaining requirements.

A lot of work remains to be done in implementing additional features and improving the performance of some of the existing ones, mainly the attribute evaluation. But, already today APPLAB provides valuable support to language designers and DSL programmers. This has been shown by using the system in a challenging application area, namely robot programming.

Chapter 9

Conclusions and Future Work

9.1 Contributions

The main contribution of this thesis is a combination of techniques for supporting rapid language prototyping and language-sensitive editing support, and the implementation of a platform (APPLAB) showing that these techniques are viable. The motivation for developing these techniques was to provide interactive tool support for domain-specific languages. Such languages have many advantages, but one of the greatest drawbacks is the problem of developing and maintaining a language at a reasonable cost. The DSL technique could be more widely used if tools were available which support the interactive development of DSLs and DSL programming.

APPLAB (*APPlication Language LABoratory*) is an interactive language-design environment as well as an interactive programming environment. It is a further development of the grammar editor of the Mjølner/Orm system. The grammar-interpreting structure-oriented editor (inherited from the Orm system) supports rapid language prototyping of the syntax of a language. This feature has been further enhanced in APPLAB by providing language-sensitive text editing, as well as rapid prototyping of the static semantics of a language.

The technique for *grammar-interpreting object-oriented recursive-descent parsing*, described in Chapter 5, has been integrated with the structure-oriented editor, thus, supporting language-sensitive text editing of any language structure of a program (or grammar) within the structure editor. The parser uses the same grammars as the structure-oriented editor, and, if needed, disambiguating rules can be specified.

The technique for *grammar-interpreting static-semantic analysis*, described in Chapter 4, has been implemented in the APPLAB system, thus, supporting rapid prototyping of semantic aspects of a language, e.g., code generation. The object-oriented attribute grammar notation, OOSL, is supported and evaluated by a demand-driven attribute evaluator. This simple, but powerful evaluation strategy is used to support the *grammar-interpreted advanced editing features* like names-menus, and interactive static-semantic error reporting implemented in APPLAB.

Due to the grammar interpretive nature of the introduced tools, the new features (like text editing and advanced editing) are supported both at the programming level and at the grammar level, thus, supporting both programmers and language designers. Furthermore, the features automatically adapt to changes in the grammar, thus, supporting rapid language prototyping.

In order to allow the specification of static-semantic rules for block structured languages, a few predefined attribute types, like list, set and dictionary, were added in APPLAB for OOSL. For this purpose, a *framework for predefined attribute types* (described in Section 4.4) was developed which facilitates the adding and changing of attribute types. Each predefined attribute type has an interface description in the OOSL grammar to which their implementing classes are connected. This means that the static-semantic analyser does not need to take any additional consideration for such predefined types.

A number of case studies have been performed with APPLAB. In the most extensive one, which was on robot programming, APPLAB was *integrated with an industrial robot*, (see Chapter 6), and a DSL based on the robot programming language RAPID was implemented in the system. The domain of robot programming contains a number of interesting programming problems. The different levels of robot programming (discussed in Section 6.2) each represent a problem domain (usually) implemented as a library. We have identified two approaches for providing improved programming support for each level (see Section 6.5):

1. extending a base (robot) language with constructs encapsulating the concepts of a particular application domain
2. extending a base (robot) language with more powerful abstraction mechanisms which allow the concepts of the application domain to be concisely expressed in the base language

The first approach results in a hierarchy of languages which could be supported by the *multi-layered grammars* outlined in Chapter 7. The second approach, which is a more general one, is supported by the object-oriented modular grammars, also outlined in Chapter 7, in combination with the proposed support for Door AGs (see the next section). Both of these techniques are as yet unimplemented but are intended to be further developed and integrated into APPLAB in order to support the reuse of language specifications both for language extensions and for multi-layered languages.

9.2 Future Work

There are many areas in connection with this work which provide interesting options for future research. Three main directions, which are particularly interesting, will be outlined here.

