A Compiler Front-End for Eiffel-3

Burghardt Groeber
Olaf Langmack

Report B-92-25
December 1992

Abstract

We have written a front-end for an Eiffel-3 compiler, which is described in this paper. It was generated automatically according to the most recent language definition. The parser generates an easy-to-use abstract syntax tree, supports elementary error recovery and provides a precise source code indication of errors. It achieves good performance and supports a strict syntax check. A summary of the concrete syntax and an abstract syntax for Eiffel-3 are given as appendices.

Key words: Eiffel-3, compiler generation, abstract syntax.

Institut für Informatik
Freie Universität Berlin
Nestorstraße 8-9
W-1000 Berlin 31

`langmack@inf.fu-berlin.de`

The object-oriented programming language Eiffel-3 was chosen as basis within the HERON-Project [FJL+92]. The project investigates the transparent distribution of Eiffel-3 programs and concurrent extensions of the language [Löh92]. The implementation of a proxy generator and other precompilers resulting from this research have the need for a compiler front-end in common. When this need was identified in February 1992, version 3 of the language had already been published (see [Mey92a]) three months before. Since Eiffel-3 turned out to be a complex language – at least from a compiler construction point of view – we did not expect to see a public domain front-end in the short run. On the other hand it was not possible for us to get access to a commercial product.

1 – Front-End Implementation

The front-end of a compiler is usually separated in two parts. The scanner splits the program source code into tokens, which represent atomic syntactical entities. The parser analyses then the sequence of tokens, to check if it complies with the concrete syntax of the programming language. An internal representation of the program is generated for the purpose of semantic analysis and as a side effect of the syntactical analysis. It has been shown that an abstract syntax tree is more suitable for further compilation than the parse tree representing the concrete syntax. There is little literature about the notion of abstract syntax trees, see [Gan83] and [Noo85] for a discussion of the relevant issues. Figure 1 gives an overview of the general structure of a compiler front-end.

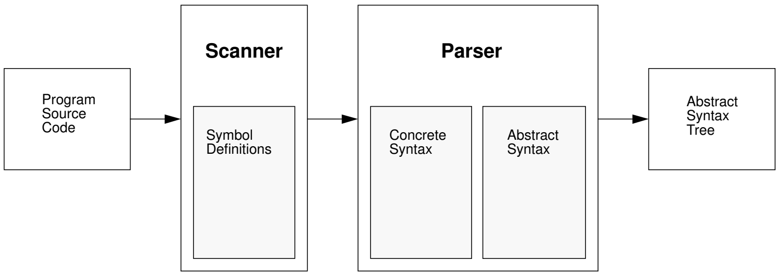

Figure 1: Front-end structure

The difficulty of implementing a compiler front-end for a given language depends mainly on the inherent lexical and syntactical complexity of the language. It may depend to some extent on the coherency of the language definition as well. On the other hand it is possible to generate all parts of a front-end automatically, based on a formal description of the respective requirements. Taking into account the complexity of Eiffel-3 and with the possibility of automatic front-end generation in mind we decided to use a front-end generator.

Front-End Generators

The role of abstract syntax trees as interface between subsequent compilation phases has been addressed by several modern compiler generation systems. A non-exhaustive reference list is [Krä85], [BS86], [GMOS88], [LMW89], [GE90], [DP90] and [GHL+92]. From these we selected the Karlsruhe compiler construction toolbox [GE90] due to its unrestricted public availability, its reported technical maturity, its complete set of generators – with respect to the complete compilation process – and the reported efficiency of the generators and the generated programs. Our expectations were met in all these aspects.

Figure 2 shows the structure of compiler front-end generation. The less obvious arrow between concrete and abstract syntax indicates the use of abstract syntax non-terminals within semantic actions of the concrete syntax. This provides guidance for tree generation during the parsing process.

