# Translating a subset of SETL/E into SETL2

W. Hasselbring

Computer Science / Software Engineering
University of Essen
F. R. Germany
willi@informatik.uni-essen.de

January 28, 1991

## Abstract

A translator for a subset of SETL/E into SETL2 built with the compiler construction system Eli will be presented. The main objective of this work was to obtain the basic specifications for a compiler that transforms SETL/E into ANSI-C. The latter transformation is in progress in parallel to our work. Additional benefits are the ability to execute and to experiment with a subset of SETL/E and to obtain a comparison with some features of SETL2. This report is assumed to be the documentation for a development step.

# Contents

# List of Figures

## 1 Introduction

The set theoretic language SETL/E is a successor of SETL [SDDS86, DF89]. For a full account on set theoretic languages and programming we refer to these books. SETL/E is at present under development at the University of Essen. The kernel of the language was at first presented in [DGH90b] and the system in [DGH90a]. Considerations on persistence and concurrency are under way but not treated in this work.

Several changes to the language definition were made since its first presentation. In this work we refer to the version given in [DFGH90]. Originally it was intended to build a pretty printer for SETL/E. Performing this would require approximately the same effort as a translation into SETL2. Because of the additional benefits we decided to perform the translation into SETL2. The SETL2 programming language was evolved from SETL too, and developed at the New York University [Sny90a].

The compiler construction system Eli is the central tool for implementing SETL/E. Eli integrates off-the-shelf tools and libraries with specialized language processors to provide a system for generating complete compilers quickly and reliably [GHK+90].

In the next section the compiler specifications are presented. The differences with SETL2 are occasionally given in section 2.3 where appropriate. There's no extra section for a comparison, because this was not a primary goal of this work and there were not too much differences discovered. Section 3 gives a short guide to use the produced translator.

The presented compiler is called a translator because the language level is not changed. We will use the word 'compiler' synonymous to 'transformer' or 'translator' because Eli is a 'compiler construction system'. You could also use 'language processor'.

## 2 The Specifications

Eli contains a collection of Tools for performing the respective compiler construction tasks as producing a scanner, parser or attribute evaluator. For a full account on compiler construction we refer to [ASU86].

Figure 1 shows the contents of the specifications file needed for our translation. This file is used as Eli's input. It contains the names of all the files that are necessary to derive an executable compiler.

The several specifications are discussed in the following sections. In section 2.1 the lexical structure is presented. Section 2.2 presents the concrete grammar specification and its relationship to the abstract grammar.

The LIDO specifications that are used to derive the attribute evaluator are presented in section 2.3. The attribute evaluator drives the semantic analysis phase of the compiler frontend and the code production.

The remaining specifications contain ANSI-C code. Some people would not call them as *specifications*. However, this part of the compiler is presented in section 2.4.

The specifications `sec.gla`, `sec.con`, `sec.sym`, `sec.rel`, `sec.abs`, `rules.lido`, `Check.c`, `Cond.lido`, `PutFile.c` and `Fehler.c` will be reused for the transformation into ANSI-C.

Section 2.5 will explain the way to derive an executable compiler or the source for this compiler with Eli.

```
     /* The specifications: */
sec.gla                         /* Lexical structure */
sec.con                         /* Concrete grammar */
sec.sym                         /* Symbol equivalences */
sec.rel                         /* Concrete/abstract relationship */
sec.abs                         /* Abstract grammar */

rules.lido                      /* All rules without attribution */
Cond.lido                       /* Static conditions */
Code.lido                       /* Code production */
Indent.lido                     /* Indentation management */
Limits.lido                     /* Limits for the translation */

ses2.head                       /* Specification headers */
ses2.init                       /* Initial statements */
ses2.finl                       /* Final statements */

Fehler.c                        /* Error handling */
Check.c                         /* Check conditions */
PutFile.c                       /* ADT for code production */
```

Figure 1: The specifications file.

## 2.1 Lexical analysis

GLA is in Eli a tool that generates lexical analyzers [Gra89]. Usually a lexical analyzer will need to deal with input consisting of some combination of literal symbols, non-literal symbols (such as identifiers or integers) and comments. Users of Eli need not deal with literal symbols (such as '`begin`' or '`*`') because they are automatically extracted from the user's grammars. A complete GLA specification consists of three parts: The non-literal description part, options and the encoding part. Eli users only give the non-literal description part; the other parts are automatically produced from the parsing grammar.

The specification for the behavior of the lexical analyzer for SETL/E is given in Figure 2. This is the content of `sec.gla`. With the exception of floating point constants we referenced *canned descriptions*. For floating point constants it was necessary to provide a regular expression and a processor that saves the corresponding string. For details see [Gra89].

## 2.2 Syntax analysis

### 2.2.1 Concrete grammar

The concrete grammar describes the context free syntax of the language to be compiled. The parser for the compiler is generated from the concrete grammar.

The terminals referenced in the concrete grammar are defined in the `sec.gla` specification. See Figure 2 in section 2.1.

The purpose of a concrete syntax is to describe the structure of the input program as the programmer writes it down. This is the structure that is recognized by the parser and built into the tree described by the attribute grammar.

```
        { Lexical structure: }

id:     C_IDENTIFIER
int:    PASCAL_INTEGER
float:  $([0-9]+\.[0-9]+|[0-9]+\.[0-9]+(e|E)(\+|\-)?[0-9]+) [mkstr]
str:    C_STRING_LIT
        ADA_COMMENT
        PASCAL_COMMENT
```

Figure 2: The lexical structure specification.

The problem with this is that the structure of the program written by the programmer is governed by rules of operator precedence that are not involved in the process of gathering and distributing information over the tree. Operator precedence and bracketing rules play an important part in building the tree, but no role whatsoever once the tree is available. Therefore if operator precedence and bracketing rules are to be provided then they should be described by a concrete syntax that is distinct from the attribute grammar.

Eli uses parser generators that accept LALR(1) grammars. An introduction to this class of *lookahead*-LR context free grammars is given in [ASU86, section 4.7]. The LALR(1) grammar for SETL/E is given in appendix A.

All the names of the nonterminals begin with the character '`x`'. This is not necessary, but seems to be useful for the attribution rules (section 2.3). The nonterminals that occur only in the concrete grammar and not in the abstract grammar (thus not in the attribution) begin with '`xc`'.