Support for Object-Oriented Languages

The object-oriented AGs which are currently supported by APPLAB allow for the specification of simple block-structured languages. In order to also support object-oriented languages it is desirable to use Door AGs which is an extension of standard AGs for which efficient incremental evaluation can be performed. The support for object-oriented languages in APPLAB will also lead to a number of other advantages:

- Object-oriented languages are well suited to modelling domain entities. Providing support for OO languages would, thus, further enhance APPLAB as a tool for DSL development. For example, the implementation of the in line procedure-specialization mechanism described in Section 6.5 would then be straight forward.
- Advanced editing support for OOSL grammars. The static semantics of the object-oriented attribute grammar notation, OOSL, used in APPLAB, could be described using Door AGs. This would allow for the specification of names-menus and static-semantic error reporting for OOSL grammars.
- Incremental updating of the evaluation structure for OOSL. The internal data structures currently generated by the static-semantic analyser to perform attribute evaluation could be described by a Door AG. Incremental evaluation of Door AGs would then support the incremental updating of the required data structure. Thus, avoiding the time consuming generation phase.

DSL Run-Time Support

An important aspect of supporting DSL programming is to provide run-time support at the source-code level. The run-time environment used in the robot case study provides this support but the APPLAB system as such does not. In a realistic setting it is vital that a system such as APPLAB does provide a run-time environment which is easy to configure for a specific language. It is an interesting area of research to look into this and develop techniques for DSL run-time support.

Reuse of Language Specifications

As previously mentioned, the techniques for supporting reuse of language specifications have not yet been implemented in APPLAB. Allowing for the reuse of language specifications would enhance the language development by allowing the development of a new language to be done by combining a number of existing building blocks. This would reduce both the time and the effort needed to produce a new language and its implementation. The object-oriented modular grammars and the multi-layered grammars outlined in this thesis are a first step towards providing such support. We intend to implement these grammars in the APPLAB system and further develop techniques for language reuse.

References

[ABB94] ABB Robotics. *ARL Reference Manual*, 1st edition, February 1994.

[AKW79] A. V. Aho, B. W. Kernighan and P. J. Weinverger. AWK - A pattern scanning and processing language, *Software Practice and Experience*, 9, pages 267-280 (April 1979).

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AMH90] M. Aksit, R. Mostert and B. Havrekort, *Compiler Generation Based on Grammar Inheritance*, Technical Report 90-07, Department of Computer Science, University of Twente, February 1990.

[AWB+94] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa. Abstracting Object Interactions Using Composition-Filters. In *Object-Based Distributed Programming*, R. Guerraoui, O. Nierstratz, M. Riveill (eds.), LNCS 791, Springer-Verlag, 1994.

[ANSI89] American National Standards Institute, *American National Standard for Information Systems-Programming Language C*, X3.159-1989.

[ALEL96] *Proceedings of ALEL'96. Workshop on Compiler Techniques for Application Domain Languages and Extensible Language Models.* Jan Bosch and Görel Hedin (eds.). Linköping, April -96, Sweden. Technical report LU-CS-TR:96-173. Dep. of Computer Science, Lund University. http://www.dna.lth.se/Research/ProgEnv/ALEL/ALEL96/ElectronicProceedings/ElectronicProc.doc.html

[AP94] Isabelle Attali and Didier Parigot. *Integrating Natural Semantics and Attribute Grammars: The Minotaur System*. Rapport de recherche no 2339, INRIA, September 1994.

[BGV90] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The Pan Language-Based Editing System For Integrated Development Environments. *SIGSOFT, Software Engineering Notes*, 15(6), December 1990.

[BBH+94] J. Bell, F. Bellegarde, J. Hook, et al. *Software Design for Reliability and Reuse: A Proof-of-Concept Demonstration*. In TRI-Ada'94 proceedings, pages 396-404, November, 1994. ftp://cse.ogi.edu/pub/pacsoft/papers/triada.ps

[Ben86] J. Bentley. Little Languages. *Comm of the ACM*, 29(8), August 1986.