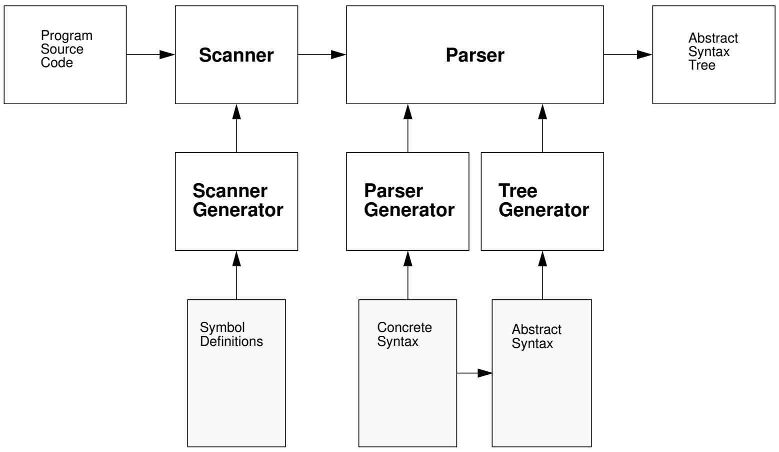

Figure 2: Front-end generation

Object-oriented front-end implementation

We identify two ways to combine object-orientation and front-end implementation: *"a rule type is modelled by a class"*, or *"a symbol is modelled by a class"*.

The ISE implementation of Eiffel-2.3 supports the first approach with a set of library classes [MN90]. The coding of a reasonably sized grammar with these classes is cumbersome, so that it is easier to implement a translator from EBNF to class source code utilizing the library classes. We know of two such translators, `yoocc` ([GW92]) and `ebnf2e` (private communication[1]). But regardless of whether such a translator is used or not, all parsers implemented with these libraries share a rather poor[2] performance and generation time.

[1] R. d'Souza, `ross@cc.gatech.edu`.

[2] Generation takes 45 minutes for a complete Eiffel-2.3 acceptor on a Sun SPARC workstation, from private communication with R. d'Souza. See also [KV92].

The second approach supports especially incremental language and parser development. This is interesting from a methodological point of view, but since Eiffel-3 is supposed to be in a stable state, we didn't consider this requirement for our tool selection. Apart from that, refer to [KV92] for research in this direction. A similar approach has been followed by [Gra92] within the Smalltalk scene.

Finally it is worth mentioning that object-orientation is approached from the compiler generation point of view as well. An object-oriented grammar specification formalism was introduced in [Gro90]; it is used by the generators of the Karlsruhe toolbox as the unique input formalism. See page 5 for a short introduction.

2 – Heron Front-End for Eiffel-3

Toolbox

The Karlsruhe toolbox provides both an LALR and LL parser generator. We used the latter, so that the reference language definition is as close as possible to the generator input.

Language definition

The definition of Eiffel-3 was first published in November 1991 in [Mey92a]. Appendix G of the book covers most of the lexical definitions, and appendix H covers most of the syntax rules in EBNF. Several "hints" on the lexical and syntactical structure of Eiffel-3 can also be found throughout the book. In July 1992 a syntax revision was informally published (see [Mey92b]). Appendices A.1-3 provide a summary of these lexical definitions and syntax rules. We regard this as the language definition. Based on it, an analysis of test data as used by the ongoing ISE and the released SIG implementation unvealed slight deviations of both implementations with respect to the published reference. These are summarized in appendix A.4 and A.5.

The HERON front-end analyses class source code according to the reference definition summarized in appendices A.1-3. It supports the concrete syntax deviations as implemented by ISE and SIG, to the extent they are known to us and documented in appendices A.4-5.

Configuration

Full-scale Eiffel implementations compile not only a single class source code file but all inherited or used source code files as well. The location of the source code of ancestors or suppliers of a class is defined in the configuration file associated with each system of classes. The structure of this file is not part of the language definition. Therefore it is vendor dependent: ISE implements the LACE language ("Language for Assembling Classes in Eiffel") as defined in appendix D of [Mey92a], while SIG implements the PDL ("program description language") and RCL ("runtime control language") languages as defined in chapter 2 and 3 of [SS91].

The recursive analysis of a system of classes is not supported by the HERON front-end, in order to keep it as vendor independent as possible, and due to the project context of our implementation: HERON investigates configuration of distributed programs, which might yield as a consequence a distribution specific configuration language.

System analysis can be implemented as follows:

1 Analyze the directly accessible source code of class A and generate the list of its immediate ancestors and suppliers. (How to generate this list is a simple exercise in using the HERON front-end, which is left to the reader.) Apply step 2 to each immediate ancestor and supplier source code.

2 Analyze the configuration file (LACE, PDL or whatever) to identify the location of the source code file in the local file system. Apply step 1 to the class source code file.