Exceptions and operator declarations are at present not handled. See also section 2.3.3.

### 2.2.2 Concrete/abstract grammar tool

The productions in the concrete grammar are mapped into the attribute grammar using the concrete/abstract grammar tool (CAGT) [Gro89a]. CAGT is a tool that is used to specify the relationship between a concrete grammar and an abstract grammar. The user provides a concrete grammar to CAGT, and interactively transforms it into the desired abstract grammar. It produces the files `sec.rel`, `sec.abs`, and `sec.sym`, which describe the mapping of the concrete grammar into the attribute grammar.

CAGT records the relationship between the original concrete grammar and the transformed abstract grammar in the file `sec.rel`. The abstract grammar itself is put in `sec.abs` and the symbol equivalences in `sec.sym`.

### 2.2.3 Abstract grammar

An abstract grammar describes all of the possible forms taken by each of the syntactic classes of a language, and therefore determines the structure of the tree used to represent programs internally. It does not, however, define the set of character strings that are well-formed program texts or specify their phrase structures.

For example, the abstract grammar for SETL/E contains a rule describing

*<Expression> <Operator> <Expression>*

as one form of an expression, but it would not determine whether `a + b * c` is a well-formed expression or, if it is, whether `b` is an operand of `+` or of `*`. Such questions would be answered by the concrete grammar.

In specifying a language processor, the abstract grammar describes the possible shapes of the trees which will be used to represent programs internally. Each rule of the abstract grammar corresponds to one particular kind of tree node. The concrete grammar, on the other hand, describes the possible character strings that make up well-formed programs. These character strings are recognized by the parser, which is controlled by the concrete grammar and invokes tree-building actions to construct the program's internal representation.

How does the parser know when to invoke a tree-building action, and which action to invoke? Since each rule of the abstract grammar corresponds to a particular kind of tree node, the action to create that node can be attached to the abstract grammar rule as a decoration. Unfortunately, that action must be attached to some concrete grammar rule in order to be available to control the parser. But CAGT understands and records the relationship between the abstract grammar and the concrete grammar.

Distinct symbols of the concrete grammar are represented by a single symbol in the abstract grammar. Thus certain rules of the concrete grammar have no identical rules in the abstract grammar. These symbol equivalences are recorded in `sec.sym` and given in appendix B.

The content of `sec.abs` (the abstract grammar) is not given here to save space. You have only to replace in the concrete grammar all occurrences of nonterminals starting with '`xc`' on the right sides in `sec.sym` with their respective left hand sides, and to remove the identical productions. In addition the literals in the productions of figure 3 were removed.

```
xParamList ::= '(' xcParams ')' .
xcCaseList ::= 'when' xExprList '=>' .
xcPrimary ::= '(' xExpr ')' .
```

Figure 3: Concrete grammar productions where the literals were removed.

Thus these productions are not present in the abstract grammar. Especially, parentheses in expressions are of no interest in an abstract grammar. In principle it would be possible to remove more literals from the abstract grammar. But the resulting grammar would not be very readable.

## 2.3 Attribute evaluation

An attribute grammar (AG) specifies context dependent computations of attribute values associated to nodes of a tree. If applied to the semantic analysis phase of a compiler the tree is the structure tree of the program, which usually is determined by the parser as an abstraction of the derivation tree. Hence the AG augments a context-free grammar (CFG) specifying the structure of that tree. Attributes are associated to symbols of the abstract grammar. An attribute value of a tree node for a symbol describes a context dependent property of that symbol instance in its tree context (like the type of an expression). Since the computation of attribute values is determined by the context of the symbol, attribution rules are associated to the productions of the CFG. Here this CFG is the above-mentioned abstract grammar.

AGs are well suited for formal and declarative descriptions of any kind of systematic information flow through recursive tree structures. They have been proven to be a suitable means for specification of the semantic analysis phase of compilers. Such compiler modules are systematically implemented by attribute evaluators which compute language properties as attributes associated to nodes of the structure tree for the program. Furthermore the AG specifies context dependent conditions which must hold if the program is correct according to the static semantics (e.g. type and scope rules). They are specified by functions over attribute values associated to productions, too.

LIGA is a language independent generator for attribute evaluators in Eli [Kas90b]. An attribute evaluator is specified by an AG written in LIGA's input language LIDO [Kas90a]. The specification comprises a context-free grammar augmented by typed attributes and specifications of context dependent attribute computations. Exchangeable backends allow to implement the evaluator in different implementation languages and to vary the implementation techniques. Its basic concepts include specific notations and structures which support common attribution schemes, refinement of the attribute grammar, and the systematic use of abstract data types in the attribution.

The AG class accepted by LIGA is that of ordered attribute grammars [Kas80]. It belongs to the classes which on the one hand frees the AG designer from planning the evaluation order as far as possible, and on the other hand allows to compute the control structure of the attribute evaluator in polynomial time. Furthermore methods for attribute storage reduction are applied at generation time.

LIDO has a functional interface to any language suitable for the specification of attribute computation. It restricts the attribution to the functional dependencies only. The functions themselves are supplied separately, written in the implementation language of the generated attribute evaluator. In the present version of LIGA this language is C. The natural and only means to influence evaluation order are attribute dependencies. Shorthand denotations are available for description of common attribution structures. LIDO is completely declarative and the attributes are typed. Their context dependent computation is specified by expressions constructed as nested function calls. The implementations of both the types and functions used in the AG is opaque to the LIGA system. They are supplied separately to be integrated into the generated attribute evaluator (see also section 2.4.1).

In our application the attribute evaluator drives the static semantic analysis phase of the compiler frontend and the code production.

LIDO permits splitting the specification of attribute computation for the productions over several files. To obtain a reusable basis we put all the rules without attribution in `rules.lido`. This represents the abstract grammar discussed in section 2.2.3 in LIDO syntax. However, `sec.abs` is necessary because the integers contained in the relationship file `sec.rel` refer to the respective positions in `sec.abs`. The content of `rules.lido` is not given here to save space. The remaining LIDO specifications are presented in the following sections. They only apply to subsets of the abstract grammar. You could remove these specifications from `ses2.specs` (figure 1) to obtain a pure syntax checker for SETL/E. But at least `rules.lido` is necessary.