[Bja96] Elizabeth Bjarnason. APPLAB: User's Guide (version 1.2). Technical Report LU-CS-IR:96-01, Department of Computer Science, Lund University, April 1996.

[BH96] Elizabeth Bjarnason and Görel Hedin. A Grammar-Interpreting Parser in a Language Design Laboratory. In *Proceedings of the Poster Session of CC'96 (International Conference on Compiler Construction)*. P. Fritzson (ed.), pages 15-24, LiTH-IDA-R-96-12, Dept. of Computer Science, Linköping University, Sweden, April 1996.

[BHN97] Elizabeth Bjarnason, Görel Hedin and Klas Nilsson. APPLAB-An Application Language Laboratory. Technical Report LU-CS-TR:97-188, Department of Computer Science, Lund University, 1997.

[BCD+88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. In *Proc. of the ACM SIGPLAN Conference on Practical Software Development Environments*, pages 14–24, 1988.

[BK86] Jon L. Bentley and Brian W. Kernighan. GRAP - A Language for Typesetting Graphs. *Communications of the ACM*, 29(8), August 1986.

[BTA96] Lodewijk Bergmans, Bedir Tekinerdogan and Mehmet Aksit. Modular and Composable Extensions to Smalltalk using Composition Filters. In *Proceedings of Workshop on Extending the Smalltalk Language*, held in conjunction with OOPSLA'96.

[Bos95a] Jan Bosch. *Layered Object Model Investigating Paradigm Extensibility*. Ph. D. thesis, Dep. of Computer Science, Lund University, Sweden, 1995.

[Bos95b] Jan Bosch. Parser Delegation An Object-Oriented Approach to Parsing. In TOOLS EUROPE '95.

[Bos96] Jan Bosch. *Compiler Support for Extensible Languages*. In ALEL96.

[CMA94] Luca Cardelli, Florian Matthes and Martín Abadi. *Extensible Syntax with Lexical Scoping*. Research Report 121, February, 1994. Digital Research Center, 130 Lytton Avenue, Palo Alto, California 94301.

[CP93] S. Chamberlain and R. Pesch. *Using LD, the GNU linker*. Free Software Foundation, Inc. and Cygnus Support, URL: http://www.cygnus.com/library/ld/ld_toc.html, March 1993.

[Cra89] J. J. Craig. *Introduction to robotics: mechanics and control*. Addison-Wesley, second edition, 1989.

[DMN68] O.-J. Dahl, B. Myhrhaug and K. Nygaard. *SIMULA 67 Common Base Language*. Publ. S-2. Norwegian Computing Centre, Oslo. 1968.

[DM82] A. J. T. Davie and R. Morrison. *Recursive Descent Compiling*. Ellis Horwood Limited Publishing, 1982.

[DALEL96] *Summary of the DALEL Workshop: Design of Application Languages and Extensible Language Models*. Jan Bosch and Görel Hedin, June 1996. http://www.cs.auc.dk/normark/NWPER96/proceedings/subworkshop-B.html

[DK97] Arie van Deursen and Paul Klint. *Little Languages: Little Maintenance?* In [DSL97b].

[Deu89] L. Peter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 System. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Volume II: Applications and Experience*, pages 57-71. Addison-Wesley, Reading, MA, 1989.

[DSL97] *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, USA, October, 1997.

[DSL97b] *Proceedings of DSL'97, First ACM SIGPLAN Workshop on Domain-Specific Languages*. University of Illinois, Computer Science Report. 1997. http://www-sal.cs.uiuc.edu/~kamin/dsl.

[DGH+84] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. *Programming Environments Based On Structured Editors: The MENTOR Experience*, chapter 7, pages 128–140. In Barstow et al., editors. *Interactive Programming Environments*. McGraw-Hill Book Company, 1983.