Eiffel-3 Peculiarities

The front-end generation requires a formal specification of lexical definitions and syntax rules. The Eiffel-3 definitions cause difficulties for the compiler implementor, which are worth to be mentioned:

- *Operator precedence* — The complexity of certain syntax rules – notably `Debug` and `Debug_key` – requires a parser generator which is capable of backtracking if operator precedence is to be handled during the parsing process. Since the LL parser generator used does not support backtracking, the generated abstract syntax tree does not reflect operator precedence[3].

- *Comments* — Due to their very nature comments are usually discarded in the scanning phase of the compilation process. This is not possible with Eiffel-3, since the following rule specifies a comment as a syntactical relevant entity:

  ```
  Unlabeled_assertion_clause = Boolean_expression | Comment .
  ```

  For other occurrences of comments as syntactical relevant entities B. Meyer uses the notion of *expected comments*[4]. Finally the notion of *free comments* identifies what is known as *comment* in ordinary programming languages.

- *Delimiter* — The special symbol `;` (semicolon) is sometimes used as an optional syntactical element without other meaning than for layout purposes and is sometimes required as delimiter in list constructs.

[3] To be implemented with simple transformations of the generated tree. This will be part of the next release of the HERON front-end.

[4] Be aware of inconsistencies within the language presentation! Block 112 of the syntax definition in appendix H of [Mey92a] defines "`end`" followed by an expected comment as end of "`routine`", whereas the example on page 172 uses an "`end`" *followed by a semicolon* followed by an expected comment.

- *Unused keyword* — The keyword `separate` is introduced as such in appendix G of [Mey92a] but never used in the syntax specification.

Error Handling

The Karlsruhe toolbox supports automatic generation of elementary error recovery and error indication procedures. So the generated front-end prints lexical and syntax errors and indicates their precise source code position.

Abstract Syntax Tree

The parser generates an abstract syntax tree if a class is recognized as syntactically correct. The abstract syntax tree is the interface for further processing of the class.

R. Noonan describes an algorithm (see [Noo85]) to generate automatically an abstract syntax specification from the BNF of the language, it is applied to Modula as an example. We did not apply this technique, since there is no public implementation of R. Noonan's algorithm available, and for Eiffel-3 we expect manual derivation of an abstract syntax to achieve a higher level of abstraction. Therefore our initial proposal for an abstract syntax for Eiffel-3 has been hand-crafted; it is presented in appendix B of this report.

The general syntax specification formalism which has been introduced by J. Grosch in [Gro90] is used. It is not restricted to any specific type of syntax and is used as input language for the various tools of the Karlsruhe toolbox. Other compiler generation systems use different formalisms. [HHKR89] and [LMW89] provide two examples.

The formalism is based on context-free grammars enriched by an inheritance mechanism. The tree consists of typed nodes. Every grammar rule describes a node type. A node type consists of a node type name and the names and types of its child nodes and the names and types of associated attributes. Names and types are separated syntactically by a colon. If the name of a child node is omitted, the name of the child node and the name of the associated node type are identical. Attributes are enclosed in brackets. The types of attributes are C data types or structures. If the type definition is omitted, the node is of type `int`. As an illustration see the following example:

```
Assign = Addr: Entity Expression .
Real   = Sign: Open [value: double] [pos: tPosition] .
```

The node type `Assign` has two child nodes: `Addr` of type `Entity` and `Expression` of type `Expression`. The node type `Real` has one child and two attributes: child `Sign` of type `Open` and the attributes `value` of type `double` and `pos` of type `tPosition`.

A node type described by a rule containing other node type descriptions enclosed in angle brackets introduces a supertype relation between node types, while other node types are subtypes of the defined node type. Where a node of a certain type is required, either a node of this type or a node of a subtype is allowed. See the following example:

```
Op_name        = [pos: tPosition] <
  Free_op      = [ident: tIdent] .
  Oper         = [op] .
> .
```

Within right-hand side rules where a node of type Op_name is required both nodes of type Free_op or Oper would suffice. Furthermore Free_op and Oper inherit the attribute pos, so they both have in fact two attributes.

Internally the tree is represented by a C data structure[5]. There is a data structure declaration for every node type. The tree node types are all of the same union type. This union consists of all node types together with a tag field Kind, which stores an internal node type identifier. These structures and syntax specific node generation procedures have been generated by the tree generator[6] according to the abstract syntax specification.

CEiffel Peculiarity

All types of comments are retained within the abstract syntax to adhere as close as possible to the reference definition and as a hook for syntactically transparent language extensions (see CEiffel, [Löh92]).