### 2.3.1 Static semantics

In languages as Pascal the static semantics analysis could result in error messages like the following:

"Error: Variable not declared!"

In SETL/E such messages would not be appropriate, because it is not necessary to declare objects. We use the term 'object' not as e.g. Smalltalk does. We don't want to give the $n$-th definition of object-oriented or object-based, but a definition[1] for our terminology:

> **Definition:** Each variable or constant of type **integer**, **real**, **string**, **boolean**, **atom**, **tuple**, **set**, or **proctype** is meant to be an **object** in SETL/E. These objects have **first-class** rights. First-class means to be expressible without giving a name. It implies having the right to be anonymous, being storable in variables and in data structures, being returnable from or passable to a procedure.
>
> This has the consequence that exceptions and user-defined operators have no first class rights, which is justifiable because of their restricted way of use. E.g. binary operators are used syntactically where other objects cannot be used and vice versa.

However, warnings like the following would be useful:

"Warning: Object used before being initialized!"

This kind of static semantics is not considered in this work. It would be necessary to construct e.g. a program dependence graph to do this analysis.

The following static semantics are checked:

- Equivalence of header and trailer names in programs, procedures and labeled control statements is checked.
- It is assured that constants are initialized.
- **Return** statements are only allowed inside procedures or lambdas.
- Recursive lambda calls with **self** are only allowed inside lambdas.
- **Quit** and **continue** statements are only allowed inside loops.

For details on the specification of these constraints see appendix C, which is the content of `Cond.lido`.

### 2.3.2 Code production

The set of shorthand notations in LIDO for abbreviation of systematic attribute value propagation is extended by a notation for chaining propagation. Chain constructs are used to propagate attribute values left to right depth first through the structure tree.

Our chain path of chain attribute `Code` begins in the root production in the file `Code.lido` (figure 4). It is necessary to have such a production to initialize the chain. For details see [Kas90a]. We use the initialization to open the output file. The functions `OpenFile`, `PutStr` etc. are discussed in section 2.4.1.

A chain leads through all subtrees reaching any access of that chain and goes up again to the chain start context. Any nonterminal on the chain path has a pair of implicit attributes for that chain (one inherited, and one synthesized). If these attributes of some nonterminals are not defined in the attribution rules, default settings are generated by the system.

[1]This definition is only valid for the presented work. It is not part of the language definition.

```
CHAIN Code: VOID;

RULE rInitChain: xInitChain ::= xProgDefn
STATIC
  CHAINSTART Code;
  xProgDefn.Code := OpenFile ();
END;
```

Figure 4: The chain start for the code production.

Attributes of the predefined type VOID are used only to state attribute dependencies. Their value is not relevant. They do not occur in the attribute evaluator.

These assumptions are well suited for our code production. For reasons of space, and because we will not reuse this attribution we don't give the whole content of `Code.lido` here. As an example see the code production for binary operations in figure 5. This attribution enforces that all binary operations are set in parentheses in the produced SETL2 program to assure the right precedences and associatives.

```
NONTERM xBinOp: opsym: STRING SYNT;

RULE rExprBinop: xExpr ::= xExpr xBinOp xExpr
STATIC
    xExpr[2].Code := PutStr (SL, '(');
    xExpr[3].Code := DEP(PutStr (SL, xBinOp.opsym), xExpr[2].Code);
    xExpr[1].Code := PutStr (SL, ')');
END;

RULE rBinop1: xBinOp ::= 'or'
STATIC
    xBinOp.opsym := 'OR';
END;
```

Figure 5: Code production for binary expressions.

DEP determines the evaluation order. It makes the first parameter dependent on the second. The attribute type STRING was previously defined in `rules.lido`.

The differences between SETL/E and SETL2 concerning the code production are of syntactical nature and not worth a great discussion. As an example see the code production for the `until` statement in figure 6. It is necessary to exchange the ordering of the expression and the statements and to change the keywords. As an example for a SETL/E-`until` statement see figure 7 on page 8. In figure 8 the produced SETL2-`until` statement is given. This simple program is not very sophisticated.

The Indentation is controlled with the `Indent` attribute on the nonterminals `xProgBody` and `xStmts`. This attribute is passed to the function `PutStr` (see section 2.4). The management is done in `Indent.lido`. As a part see figure 9 on page 9. OFFSET is a defined integer constant that controls the additional indentation for inner blocks.

```
RULE rUntilStmt: xLoops ::= 'do' xStmts 'until' xExpr
STATIC
    xExpr.Code := PutStr (INCLUDING xStmts.Indent, 'UNTIL');
    xStmts.Code := DEP (PutStr (INCLUDING xStmts.Indent, 'LOOP'),
                        xExpr.Code);
END;
```

Figure 6: Code production for the until statement.

```
program example;
  x := 0;
  do
    x +:= 1;
    until x > 10 or x > 11
  end do;
end example;
```

Figure 7: A simple SETL/E program.

```
PROGRAM example ;
   x := 0 ;
  UNTIL ( ( x > 10 ) OR ( x > 11 ) )
  LOOP
     x + := 1 ;
  END LOOP ;
END example ;
```

Figure 8: The produced SETL2 program.

```
NONTERM xProgBody, xStmts: Indent: INT INH;   % A priori Indentation

RULE rProgDefn:
   xProgDefn ::= 'program' id ';'
                     xProgBody
                 'end' id ';'
STATIC
  xProgBody.Indent := OFFSET;
END;

RULE rProcDefn:
   xProcDefn ::= 'procedure' id xParamList ';'
                     xProgBody
                 'end' id ';'
STATIC
   xProgBody.Indent := ADD (INCLUDING xProgBody.Indent, OFFSET);
END;
```

Figure 9: A part of the indentation management.