[Duc97] Stéphane Ducasse. Message Passing Abstractions as Elementary Bricks for Design Pattern Implementation. In *Proceedings of LSDF'97, Workshop on Language Support for Design Patterns and Frameworks*, held in conjunction with ECOOP'97, Jyväskylä, Finland, 1997.

[DBP95] Stéphane Ducasse, Mireille Blay-Fornarino and Anne-Marie Pinna. A Reflective Model for First Class Dependencies. In *Proceedings of OOPSLA'95*, pages 265-280, Austin, October 1995. ACM. RR-95-24.

[DC90] G. D. P Dueck and G. V. Cormack. Modular Attribute Grammars. *Computing Journal*, 33, 2, 164-172, 1990.

[Ekb96] B. Ekberg. *Grammatikstyrd LSA-editor*. (In Swedish) Master's thesis. LU-CS-EX:96-3. Dept. of CS, Lund University, Sweden. March 1996.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1986.

[Eng84] J. Engelfreit. Attribute Grammars: Attribute Evaluation Methods. In B. Lohro (ed.), *Methods and Tools for Compiler Construction*. Cambridge University Press, pages 103-138. 1984.

[FNP97] Richard E. Faith, Lars S. Nyland and Jan F. Prins. KHEPERA: A System for Rapid Implementation of Domain-Specific Languages. In DSL97.

[FMY92] R. Farrow, T. J. Marlowe and D. M. Yellin. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Maciso). ACM, New York, pages 223-234.

[FJKS85] P. H. Feiler, F. Jalili, G. E. Kaiser, and J. H Schlichter. ISDS: A Retrospective of the DOSE System. Tech. report, Siemens R & T Laboratories, 105 College Road East, Princeton, NJ 08540, 1985.

[FJS86] Peter H Feiler, Fahimeh Jalili, and Johann H. Schlichter. An Interactive Prototyping Environment for Language Design. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, volume II, pages 106–116. IEEE, 1986.

[Gam95] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software*. Addison-Wesley, 1995.

[GCN92] D. Garlan, L. Cai and R. L. Nord. A Transformational Approach to Generating Application-Specific Environments. In *Proc of 5th ACM SIGSOFT Symposium on Software Development Environments*, 17(5),1992.

[GKL94] D. Garlan, C. W. Krueger, and B. Staudt Lerner. TransformGen: Automating the Maintenance of Structure-Oriented Environments. *ACM TOPLAS*, 16(3):727–774, May 1994.

[GHL+92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M Waite. Eli: A Complete Compiler Constructions System. *Comm. of the ACM* 35, 1992.

[GWB+96] William G. Griswold, Richard Wolski, Scott B. Baden, Stephen J. Fink and Scott R. Kohn. Programming Language Requirements for the Next Millennium. Presented at the Workshop on Software Engineering and Programming Languages, SEPL, June 1996. file://ftp.cs.washinton.edu/pub/se/sepl/griswold.ps.Z

[GMN97] Carl Gunter, John Mitchell and David Notkin. Strategic Directions in Software Engineering and Programming Languages. To appear in *ACM Computing Surveys*, 1997.

[Gus90] Anders Gustavsson. Software Configuration Management in an Integrated Environment. Dep. of Computer Science, Lund University, Sweden. 1990.

[HHK97] Maarit Harsu, Juha Hautamäki and Kai Koskimies. A Language Implementation Framework in Java. In LSDF97, 1997.

[Hed89a] Görel Hedin. An Object-Oriented Notation for Attribute Grammars. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, S. Cook (ed.), British Informatics Society Ltd., Nottingham, 329-345, 1989.

[Hed89b] G. Hedin. *A Recursive-Descent Parser Based On Object-Oriented Grammars*. Draft, 1989.

[Hed92a] Görel Hedin. *Incremental Semantic Analysis*. Ph. D. thesis, Department of Computer Science, Lund University, Sweden, March 1992.