Test

In the same way as any kind of real-world software test is usually done by users, compiler testing is usually done by programmers. We did not try to do more than others in this respect. Only test data provided by ISE and the source files shipped with the SIG release 1.2 were used as test data for the HERON front-end. From our point of view this turned out to be satisfactory both with respect to functionality and performance issues of the first release[7]. Our intention is to make the generated source code accessible in the public domain and incorporate bug fixes sent to us by users. When the front-end finally reaches a stable state, we will put all generator input specifications in the public domain as well.

Usage

Appendix C describes how to access the HERON front-end source code and gives an example of how the generated abstract syntax tree can be processed by a simple application program. Appendix D gives a selection of operations for abstract syntax tree access as provided by the Karlsruhe toolbox.

[5]The data structure definition is part of the public accessible archive file. See appendix C, page 21.

[6]Acronym ast, "Abstract Syntax Tree Generator", see [Gro92a].

[7]F. van Breughel has presented a method to generate test data for parsers based on the attributation of the parser generator input grammar [vB89]. Further development of a full-scale Eiffel-3 language system would benefit from the availability of such a grammar based test facility. Implementation of such a facility is tempting too, since it could be implemented relatively easy as a grammar attributation scheme which itself is to be evaluated by the Karlsruhe toolbox attribute evaluator generator.

3 – Comparison

Functionality

Full-Scale Implementations

At the time of this writing there is only one full-scale Eiffel-3 implementation available, and little is known from another ongoing implementation. Table 1 gives an overview of full-scale Eiffel implementations.

Vendor	Language	Compiler	Configuration
ISE	Eiffel-2	2.3	SDF
SIG	Eiffel-3	1.2	PDL, RCL
ISE	Eiffel-3	(not released)	LACE

Table 1: Full-scale compilers

The implementation of the HERON front-end implies an automatic check of the lexical definitions and syntax rules by the used generators. This has unvealed inconsistencies in the presentation of the language definition. It allowed also to identify and describe syntax deviations of other front-end implementations.

Front-End Implementations

At the present time some front-ends for Eiffel-2.3 but none for Eiffel-3 are known. Table 2 gives an overview of the known[8] public front-ends for Eiffel-2.3.

Authors	Language	Generator	Input	Output	Reference
Blach	2.3	CoCo	concrete LL(1)	Acceptor	[Bla92]
d'Souza	2.3	ISE Libraries	concrete LL(1)	Acceptor	R. d'Souza
(University of Sheffield)	2.3	ISE Libraries	(?)	(?)	(?)

Table 2: Public Eiffel-2.3 front-ends

These front-ends for Eiffel-2.3 share the lack of abstract syntax tree generation, which is to be implemented using the mechanisms provided by the corresponding generator. This means hand-coding has to be done with the programming language associated with the generator. As opposed to this approach, the HERON front-end uses abstract syntax tree generation procedures that were automatically generated according to a formal specification. This did improve generation efficiency but established a formal interface between compilation phases as well.

[8]Existence of the Sheffield-Parser communicated by N. Hoar.

Performance

The performance of front-end generation and of the compilation itself are of immediate practical relevance. The concept of incremental compiler construction, as addressed by research in object-oriented compiler construction, does become purely theoretical if implementations need dozens of minutes instead of minutes for compiler generation or minutes instead of seconds for the compilation itself.

The generation of the HERON front-end is fast due to the use of the Karlsruhe toolbox. It takes less than 4 minutes on a Sun SPARC workstation for the entire language. This is less than 10% of the generation time if the ISE Eiffel-2.3 lexical and parsing classes are used[9].

Thanks to the use of the Karlsruhe toolbox the generated HERON front-end itself processes 4000 source lines per second including tree generation. Since it is the first in the public domain for Eiffel-3 and the front-end performance of full-scale compilers cannot be isolated[10], its performance cannot be compared with others. It achieves a very good throughput compared with what is known from front-ends for Eiffel-2.3 that are implemented using the ISE lexical and parsing libraries[11].

Acknowledgements

We thank J. Grosch (GMD) for toolbox support, P. Stephan (ISE) for providing test data, R. d'Souza (Georgia Tech) for communicating his experience with the ISE Eiffel-2.3 lexical and parsing libraries and last but not least M. Biersack. Comments of T. Wolff, I. Piens, R. Rojas and K.-P. Löhr helped to improve the quality of this report. The work described here has been funded by the German Research Council (DFG).