### 2.3.3 Limits for the translation

In principle it would be possible to translate all SETL/E constructs into SETL2 because of their ability to simulate the universal Turing-Machine. But some features would demand a not justifiable effort for our purposes. These features are:

- Exceptions are not handled because SETL2 has no exceptions.
- Operators are not handled because SETL2 has no user-defined operators. An extension supports operator overloading [Sny90b], but this is not the same.
- SETL2 does not support recursive `lambda` calls. In SETL/E this is done with `self`.
- Control statements with labels are handled with an appropriate warning message, but `quit` and `continue` statements on such labels are refused.
- The scope of objects is by default local to the program body where these objects are used. In SETL/E it is possible to make objects visible in inner blocks via `visible`-declarations. This is done in SETL2 with `var`. It is possible to hide `visible`-declarations from outer blocks in inner blocks with new `visible`-declarations. This works in both languages.

  Additionally it is possible in SETL/E to hide such objects only for the corresponding program body with a `hidden`-declaration. Because this is new with respect to [DGH90b] see the example in figure 10.

  Such declarations are not available in SETL2.
- In SETL2 the only exception to the rule that bound variables are local to iterators is in the `exists` expression [Sny90a, page 7]. In SETL/E this exception was not made.

```
program prog;
    visible x := 5;
    p();

    procedure p;
        hidden x := 1;
        q();

        procedure q;
           -- now: x = 5
        end q;
    end p;
end prog;
```

Figure 10: An example for the **hidden** declaration. If `x` would be declared **visible** in `p`, then the value of `x` in `q` would be `1`.

However, the **exists** and **notexists** expressions are translated with a warning message that the visibility rules are not preserved.

- In SETL2 the **exists** expression sets its bound variables on exit, to the value found if successful or **om** if unsuccessful. This is sometimes useful in constructs as

  "`while exists x in { ... } | condition(x) do ...`"

A found set-element is directly available via `x`, but this bound variable is not local to the loop, what is the case in **for** loops.

For these reasons we introduced the `whilefound` loop. Because this is new with respect to [DGH90b] see the example in figure 11. The loop body is executed, if an **exists** expression with the same iterator would yield **true**. The bound variables are local to the `whilefound` loop as they are in **for** loops. The iterator is reevaluated for every iteration unlike in **for** loops.

```
program prog;
    x := 5;
    S := {1, 2, 3};
    whilefound x in S | x < 4  do
        eat (x);        -- now: x = 1 or x = 2 or x = 3
        S less := x ;
    end whilefound;
    -- now: x = 5 and S = { }
end prog;
```

Figure 11: An example for the `whilefound` loop.

The `whilefound` loop is translated into an appropriate "`while exists ...`" loop with a warning message that the visibility rules are not preserved.

- In SETL/E there are the selection operators `arb` for deterministic selection and `select` for non-deterministic selection. In SETL2 there is only the deterministic selection.
- There is no multivalued map iterator in SETL2. For a description of this iterator see e.g. [SDDS86, page 129].
- The predefined unary operator `type` provides in SETL/E the type of its operand as a predefined atomic constant. Whereas in SETL2 the built-in procedure `type` returns a character string representation of the type of its argument. However, these constructs are translated with appropriate warning messages.

  But take into account that e.g. "`type(type(x))`" will not produce what would be expected! This problem is ignored for the moment.

  In SETL2, there are additionally `is_`*type*`(v)` built-in procedures for all types and for maps. In SETL/E, there are the unary set operators `is_map` (provides `true` for sets that are multi- or single-valued maps) and `is_smap` (provides `true` only for for sets that are single-valued maps). Thus `is_map` is translated with an appropriate warning message and `is_smap` is refused.

Messages for these restrictions are generated by directly calling the error handling function `fehler` (see section 2.4.2) in the attribution rules in `Limits.lido`.

However, we cannot give the guaranty that all the programs that our compiler translates are accepted by the SETL2 compiler.

## 2.4 Abstract data types and ANSI-C code

### 2.4.1 The abstract data type `PutFile`

In LIDO attribute domains are considered as abstract data types (ADTs). The AG specification and the generation of evaluators is independent of the implementation of the ADTs used in the AG. Implementation considerations are completely opaque to the AG specification.

ADTs may define some state transition model. In that case the values of the ADT represent states. Its operations are state transition functions and access functions which yield results depending on the ADT state. A simple example with only state transition functions is an output ADT.

Any restrictions on the state transition protocol of the ADT can be specified by attribute dependencies. These restrictions will be obeyed automatically by the generation of the attribute evaluator.

In our application the output ADT is a data sink. No information is accessed via the attributes representing its states. They serve only one purpose, to guarantee the specified sequence of operations. Such pure state attributes can be eliminated completely from the evaluator. In LIDO such attributes are specified to have the predefined type `VOID` (see figure 4 on page 7). It indicates that no storage has to be allocated. The function `DEP` yields its first argument as result and discards the second. Hence, it establishes only a dependency on the second argument. Such attribution rules are translated simply to the function calls at the appropriate place in the attribute evaluator.

The abstract data type `PutFile` implements the following functions for producing an output file:

`extern void OpenFile ();` Opens the output file.

`extern void CloseFile ();` Closes the output file.

`extern void PutStr (const int, const char *);` Outputs the string that is given as the second parameter. The first parameter controls the indentation. If it is the defined integer constant `SL`, the string is put on the actual line. Otherwise it is put on the next line with the given indentation (`SL` is defined as a negative integer and means *same line*).

`extern void PutId (const int, const int);` Outputs the string that represents the identifier given as the second parameter. GLA stores these strings in a global array and the attribute evaluator only handles the indices to this array.

The first parameter controls the indentation as above.

`extern void PutInt (const int);` Outputs the string that represents the integer constant given as the second parameter. The string is put on the actual line.

`extern void PutFloat (const int);` Outputs the string that represents the float constant given as the second parameter. This string is stored in the same array as identifiers. The string is put on the actual line.

`extern void PutStrVal (const int);` Outputs the string that represents the string constant given as the second parameter. This string is also stored in the same array as identifiers. The string is put on the actual line.

This specification is given in `PutFile.h` and the implementation in `PutFile.c`.
Eli also provides an ADT for text output. But this output module seemed not to be appropriate for our purposes.

### 2.4.2 Error handling

The error handling function `fehler` emits the messages to the user on `stderr`. It distinguishes six message types:

`WARN` Warning.

`ABORT` Unrepairable error.

`RECOVER` Repairable error.

`COMPILER` Compiler error.

`DATEI` File error as "*Incorrect extension of input file*".

`SYSTEM` File error as "*Permission denied*".