[Hed92b] Görel Hedin. Context-Sensitive Editing in Orm. In K. Systä el al., editors, *Proceedings of the Nordic Workshop on Programming Environment Research, Tampere, Finland*. Tampere University of Technology. Software Systems Lab. TR 14. January 1992.

[Hed94] Görel Hedin. An Overview of Door Attribute Grammars. In *Proceedings of the 5th International Conference on Compiler Construction (CC'94)*, LNCS 786, Springer-Verlag, pp. 31-51, Edinburgh, April 1994.

[Hed97a] Görel Hedin. Attribute Extension - A Technique for Enforcing Programming Conventions. In *Nordic Journal of Computing. Special Issue on Programming Environments*, 4, pp 93-122, 1997.

[HB96] G. Hedin and J. Bosch. Summary of the DALEL Workshop: Design of Application Languages and Extensible Language Models. June 1996. In DALEL96. http://www.iesd.auc.dk/~normark/NWPER96/proceedings/subworkshop-B.html

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental Generation of Parsers. *IEEE Transactions on Software Engineering*, SE-16(12), December 1990. Also in Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, (June 1989), pages 179-191.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall Int., London, UK, 1985.

[Hud96] Paul Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28(4es), December 1996.

[Hud97] Paul Hudak. *The Promise of Domain-Specific Languages*. Keynote address at DSL97. http://www.haskell.org/hudak/dsl.ppt

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publishing Company, 1992.

[Jal83] F. Jalili. A General Linear-Time Evaluator for Attribute Grammars. *SIGPLAN Notices*, 18(9):35-44, September 1983.

[Joh79] S.C. Johnson. Yacc: Yet Another Compiler Compiler. *Unix Programmer's Manual, 7th Edition*, Bell Telephone Laboratories, January 1979.

[Joh88] Ralph E. Johnsson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22-35, June/July 1988.

[JBP90] M. Jourdan, C. Le Bellec and D. Parigot. The OLGA Attribute Grammar Description Language: Design, Implementation and Evaluation.In *Proc of International Conference on Attribute Grammars and their Applications*. LNCS , volume 461, Springer-Verlag, New York, 222-237, 1990.

[JP91] Martin Jourdan and Didier Parigot. Internals and Externals of the FNC-2 Attribute Grammar System. In *Attribute Grammars, Applications and Systems*. Lecture Notes of Computer Science, volume 545, Springer-Verlag, New York,485-504, 1991.

[JKN95] E. Järnvall, K. Koskimies and M. Nittymäki. Object-Oriented Language Engineering with TaLE. *Object-Oriented Systems* 2, pp. 77-98. 1995.

[KK85] G. E. Kaiser and E. Kant. Incremental Parsing Without a Parser. *The Journal of Systems and Software*, 5(2):121–144, May 1985.

[Kam93] Sam Kamin. *Report of a Workshop on Future Directions in Programming Languages and Compilers*. January, 1993. file::/ftp.cs.washington.edu/pub/se/sepl/kamin-report.ps.Z

[Kam96] Sam Kamim. The Challenge of Language Technology Transfer. *ACM Computing Surveys* 28(4es), December 1996.

[KH97] Samuel N. Kamin and David Hyatt. A Special-Purpose Language for Picture-Drawing. In DSL97, October 1997.

[Kas96] Uwe Kastens. *Construction of Application Generators Using Eli*. In ALEL96.

[Ker82] B. W. Kernighan. *PIC - a Language for Typesetting Graphics*. Software - Practice and Experience. 12:1, pages 1-21, 1982.

[Kie95] R. B. Kieburtz. *Software Design for Reliability and Reuse. Method Definition*. ftp://ftp.cse.ogi.edu/pub/pacsoft/final_report/sdrr.ps

[Kli91] P. Klint. *A Meta-Environment For Generating Programming Environments*, pages 105–124. Springer-Verlag New York Inc, 1991.