[9] Private communication with R. d'Souza.

[10] Nevertheless we assume the front-end to be the most critical part of the compiler with respect to its overall performance. Todays availability of huge main memory allows to keep the abstract syntax tree in main memory entirely, which we expect to be the key issue for the performance of following compilation phases. Even with *complex* semantic analysis as it is the case for Eiffel-3.

[11] Private communication with R. d'Souza and see [KV92].

References

[Bla92] R. Blach. Ein LL(1)-Parser für Eiffel. In Hoffmann [Hof92].

[BS86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547, 1986.

[DP90] H. Dobler and K. Pirklbauer. Coco-2 – a new compiler compiler. Technical Report TR–90/1, Institut für Informatik – Johannes Kepler Universität, 1990.

[FJL+92] S. Finke, P. Jahn, O. Langmack, K.-P. Löhr, I. Piens, and T. Wolff. Distribution and inheritance in the HERON approach to heterogeneous computing. Report B-92-23, Freie Universität Berlin – Institut für Informatik, October 1992.

[Gan83] H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programming*, 3(3):223, 1983.

[GE90] J. Grosch and H. Emmelmann. *A Tool Box for Compiler Construction*, volume 477 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[GHL+92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2), February 1992.

[GMOS88] S. Grabau, A. Mast, R. Obermayer, and L. Schmitz. Cosy – manual (version 1.0). Technical Report 8803, Fakultät für Informatik, Universität der Bundeswehr München, 1988.

[Gra92] J. O. Graver. T-gen: a string-to-object translator generator. *Journal of Object-Oriented Programming*, 5(5):35, 1992.

[Gro90] J. Grosch. Object-oriented attribute grammars. In A. E. Harmanci and E. Gelenbe, editors, *Proceedings of the Fifth International Symposium on Computer and Information Science*, page 807, October 1990.

[Gro92a] J. Grosch. Ast – A generator for abstract syntax trees. Report 15, GMD Forschungsstelle an der Universität Karlsruhe, August 1992.

[Gro92b] J. Grosch. Rex - A scanner generator. Report 5, GMD Forschungsstelle an der Universität Karlsruhe, July 1992.

[GW92] P. Grape and K. Waldén. Automating the development of syntax tree generators for an evolving language. (Manuscript provided as “hand-out” at the conference). In Hoffmann [Hof92].

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF. Reference manual. CWI-Report CS-R–8926, CWI – Amsterdam, 1989.

[Hof92] H.-J. Hoffmann, editor. *Eiffel – Fachtagung des German Chapter of the ACM*, volume 35 of *Berichte des German Chapter of the ACM*. Teubner, 1992.

[Krä85] B. Krämer. Interactive graphical specification in a syntax-directed environment: The segras-lab experience. GRASPIN – Technical Paper 25/2, Gesellschaft für Mathematik und Datenverarbeitung, 1985.

[KV92] K. Koskimies and J. Vihavainen. Incremental parser construction with metaobjects. Report A-1992-5, Department of Computer Science – University of Tampere, November 1992.

[LMW89] P. Lipps, U. Moencke, and R. Wilhelm. Optran – a language/system for the specification of program transformations: System overview and experiences. In *Proceedings "2nd Workshop Compiler compilers and high speed compilation"*, volume 371 of *Lecture Notes in Computer Science*, page 52, 1989.

[Löh92] K.-P. Löhr. Concurrency annotations. In *Proceedings Conference on Object-Oriented Programming: Systems, Languages and Applications 1992*, 1992.

[Mey92a] B. Meyer. *Eiffel – The Language*. Object-Oriented Series. Prentice-Hall, 1st edition, 1992.

[Mey92b] B. Meyer. Eiffel-3 syntax revision. comp.lang.eiffel, July 1992.

[MN90] B. Meyer and J.-M. Nerson. *Eiffel: The Libraries*. Interactive Software Engineering Inc., 1990.

[Noo85] R. E. Noonan. An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3/4):225, 1985.

[SS91] M. Schweitzer and L. Strether. *The Eiffel/S Compiler and Runtime System*. SIG Computer GmbH, release 1.2, 1st edition, 1991.

[vB89] F. van Breughel. Testing compilers using grammars. Master's thesis, Eindhoven University of Technology, August 1989.