The type definition for this type and the function prototype are given in figure 12. The function definition is given in `Fehler.c`.
The functions `Assert` and `Equal` call the error handling function if appropriate with error type `RECOVER`. They check conditions in the attribution and are defined in `Check.c`:

`void Assert (const int cond, const char *report)` calls `fehler` with the `report`, if the condition is not satisfied.

`void Equal (const int a, const int b, const char *report)` calls `fehler` with the `report`, if the first two parameters are not equal.

```
 /* Type for error handling: */
typedef enum {WARN=1, ABORT, RECOVER, COMPILER, DATEI, SYSTEM} ERRORTYPE;

extern void fehler(
        const char functionname[] ,  /* Name of calling function */
        const ERRORTYPE type ,       /* Error type */
        const char message[]         /* Error message */
        );
```

Figure 12: The error handling (`Fehler.h`).

### 2.4.3 Miscellaneous

The header files for the above-mentioned C functions are given in figure 13. They supply the derived compiler with the necessary constant and type definitions and the function prototypes.

```
#include "Fehler.h"
#include "Check.h"
#include "PutFile.h"
```

Figure 13: The header files in `ses2.head`.

In `ses2.init` there are C-statements that are to be executed before the scanner starts lexical analysis. At present this is used to print out the actual compiler version.

In `ses2.finl` there are C-statements that are to be executed after attribute evaluation. At present this is used to print out the error counts.

## 2.5 Derivation

Eli is a particular instantiation of a system for managing software tools called Odin [CO90]. It operates within a universe of *objects*, each of which is a UNIX[2] file or directory. A user manipulates the objects in Eli's universe by making *requests* to Eli during a session or in batch mode.

To derive an executable compiler the following request would be appropriate:

```
ses2.specs +fold:  exe > ses2
```

For details see [Gro89b]. The parameter `fold`[3] causes Eli to manufacture a compiler without case distinctions, what is necessary in SETL/E for identifiers and keywords, but not in string constants.

To put the source for this compiler e.g. in the directory `COMPILER` the following request would be appropriate:

```
ses2.specs +fold:  source > COMPILER
```

[2]UNIX is a trademark of AT&T.

[3]At present this parameter is ignored by Eli, but we hope that this problem will be eliminated by the Compiler Tools Group in the near future. Possibly this will require changes in the GLA specification.

## 3 User's guide

The SETL/E input file must have the extension '`.se`'. To translate e.g. `input.se` enter

```
ses2 input.se
```

To execute the produced SETL2 program in `input.stl` see [Sny90a, section 3].

## 4 Conclusions

The Eli specifications for a compiler that translates a subset of SETL/E into SETL2 were presented. Essential parts of these specification will be reused for a transformation of SETL/E into ANSI-C.

Except for the limits for the translation the differences found between SETL/E and SETL2 are merely of syntactical nature and not worth a great discussion. But take into account that we only considered subsets of both languages in this work.

## Acknowledgements

Thanks to K.J. Prott for the help with Eli.

The extensive comments on the first version of the language definition by Fritz Henglein are appreciated and the discussion of some features of SETL/E with him was very helpful.