[KLLM93] J. L. Knudsen, M. Löfgren, O. Lehrmann-Madsen, and B. Magnusson, editors. *Object-Oriented Environments The Mjølner Approach*. Prentice Hall, 1993.

[Kos89] Kai Koskimies. Software Engineering Aspects in Language Implementation. In *Proceedings of the 2nd Workshop on Compiler Compilers and High Speed Compilation*. Lecture Notes in Computer Science, vol. 371, D. Hammer, Ed. Springer-Verlag, New York, 39-51.

[Kos96] Kai Koskimies. *Frameworks and Application-Oriented Languages*. Published in ALEL96.

[KM95] K. Koskimies and H. Mössenböck. Designing a Framework by Stepwise Generalization. In *Proceedings of 5th European Software Engineering Conference (ESEC'95)*, Sitges, Spain. Lecture Notes in Computer Science 989, Springer, pp. 479-498. 1995.

[KV92] K. Koskimies and J. Vihavainen. Incremental Parser Construction With Metaobjects. Technical Report A-1992-5, Dep. of CS, University of Tampere, Finland, P.O.Box 607, SF-33101 Tampere. Finland, Nov. 1992.

[LSDF97] *Proc of LSDF'97 Workshop on Language Support for Design Patterns and Object-Oriented Frameworks*. Jan Bosch, Görel Hedin and Kai Koskimies (eds.) Held in conjunction with ECOOP'97. Research Report 6/97. Dep of CS, University of Karlskrona/Ronneby, SE-375 25 Karlskrona, Sweden.

[Lea66] B. M. Leavenworth. *Syntax Macros and Extended Translation*. CACM, 9:790-793, 1966.

[LMN93] Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley Publishing Company, 1993.

[LS84] M. E. Lesk and E. Schmidt. Lex - a Lexical Analyzer Generator. Bell Laboratories, Murray Hill, New Jersey. Reprinted in *Unix Programmer's Manual: Supplementary Documents*, distributed with 4.2BSD, 1984.

[Mad89] William Maddox. *Semantically-Sensitive Macroprocessing.* Ph. D. thesis. Report no UCB/CSD 89/545. Computer Science Division, University of California, Berkeley, California 94720.

[Mag93] Boris Magnusson. *The Mjølner Orm System*, chapter 1, pages 11–23. In Knudsen et al. KLLM93, 1993.

[MA96] Boris Magnusson and Ulf Asklund. Fine Grained Version Control of Configurations in COOP/Orm. In Proceedings of ICSE 6th Workshop on Software Configuration Management (SCM-6), I. Sommerville (Ed.), Berlin. Springer LNCS 1167, pp 169-174. 1996.

[MAM93] B. Magnusson, U. Asklund, and S. Minör. Fine-Grained Revision Control for Collaborative Software Development. In *SIGSOFT'93, First ACM SIGSOFT Symposium on the Foundations of Software Engineering* (FSE-1). 1993.

[MHM+90] B. Magnusson, G. Hedin, S. Minör, et al. An overview of the Mjølner/Orm environment. In J. Bezivin et al., editors, *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, pages 635-646, Paris, June 1990. Angkor.

[Min90] Sten Minör. *On Structure-Oriented Editing.* Ph. D. thesis, Department Of Computer Science, Lund University, 1990.

[Min92] Sten Minör. Interacting with Structure-Oriented Editors. *International Journal of Man-Machine Studies*, 37(4), October 1992.

[MH93] Sten Minör and Görel Hedin. *Grammar Interpretation in Orm*, chapter 20, pages 297–306. In Knudsen et al. KLLM93, 1993.

[MM94] Sten Minör and Boris Magnusson. Using Mjølner Orm as a Structure-Based Meta Environment. Published in *Structure-Oriented Editors and Environments*, L. Neal and G. Swillus (eds.), 1994.