Appendix A – Concrete Syntax

A.1 – Lexical definitions not covered by appendix G of [Mey92a]:

- The regular expression[12] "`!{\t\s\n}*!`" shall be recognized as "`!!`" (see page 285 of [Mey92a]). I.e. the creation symbol may contain a sequence of arbitrary *white space* characters.
- The token "`Manifest_string`" can contain symbols matching the regular expression "`%\n{\t\s}*%`" (see page 390). I.e. it may contain a sequence of tabulation or space characters prefixed with a newline character enclosed in percent signs.

A.2 – Syntax definitions[13] not covered by appendix H of [Mey92a]:

```
Feature_declaration_list =
  { Feature_declaration [ ";" ] ... }

Entity_declaration_list =
  { Entity_declaration_group [ ";" ] ... }

Assertion = { Assertion_clause [ ";" ] ... }

Compound = { [ Instruction ] [ ";" ] ... }
```

(Note that these rules *approximate* what is explained informally on page 60, 114, 121 and 234 respectively. Some of these definitions cannot be expressed directly with a single syntax rule.)

A.3 – Updates to appendix H as published in [Mey92b]:

```
Address = "$" Address_mark
Address_mark = Identifier | "Current" | "Result"

Binary_expression = Expression Infix_operator Expression

Constant_attribute = Entity

Expression = Call | Operator_expression | Equality |
             Manifest_constant | Manifest_array | Old | Strip
```

[12] This syntax of regular expressions is defined in [Gro92b].

[13] Rules are denoted as in [Mey92a], only "=" is used instead of the baroque "$\triangleq$". Parentheses are used for grouping.

```
Unary_expression = Prefix_operator Expression

Unqualified_call = Entity [ Actuals ]
```

Deviations from appendices A.1-3 as indicated by source files that are accepted by the ISE or SIG compiler. The corresponding test data were provided by ISE or shipped with the SIG Release 1.2.

A.4 – ISE Prerelease

```
New_export_item = [ Clients ] Feature_set

Feature_declaration_list =
  { ( Feature_declaration [ ";" ] ) ... }

Entity_declaration_list =
  { ( Entity_declaration_group [ ";" ] ) ... }
```

A.5 – SIG Release 1.2

```
Entity_declaration_list =
  { ( Entity_declaration_group  [ "," ] [ ";" ] ) ... }

Parent_list = { Parent [ ";" ] ... }
```

Within the nonterminal rule "`Parenthesized_qualifier`" the parentheses – balanced tokens "`(`" and "`)`" – are optional for the qualifiers "`Character_constant`" and "`Manifest_string`".