## A The concrete grammar

```
/************************************************************
Concrete grammar for SETL/E
************************************************************/
xInitChain ::=   xProgDefn .
/******************************************
Program and procedure definition:
******************************************/
xProgDefn ::=   'program' id ';' xProgBody 'end' id ';' .
xProgBody ::=   xDecls xStmts xProcDefns .
xProcDefns ::=   xProcDefns xProcDefn .
xProcDefns ::=   .
xProcDefn ::=   'procedure' id xParamList ';' xProgBody 'end' id ';' .
xParamList ::=   .
xParamList ::=   '(' xcParams ')' .
xParamMode ::=   .
xParamMode ::=   'rd' .
xParamMode ::=   'rw' .
xParamMode ::=   'wr' .
xcParams ::=   xcParams ',' xParamMode id .
xcParams ::=   xParamMode id .
xProcDefn ::=   'exception' .  % to be completed
xProcDefn ::=   'operator' .   % to be completed
/******************************************
Declarations:
******************************************/
xDecls ::=   xDecls xDecl .
xDecls ::=   .
xDecl ::=   xDeclKey xcVars ';' .
xcVars ::=   xcVars ',' xSingleVar .
xcVars ::=   xSingleVar .
xSingleVar ::=   id ':=' xExpr .
xSingleVar ::=   id .
xDeclKey ::=   'visible' .
xDeclKey ::=   'hidden' .
xDeclKey ::=   'visible' 'constant' .
xDeclKey ::=   'hidden' 'constant' .
xDeclKey ::=   'constant' .
/******************************************
Statements:
******************************************/
xStmts ::=   xStmts xStmt ';' .
xStmts ::=   xStmt ';' .
%
% Simple Statements:
xStmt ::=   'pass' .
xStmt ::=   'stop' .
xStmt ::=   'return' xExpr .
xStmt ::=   'return' .
%
% Assignments:
xStmt ::=   xLValue ':=' xExpr .
xStmt ::=   xLValue xBinOp ':=' xExpr .
xStmt ::=   xLValue xFrom xcSimpleLV .
xFrom ::=   'from' .
xFrom ::=   'frome' .
xFrom ::=   'fromb' .
%
% Function calls:
xStmt ::=   id '(' xExprList ')' .
xStmt ::=   id '(' ')' .
%
% Recursive lambda calls:
xStmt ::=   'self' '(' xExprList ')' .
xStmt ::=   'self' '(' ')' .
%
% Conditional statements:
xStmt ::=   'if' xExpr 'then' xStmts xElIfStmts 'end' 'if' .
xStmt ::=   'if' xExpr 'then' xStmts xElIfStmts 'else' xStmts 'end' 'if' .
xElIfStmt ::=   'elseif' xExpr 'then' xStmts .
xElIfStmts ::=   xElIfStmts xElIfStmt .
xElIfStmts ::=   .
%
% Case statements:
xStmt ::=   'case' xExpr xCaseStmts 'else' xStmts 'end' 'case' .
xStmt ::=   'case' xExpr xCaseStmts 'end' 'case' .
xCaseStmts ::=   xCaseStmts xcCaseStmt .
xCaseStmts ::=   xcCaseStmt .
xcCaseStmt ::=    xcCaseList xStmts .
xcCaseList ::=    'when' xExprList '=>' .
%
% Loop statements:
xStmt ::=   xcLoopStmt 'end' 'loop' .
xStmt ::=   xcForStmt 'end' 'for' .
xStmt ::=   xcWhileStmt 'end' 'while' .
xStmt ::=   xcWhilefound 'end' 'whilefound' .
xStmt ::=   xcUntilStmt 'end' 'do' .
xStmt ::=   id ':' xLoops 'end' id .
xLoops ::=   xcLoopStmt .
xLoops ::=   xcForStmt .
xLoops ::=   xcWhileStmt .
xLoops ::=   xcWhilefound .
xLoops ::=   xcUntilStmt .
xcLoopStmt ::=   'loop' xStmts .
xcForStmt ::=   'for' xIterator 'do' xStmts .
xcWhileStmt ::=   'while' xExpr 'do' xStmts .
xcWhilefound ::=   'whilefound' xSimpleIts '|' xExpr 'do' xStmts .
xcUntilStmt ::=   'do' xStmts 'until' xExpr .
xStmt ::=   'quit' .
xStmt ::=   'quit' id .
xStmt ::=   'continue' .
xStmt ::=   'continue' id .
/********************************************
Iterators:
********************************************/
xIterator ::=   xSimpleIts '|' xExpr .
xIterator ::=   xSimpleIts .
```

```
xSimpleIts ::=   xSimpleIts ',' xSimpleIt .
xSimpleIts ::=   xSimpleIt .
xSimpleIt ::=   xLValue 'in' xExpr .
xSimpleIt ::=   xLValue '=' id xMapSel .
/*******************************************
Map Selectors for simple iterators:
*******************************************/
xMapSel ::=   '(' xcLValList ')' .
xMapSel ::=   '{' xcLValList '}' .
xcLValList ::=   xcLValList ',' xLValue .
xcLValList ::=   xLValue .
/*******************************************
Left hand side values:
*******************************************/
xLValue ::=   xcSimpleLV .
xcSimpleLV ::=   id .
xcSimpleLV ::=   id xSelector .
xLValue ::=   '[' xcComps ']' .
xcComps ::=   xcComps ',' xcComp .
xcComps ::=   xcComp .
xcComp ::=   xLValue .
xcComp ::=   '-' .
/*******************************************
Selectors:
*******************************************/
xSelector ::=   '(' xExprList ')' .
xSelector ::=   '{' xExprList '}' .
xSelector ::=   '(' xExpr '..' ')' .
xSelector ::=   '(' xExpr '..' xExpr ')' .
/*******************************************
Former:
*******************************************/
xFormer ::=   xExpr .
xFormer ::=   xExpr ',' xExprList .
xFormer ::=   xExpr '..' xExpr .
xFormer ::=   xExpr ',' xExpr '..' xExpr .
xFormer ::=   xExpr ':' xIterator .
/*********************************************
Expressions:
*******************************************/
xExprList ::=   xExprList ',' xExpr .
xExprList ::=   xExpr .
/*******************************************
Primary Expressions:
*******************************************/
xcPrimary ::=   id .
xcPrimary ::=   int .
xcPrimary ::=   float .
xcPrimary ::=   str .
xcPrimary ::=   'true' .
xcPrimary ::=   'false' .
xcPrimary ::=   'om' .
xcPrimary ::=   'atom' .
xcPrimary ::=   'boolean' .
xcPrimary ::=   'integer' .
xcPrimary ::=   'real' .
xcPrimary ::=   'string' .
xcPrimary ::=   'tuple' .
xcPrimary ::=   'set' .
xcPrimary ::=   'proctype' .
xcPrimary ::=   'argv' .
xcPrimary ::=   '{' xFormer '}' .
xcPrimary ::=   '[' xFormer ']' .
xcPrimary ::=   xUnOp xcPrimary .
xcPrimary ::=   xBinOp '/' xcPrimary .
%
% Quantifiers:
xcPrimary ::=   xQuantifier .
xQuantifier ::=   xQualifier xSimpleIts '|' xcPrimary .
xQualifier ::=   'exists' .
xQualifier ::=   'notexists' .
xQualifier ::=   'forall' .
%
% Conditional Expressions:
xcPrimary ::=   'if' xExpr 'then' xExpr xElIfExprs 'end' 'if' .
xcPrimary ::=   'if' xExpr 'then' xExpr xElIfExprs 'else' xExpr 'end' 'if' .
xElIfExprs ::=   xElIfExprs xElIfExpr .
xElIfExprs ::=   .
xElIfExpr ::=   'elseif' xExpr 'then' xExpr .
%
% Case Expressions:
xcPrimary ::=   'case' xExpr xCaseExprs 'end' 'case' .
xcPrimary ::=   'case' xExpr xCaseExprs 'else' xExpr 'end' 'case' .
xCaseExprs ::=   xCaseExprs xcCaseExpr .
xCaseExprs ::=   xcCaseExpr .
xcCaseExpr ::=   xcCaseList xExpr .
%
% Lambda Expressions:
xcPrimary ::=   xLambda .
xLambda ::=   'lambda' xParamList ':' xProgBody 'end' 'lambda' .
%
% Recursive lambda calls:
xcPrimary ::=   'self' '(' xExprList ')' .
xcPrimary ::=   'self' '(' ')' .
xcPrimary ::=   'self' .
%
xcPrimary ::=   id xSelector .
xcPrimary ::=   id '(' ')' .
%
xcPrimary ::=   '(' xExpr ')' .
/*****************************************
Binary operations:
*****************************************/
xExpr ::=   xExpr xcOrOp xcOrTerm / xcOrTerm .
xcOrOp ::=   'or' .
xcOrTerm ::=  xcOrTerm xcAndOp xcAndTerm / xcAndTerm .
xcAndOp ::=   'and' .
xcAndTerm ::=  xcAndTerm xcBoolOp xcBoolTerm / xcBoolTerm .
```

```
xcBoolOp ::=  '=' .
xcBoolOp ::=  '/=' .
xcBoolOp ::=  '<' .
xcBoolOp ::=  '<=' .
xcBoolOp ::=  '>' .
xcBoolOp ::=  '>=' .
xcBoolOp ::=  'in' .
xcBoolOp ::=  'notin' .
xcBoolOp ::=  'subset' .
xcBoolOp ::=  'incs' .
xcBoolTerm ::=  xcBoolTerm xcSetOp xcSetTerm / xcSetTerm .
xcSetOp ::=  'with' .
xcSetOp ::=  'less' .
xcSetOp ::=  'lessf' .
xcSetTerm ::=  xcSetTerm xcAddOp xcAddTerm / xcAddTerm .
xcAddOp ::=  '+' .
xcAddOp ::=  '-' .
xcAddOp ::=  'max' .
xcAddOp ::=  'min' .
xcAddTerm ::=  xcAddTerm xcMulOp xcMulTerm / xcMulTerm .
xcMulOp ::=  '*' .
xcMulOp ::=  'div' .
xcMulOp ::=  'mod' .
xcMulTerm ::=  xcMulTerm xcPowOp xcPowTerm / xcPowTerm .
xcPowOp ::=  '**' .
xcPowTerm ::=  xcPrimary .
%
xBinOp ::=  xcOrOp .
xBinOp ::=  xcAndOp .
xBinOp ::=  xcBoolOp .
xBinOp ::=  xcSetOp .
xBinOp ::=  xcAddOp .
xBinOp ::=  xcMulOp .
xBinOp ::=  xcPowOp .
/*******************************************
Unary Operators:
*******************************************/
xUnOp ::=  '+' .
xUnOp ::=  '-' .
xUnOp ::=  '#' .
xUnOp ::=  'not' .
xUnOp ::=  'pow' .
xUnOp ::=  'arb' .
xUnOp ::=  'select' .
xUnOp ::=  'domain' .
xUnOp ::=  'range' .
xUnOp ::=  'type' .
xUnOp ::=  'is_map' .
xUnOp ::=  'is_smap' .
```