[Nil96a] Klas Nilsson. *Industrial Robot Programming.* Ph. D. thesis, Department of Automatic Control, Lund Institute of Technology, May 1996.

[Nil96b] Thomas Nilsson. Application-Domain Languages: Some Suggestions for Research. In [ALEL96]. 1996.

[NBL98] K. Nilsson, A. Blomdell and O. Laurin. Open Embedded Control. To appear in *Real-Time Systems - The International Journal of Time Critical Computing*, 1998.

[Not85] D. Notkin. The GANDALF Project. *The Journal of Systems and Software*, 5(2):91–105, May 1985.

[Nyb97] Mats Nyberg. *Integrated Robot Programming.* Master thesis. LUNDFD6/ NFCS-5098/1--25/1997. Dept. of Computer Science, Lund University, Sweden. December 1997.

[Nor87] Kurt Nørmark. Transformations and Abstract Presentations in a Language Development Environment. Ph. D. thesis, Computer Science Department, Aarhus University, Denmark, February 1987.

[Paa95] Jukka Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, Vol. 27, No. 2, June 1995.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments.* Ph. D. thesis, University of Amsterdam, 1992.

[RT87] Thomas Reps and Tim Teitelbaum. *Language Processing in Program Editors.* IEEE Computer, Vol.20, No. 11, November 1987.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[RJ97] Don Roberts and Ralph Johnson. Patterns for Evolving Frameworks. Chapter 26 of *Pattern Language of Program Design 3*, Robert C. Martin, Dirk Riehle and Frank Buschmann (editors).Addison-Wesley, 1997.

[Sam69] Jean E. Sammet. *PROGRAMMING LANGUAGES: History and Fundamentals*. Series in Automatic Computation. Prentice-Hall, 1969.

[San78] E. Sandewall. *Programming in an Interactive Environment: the "LISP" Experience*. Computing Surveys, Vol 10, No 1, pages 35-71, March 1978.

[SG96] Mary Shaw and David Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SK90] Y. Shinoda and T. Katayama. Object-Oriented Extension of Attribute Grammars and its Implementation. In *Proceedings of the International Conference on Attribute Grammars and their Applications*. Lecture Notes in Computer Science, vol. 461. Springer-Verlag, New York, 177-191. 1990

[SEPL95] *Report on the Workshop on Software Engineering and Programming Language*, Stanford, Sept. 1995. John Mitchell. ftp://ftp.cs.washington.edu/pub/se/sepl/stanford.ps.Z

[SEPL96] Workshop on Software Engineering and Programming Languages. June 1996. http://www.cs.washington.edu/research/se/sepl/position.html

[Sta87] SIS Standardiseringsgrupp. *Databehandling Programspråk-SIMULA, Svensk standard SS 63 61 14*. SIS, 1987.

[TR81] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. In Communications of the ACM, 24:9, pages 563-573. September 1981.

[TMC97] ScottThibault, Renaud Marlet and Charles Consel. A Domain-Specific Language for Video Device Drivers: from Design to Implementation. In DSL97, October 1997.

[VG92] M. L. Van De Vanter and S. L. Graham. Coherent User Interfaces For Language-Based Editing Systems. *International Journal Of Man-Machine Studies*, 37(4), October 1992.

[VSK89] H. H. Vogt, S. D. Swierstra and M. F. Kuiper. Higher-Order Attribute Grammars. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, ACM Sigplan Notices, 24(7), 1989.

[War94] Martin P. Ward. Language Oriented Programming. *The Journal of Software-Concepts and Tools*, Vol. 15, 1994. Also ftp.dur.ac.uk:/pub/techreports/Martin.Ward

[Wat82] R. C. Waters. Program Editors Should Not Abandon Text Oriented Commands. *SIGPLAN Notices*, 17(7), July 1982.

[Win87] Winograd, T. A., *Muir: A Tool for Language Design*, Report No. STAN-CS-87-1151, Department of Computer Science, Stanford University, 1987