Appendix B – Abstract syntax

```
Eiffel                  = Classes .

Classes                 = <
  Classes0              = .
  Classes1              = Class_declaration Classes .
> .

Class_declaration       = Indexing: Index_list
                          Class_header
                          Formal_generics
                          Obsolete: Manifest_constant
                          Inheritance: Parent_list
                          Creators
                          Features
                          Invariant: Condition
                          [comment: tStringRef]
                        .

Index_list              = <
  Index_list0           = .
  Index_list1           = Index_clause Index_list .
> .

Index_clause            = Index: Manifest_constant Index_terms: List .

Class_header            = <
  Class_header0         = .
  Class                 = Name: Manifest_constant .
  Expanded_class        = Name: Manifest_constant .
  Deferred_class        = Name: Manifest_constant .
> .

Features                = <
  Features0             = .
  Features1             = Feature_clause Features .
> .

Feature_clause          = Clients [comment: tStringRef] Feature_decls .

Clients                 = <
  Clients0              = .
  Clients1              = List .
> .
```

```
Feature_decls             = <
  Feature_decls0          = .
  Feature_decls1          = Feature_decl Feature_decls .
> .

Feature_decl              = New_feature_list Declaration_body .

Declaration_body          = Formal_args Type_mark: Type Const_or_routine .

Const_or_routine          = <
  Const_or_routine0       =  .
  M_const                 = Manifest_constant .
  Unique                  = [pos: tPosition]  .
  Routine                 = Obsolete: Manifest_constant
                            [comment: tStringRef]
                            Precondition: Condition
                            Local_decls: Formal_args
                            Routine_body
                            Postcondition: Condition
                            Rescue: Compound
                            [end_comment: tStringRef]
                          .
> .


Parent_list               = <
  Parent_list0            = .
  Parent_list1            = Parent Parent_list .
> .

Parent                    = Class_type Feature_adaptation .

Feature_adaptation        = <
  Feature_adaptation0     = .
  Feature_adaptation1     = Rename: Rename_list
                            New_export: New_export_list
                            Undefine: Feature_list
                            Redefine: Feature_list
                            Select: Feature_list .
> .

Rename_list               = <
  Rename_list0            = .
  Rename_list1            = Rename_pair Rename_list .
> .
```

```
Rename_pair             = Name1: Feature_name Name2: Feature_name .

Creators                = <
  Creators0             = .
  Creators1             = Creation_clause Creators .
> .

Creation_clause         = Clients [comment: tStringRef] Feature_list .

New_feature_list        = <
  New_feature_list0     = .
  New_feature_list1     = Feature_name New_feature_list .
> .

Feature_name            = <
  Frozen                = Feature_name .
  Ident_name            = Id .
  Op                    = <
    Prefix              = Op_name .
    Infix               = Op_name .
  > .
> .

Op_name                 = [pos: tPosition] <
  Free_op               = [ident: tIdent] .
  Oper                  = [op] .
> .

New_export_list         = <
  New_export_list0      = .
  New_export_list1      = New_export_item New_export_list .
> .

New_export_item         = Clients Feature_set: Feature_list .

Feature_list            = <
  All                   = .
  Feature_list0         = .
  Feature_list1         = Feature_name Feature_list .
> .
```

```
Formal_args               = <
  Formal_args0            = .
  Formal_args1            = Entity_decl_group Formal_args.
> .

Routine_body              = <
  Routine_body0           = .
  Effective               = <
    Internal              = <
      Do_body             =  Compound .
      Once_body           =  Compound .
    > .
    External              = Lang1: String Lang2: Manifest_constant .
  > .
  Deferred                = [pos: tPosition] .
> .

Entity_decl_group         = Id_list: List Type_mark: Type .

Formal_generics           = <
  Formal_generics0        = .
  Formal_generics1        = Formal_generic Formal_generics .
> .

Formal_generic            = Id: Manifest_constant  Constraint .

Constraint                = <
  Constraint0             = .
  Constraint1             = Class_type .
> .

Compound                  = <
  Compound0               = .
  Compound1               = Instruction Compound .
> .

Then_part_list            = <
  Then_part_list0         = .
  Then_part_list1         = Then_part Then_part_list .
> .

Then_part                 = Guard: Expression Compound .
```

```
When_part_list          = <
  When_part_list0       = .
  When_part_list1       = When_part When_part_list .
> .

When_part               = Choices Compound .

Choices                 = <
  Choices0              = .
  Choices1              = Choice Choices .
> .

Choice                  = <
  Interval              = From: Manifest_constant To: Manifest_constant .
  Val                   = Manifest_constant .
> .

Instruction             = <
  Instruction0          = .
  Creation              = Type Entity Unqual_call .
  Call_instruct         = Call .
  Assign                = Addr: Entity Expression .
  Rev_assign            = Addr: Entity Expression .
  Conditional           = Then_part: Then_part_list
                          Else_part: Compound .
  Multi_branch          = Guard: Expression
                          When_part_list
                          Else_part: Compound .
  Loop                  = Initialization: Compound
                          Invariant: Condition
                          Variant: Assertion_clause
                          Loop_body .
  Check                 = Assertion .
  Debug                 = Debug_keys: List Compound .
  Retry                 = [pos: tPosition] .
> .

Loop_body               = <
  Loop_body0            = .
  Loop_body1            = Exit: Expression Compound .
> .
```

```
Type                     = <
  Type0                  = .
  Class_type             = Id Actual_generics .
  Class_type_expanded    = Id Actual_generics .
  Bit_type               = Manifest_constant .
  Simple_type            = Manifest_constant .
  Anchored               = Entity .
> .

Actual_generics          = <
  Actual_generics0       = .
  Actual_generics1       = Type_list .
> .

Condition                = <
  Condition0             = .
  Condition1             = [extension] Assertion .
> .

Assertion                = <
  Assertion0             = .
  Assertion1             = Assertion_clause Assertion .
> .

Assertion_clause         = <
  Assertion_clause0      = .
  Assertion_clause1      = Tag: Manifest_constant Expression .
> .

Type_list                = <
  Type_list0             = .
  Type_list1             = Type Type_list .
> .

Call_chain               = <
  Call_chain0            = .
  Call_chain1            = Unqual_call Call_chain .
> .

Unqual_call              = <
  Unqual_call0           = .
  Unqual_call1           = Entity Actuals .
> .
```

```
Actuals                  = <
  Actuals0               = .
  Actuals1               = Actual_list .
> .

Actual_list              = <
  Actual_list0           = .
  Actual_list1           = Actual Actual_list .
> .

Actual                   = <
  Addr                   = Entity .
  Expression             = <
    Expression0          = .
    Comment              = [comment: tStringRef] .
    Simple_expr          = Manifest_constant .
    Call                 = Qual: Expression Call_chain .
    Bin_expr             = Lop: Expression Op Rop: Expression .
    Un_expr              = Op Expression .
    Parenth              = Expression .
    Manifest_array       = <
      Manifest_array0    = .
      Manifest_array1    = Expression Manifest_array .
    > .
    Old                  = Expression .
    Strip                = List .
  > .
> .

List                     = <
  NoList                 = .
  List                   = Manifest_constant List .
> .
```

```
Manifest_constant        = <
  Manifest_constant0     = .
  Ch                     = [ch] [pos: tPosition] .
  True                   = [pos: tPosition] .
  False                  = [pos: tPosition] .
  Int                    = Sign: Oper [value: long]   [pos: tPosition] .
  Real                   = Sign: Oper [value: double] [pos: tPosition] .
  _CHARACTER             = [pos: tPosition] .
  _INTEGER               = [pos: tPosition] .
  _REAL                  = [pos: tPosition] .
  _DOUBLE                = [pos: tPosition] .
  _BOOLEAN               = [pos: tPosition] .
  _STRING                = [pos: tPosition] .
  _BIT                   = [pos: tPosition] .
  _NONE                  = [pos: tPosition] .
  Bitseq                 = [literal: tStringRef] [pos: tPosition] .
  String                 = [string: tStringRef] [pos: tPosition] .
  String0                = .
  Entity                 = <
    Entity0              = .
    Current              = [pos: tPosition] .
    Result               = [pos: tPosition] .
    Id                   = [ident: tIdent] [pos: tPosition] .
  > .
> .
```