## B The symbol equivalences

```
xExprList ::=
       xcCaseList.

xExpr ::=
       xcPrimary xcMulTerm xcPowTerm xcOrTerm xcAddTerm xcAndTerm xcBoolTerm
       xcSetTerm.

xBinOp ::=
       xcOrOp xcMulOp xcPowOp xcAddOp xcAndOp xcBoolOp xcSetOp.

xSingleVar ::=
       xcVars.

xParamList ::=
       xcParams.

xCaseExprs ::=
       xcCaseExpr.

xCaseStmts ::=
       xcCaseStmt.

xLoops ::=
       xcLoopStmt xcForStmt xcWhileStmt xcWhilefound xcUntilStmt.

xLValue ::=
       xcComps xcComp xcLValList xcSimpleLV.
```

## C The static conditions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Static conditions:
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Header and trailer names:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

RULE rProgDefn:
   xProgDefn ::= 'program' id  ';'
 xProgBody
                 'end' id ';'
STATIC
  CONDITION Equal (id[1].sym, id[2].sym,
           'Program name in header and trailer have to be identical');
END;

RULE rProcDefn:
   xProcDefn ::= 'procedure' id xParamList ';'
 xProgBody
 'end' id ';'
STATIC
   CONDITION Equal (id[1].sym, id[2].sym,
       'Procedure name in Header and Trailer have to be identical');
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Control statements with labels:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

RULE rStmtLS: xStmt ::= id ':' xLoops 'end' id
STATIC
CONDITION Equal (id[1].sym, id[2].sym,
       'Label name in Header and Trailer have to be identical');
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Constants initialized?
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

NONTERM xDecl, xDeclKey:  IsConst: BOOL SYNT;

RULE rDecl: xDecl ::= xDeclKey xSingleVar ';'
STATIC
    xDecl.IsConst := xDeclKey.IsConst;
END;

RULE rDeclKeyV: xDeclKey ::= 'visible'
STATIC
    xDeclKey.IsConst := FALSE;
```

```
END;
RULE rDeclKeyH: xDeclKey ::= 'hidden'
STATIC
    xDeclKey.IsConst := FALSE;
END;
RULE rDeclKeyVC: xDeclKey ::= 'visible' 'constant'
STATIC
    xDeclKey.IsConst := TRUE;
END;
RULE rDeclKeyHC: xDeclKey ::= 'hidden' 'constant'
STATIC
    xDeclKey.IsConst := TRUE;
END;
RULE rDeclKeyC: xDeclKey ::= 'constant'
STATIC
    xDeclKey.IsConst := TRUE;
END;

RULE rSingleVar2: xSingleVar ::= id
STATIC
    CONDITION Assert (NOT (INCLUDING xDecl.IsConst),
                 'It is necessary to initialize a constant!');
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conditions for return:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

RULE rStmtRET1: xStmt ::= 'return' xExpr
STATIC
   CONDITION Assert (INCLUDING xProgBody.InProcedure,
     'Return statements are only allowed inside of procedures');
END;
RULE rStmtRET2: xStmt ::= 'return'
STATIC
   CONDITION Assert (INCLUDING xProgBody.InProcedure,
     'Return statements are only allowed inside of procedures');
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conditions for self:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

RULE rStmtSelf1: xStmt ::= 'self' '(' xExprList ')'
STATIC
   CONDITION Assert (INCLUDING xProgBody.InLambda,
     'Recursive lambda calls are only allowed inside of lambdas');
END;
RULE rStmtSelf2: xStmt ::= 'self' '(' ')'
STATIC
   CONDITION Assert (INCLUDING xProgBody.InLambda,
     'Recursive lambda calls are only allowed inside of lambdas');
END;
```

```
RULE rExprSelf1: xExpr ::= 'self' '(' xExprList ')'
STATIC
   CONDITION Assert (INCLUDING xProgBody.InLambda,
     'Recursive lambda calls are only allowed inside of lambdas');
END;
RULE rExprSelf2: xExpr ::= 'self' '(' ')'
STATIC
   CONDITION Assert (INCLUDING xProgBody.InLambda,
     'Recursive lambda calls are only allowed inside of lambdas');
END;
RULE rExprSelf3: xExpr ::= 'self'
STATIC
   CONDITION Assert (INCLUDING xProgBody.InLambda,
     'Recursive lambda calls are only allowed inside of lambdas');
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Procedure and lambda environment:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

NONTERM xProgBody:  InProcedure, InLambda: BOOL INH;

RULE rProgDefn:
   xProgDefn ::= 'program' id  ';'
                  xProgBody
               'end' id ';'
STATIC
  xProgBody.InProcedure := FALSE;
  xProgBody.InLambda := FALSE;
END;

RULE rProcDefn:
   xProcDefn ::= 'procedure' id xParamList ';'
                  xProgBody
               'end' id ';'
STATIC
   xProgBody.InProcedure := TRUE;
   xProgBody.InLambda := FALSE;
END;

RULE rLambda:
   xLambda ::= 'lambda' xParamList ':'
                 xProgBody
             'end' 'lambda'
STATIC
   xProgBody.InProcedure := TRUE;
   xProgBody.InLambda := TRUE;
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conditions for quit and continue:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

RULE rQuitStmt: xStmt ::= 'quit'
```

```
STATIC
   CONDITION Assert (INCLUDING xStmts.InLoop,
     'Quit statements are only allowed inside of loops');
END;

RULE rQuitLStmt: xStmt ::= 'quit' id
STATIC
   CONDITION Assert (INCLUDING xStmts.InLoop,
     'Quit statements are only allowed inside of loops');
END;

RULE rContinueStmt: xStmt ::= 'continue'
STATIC
   CONDITION Assert (INCLUDING xStmts.InLoop,
     'Continue statements are only allowed inside of loops');
END;

RULE rContinueLStmt: xStmt ::= 'continue' id
STATIC
   CONDITION Assert (INCLUDING xStmts.InLoop,
     'Continue statements are only allowed inside of loops');
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Loop environment:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

NONTERM xStmts:    InLoop: BOOL INH;

RULE rProgBody: xProgBody ::= xDecls xStmts xProcDefns
STATIC
    xStmts.InLoop := FALSE;
END;

RULE rStmts1: xStmts ::= xStmts xStmt ';'
STATIC
   TRANSFER InLoop;
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conditional statements:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

RULE rIfStmt: xStmt ::= 'if' xExpr 'then' xStmts
                           xElIfStmts
                        'end' 'if'
STATIC
    xStmts.InLoop := INCLUDING xStmts.InLoop;
END;

RULE rIfElStmt: xStmt ::= 'if' xExpr 'then' xStmts
                             xElIfStmts
                           'else' xStmts
                           'end' 'if'
```

```
STATIC
    xStmts[1].InLoop := INCLUDING xStmts.InLoop;
    xStmts[2].InLoop := INCLUDING xStmts.InLoop;
END;

RULE rElIfStmt: xElIfStmt ::= 'elseif' xExpr 'then' xStmts
STATIC
    xStmts.InLoop := INCLUDING xStmts.InLoop;
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Case statements:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

RULE rSwitchEStmt: xStmt ::= 'case' xExpr xCaseStmts
                             'else' xStmts
                             'end' 'case'
STATIC
    xStmts.InLoop := INCLUDING xStmts.InLoop;
END;

RULE rCaseStmt: xCaseStmts ::= xExprList xStmts
STATIC
    xStmts.InLoop := INCLUDING xStmts.InLoop;
END;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Loops:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

RULE rLoopStmt: xLoops ::= 'loop' xStmts
STATIC
    xStmts.InLoop := TRUE;
END;

RULE rForStmt: xLoops ::= 'for' xIterator 'do'
                                xStmts
STATIC
    xStmts.InLoop := TRUE;
END;

RULE rWhileStmt: xLoops ::= 'while' xExpr 'do'
                                    xStmts
STATIC
    xStmts.InLoop := TRUE;
END;

RULE rWhilefound: xLoops ::= 'whilefound' xSimpleIts '|' xExpr 'do'
                                          xStmts
STATIC
    xStmts.InLoop := TRUE;
END;

RULE rUntilStmt: xLoops ::= 'do' xStmts 'until' xExpr
```

```
STATIC
    xStmts.InLoop := TRUE;
END;
```

## References

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

[CO90] G. Clemm and L. Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

[DF89] E.E. Doberkat and D. Fox. *Software Prototyping mit SETL.* Leitfäden und Monographien der Informatik. Teubner-Verlag, Stuttgart, 1989.

[DFGH90] E.E. Doberkat, W. Franke, U. Gutenbeil, and W. Hasselbring. SETL/E Sprachbeschreibung Revision 0.3. Internal memo, University of Essen, December 1990.

[DGH90a] E.E. Doberkat, U. Gutenbeil, and W. Hasselbring. SETL/E – A prototyping system based on sets. In W. Zorn, editor, *Tagungsband TOOL90*, pages 109–118. University of Karlsruhe, November 1990.

[DGH90b] E.E. Doberkat, U. Gutenbeil, and W. Hasselbring. SETL/E Sprachbeschreibung Version 0.1. Informatik-Bericht 01-90, University of Essen, March 1990.

[GHK+90] R.W. Gray, V.P. Heuring, S.P. Krane, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. Software Engineering Group Report 89-1-1, University of Colorado, Boulder, June 1990.

[Gra89] R. Gray. GLA: Non-literal symbol specification. Technical report, Electrical and Computer Engineering Department, University of Colorado, Boulder, 1989.

[Gro89a] Compiler Tools Group. CAGT reference manual. Technical report, Electrical and Computer Engineering Department, University of Colorado, Boulder, 1989.

[Gro89b] Compiler Tools Group. Eli user interface reference manual. Technical report, Electrical and Computer Engineering Department, University of Colorado, Boulder, 1989.

[Kas80] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.

[Kas90a] U. Kastens. LIDO: A specification language for attribute grammars. Technical report, University of Paderborn, 1990.

[Kas90b] U. Kastens. LIGA: A language independent generator for attribute evaluator. Technical report, University of Paderborn, 1990.

[SDDS86] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets – An Introduction to SETL.* Graduate Texts in Computer Science. Springer-Verlag, 1986.

[Sny90a] W.K. Snyder. The SETL2 programming language. Technical Report 490, New York University, September 1990.

[Sny90b] W.K. Snyder. The SETL2 programming language: Update on current developments. Technical report, New York University, September 1990.