Appendix C – Usage

The HERON front-end "ep" is available in the public domain. It runs on Sun-Unix (Version 4.1, Architectures Sun-3 and Sun-4) or Ultrix-32 (Version 3.1). It can be obtained via anonymous file transfer from:

```
ftp.fu-berlin.de:/pub/heron/ep.tar.Z
```

Part of this archive file is the following C program. It illustrates how the abstract syntax tree can be used for the generation of an identifier list:

```
#include "Tree.h"
#include <stdio.h>

void find_Id(tree_node)
tTree tree_node;
{
  if(tree_node->Kind == kId) {
  WriteIdent(stdout, tree_node->Id.ident);
  fprintf(stdout,"\n");
  }
}

int main (argc, argv)
int argc;
char **argv;
{
  tTree tree;
  FILE *fp;

  InitStringMemory();
  tree = ReadTree(fp);
  TraverseTreeTD(tree, find_Id);
}
```

Note the use of identifiers of nonterminals and attributes – as defined by the abstract syntax (see page 20) – in operation `find_Id`. See the following appendix for other operations to deal with the tree, nodes or attributes. The include file `Tree.h` defines the generated C data structures for the abstract syntax tree.

Appendix D – Syntax Tree Access

The following table gives a selection of operations for tree, node and attribute access as generated by the tree generator of the Karlsruhe toolbox. It is a summary of the respective section of [Gro92a]. Header file identifiers refer to files of the public archive.

`Tree.h`	
`Tree_IsType`	test a node for a certain type
`WriteTreeNode`	write ASCII node
`WriteTree`	write ASCII tree
`ReadTree`	read ASCII tree
`PutTree`	read binary tree
`GetTree`	write binary tree
`TraverseTreeTD`	top down tree traversal (reverse depth first)
`TraverseTreeBU`	bottom up tree traversal (depth first search)
`ReverseTree`	reverse tree
`CopyTree`	tree copy
`CheckTree`	check tree syntax
`QueryTree`	browse tree
`IsEqualTree`	test tree equality

`StringMem.h`	
`WriteString`	write attributes of type `tStringRef`

`Idents.h`	
`WriteIdent`	write attributes of type `tIdent`

`Positions.h`	
`WritePosition`	write attributes of type `tPosition`
`Compare`	compare two attributes of type `tPosition`