An Exception Handling Construct for Functional Languages

Manfred Bretz, Jürgen Ebert
EWH Koblenz, Informatik
D-5400 Koblenz/West Germany

1. Introduction and Overview

Exception handling is a way of dealing with situations at program runtime, which could affect program reliability. Exception handling covers error handling and error recovery, as well as programming techniques for dealing with legal but presumably rare, thus "exceptional", situations.

Even for conventional (von Neumann-) languages there are relatively few workable approaches to this problem, compared to the overwhelming number of papers on other program constructs. Only a few languages have a construct for exception handling, the most important being PL/I [19], ADA [16] and CLU [15]. Basic conceptual work on exception handling has been done by Cristian [4,5], Goodenough [9] and Yemini&Berry [21,22].

The so-called replacement model of Yemini and Berry seems to be the most powerful approach, since it allows a variety of handling options, like resuming the interrupted operation, retrying the interrupted operation in a changed state, or terminating the interrupted operation in a defined way. The model adopts an expression-oriented von Neumann view, using ALGOL 68 as the host language to carry the proposed constructs.

If a language is conformant to the paradigm of functional programming, there are some additional basic problems, when an exception handling construct is to be introduced:

a) There is a fundamental conflict between parallel/nondeterministic function evaluation on one hand and sequential/deterministic evaluation on the other hand (independent of which evaluation strategy is followed). If there are (e.g.) two exceptional points inside a given function the result of a corresponding parallel function application could be different, depending on which signal operation is executed first (e.g. using the language described below, the function

```
def f := λx.(signal I 3) + (signal I 5)
          signals I;
```

in an application like

```
f(1) handle I:= λx.x terminate
```

could yield *9* or *5* as its result).

b) Exception handling might cause side effects in expression evaluation and hence might violate the property of referential transparency. Anything done by a handler to remove an exception occurrence within an expression is a side effect, since it depends on the environment in which the exception is evaluated instead of where it is defined.

c) Since there may be higher order functions, which yield functions as their results, an unrestricted use of exceptions may lead to situations where knowledge of which exceptions might be signalled inside a given expression might be lost.

Because of these difficulties today's functional programming languages do not have strong exception handling facilities, the only major exception being ML [18] where there is a raise/handle construct. But ML allows only for termination as the single handler response.

In this paper, we show how the approach of Yemini and Berry can be brought into the context of functional programming, thus allowing resume, retry and terminate as handler responses. While problem a) is intrinsic to exception handling (thus, ML has a sequential semantics, too), we solve problem b) by introducing handlers (to a certain extent) as additional function parameters where exceptions can only explicitly be transferred into a different environment, and deal with problem c) by using a strong but polymorphic typing approach to restrict the use of exception handling to those cases, where security can be achieved.

In section 2, we define a language construct for exception handling, by introducing a sample ISWIM-like [12] language, called ALEX, as the basis of the discussion. We give some examples to show the usefulness of the approach and to explain informally the meaning and intentions of the construct. In section 3, we give the concrete semantics of ALEX using the operational SECD approach of [11]. Section 4 contains the type inference rules, which are an extension of the usual (polymorphic) type system for functional languages by an additional exception type and its consistency conditions. We finish the paper with a detailed example.

ALEX has been implemented on a UNIX-based system using graph technology according to [8]. The translator, which translates a given ALEX program into an internal graph representation, is built using the compiler tools LEX and YACC [13,10]. The representation is a directed, attributed and ordered graph which represents the abstract syntax as well as the dataflow of the given functional program, also being the internal code on which the evaluator operates [6]. Type inference, derived from the rules below, is done by building an additional type subgraph to the functional graph using the same approach. The

(functional) graph is taken as an input for a SECD-like graph interpreter which attributes values to its vertices according to [7].

ALEX, as described in the paper, is kept as simple as possible, since it is only used as a vehicle for describing the fundamental concepts and its formal background. Thus, the focus is not on pragmatics which are somewhat verbose for clarity's sake. But several practical extensions to ALEX are quite easy, some of which like

- multiple handlers
- default exception clauses
- unparameterized exceptions

have also been sucessfully added to the prototype implementation.

2. ALEX - An Applicative Language with a Language Construct for Exception Handling

This paper uses the applicative language ISWIM [12] to carry its exception handling proposal. The sample language is called ALEX. We present a decorated abstract syntax of ALEX, summarize the essentials of the exception handling construct and give three examples involving the construct.

2.1 Syntax

The language of ALEX expressions *expr* is given by the following decorated abstract syntax. The comments should help the reader to understand the intended meaning of the rules.

expr =

(a1) c
/* constant */

(a2) | id
/* identifier */

(a3) | "if" $expr_1$ "then" $expr_2$ "else" $expr_3$
/* conditional */

(a4) | $expr_1$ $expr_2$
/* application */

(a5) | $expr_1$ $expr_2$ "handle" handler
/* application with handler; associated with $expr_1$ may be an exception id in which case there must be a handler for id. */

(a6) | "λ" id "." expr
/* abstraction */

(a7) | "λ" id_1 "." expr "signals" id_2
/* abstraction with exception; the specified function may signal the exception id_2 i.e. may be a signaller. */

(a8) | "fix" id "." expr
/* fixpoint */

(a9) | "signal" id expr
/* signal; the exception id is signalled and $expr$ is the parameter of the signalled exception */

handler =

id_1 ":=" "λ" id_2 "." expr stat
/* $\lambda id_2.expr$ denotes the λ-expression (handler body) which is the paramerterized handler for the exception id_1 */

stat =

"resume"
| "retry"
| "terminate"

Remark:
We extend our notation by adding the ability to name ALEX-expressions. The syntax for definitions is: *"def" id ":=" expr ";"*. The definition facility is only introduced for abbrevation (and not for the definition of recursive functions).

2.2 Essentials of Exception Handling

The essential characteristics of the ALEX exception handling mechanism are [8,15,21]:

- Exceptions must be declared within the functions' interfaces.
- Handlers are statically bound to exceptions.
- The immediate invoker of a function is considered responsible to handle that function's exceptions.

- Exceptions can be propagated explicitly along the dynamic invocation chain.
- Resume, retry and terminate are the possible handler responses.
- Exceptions can be parameterized.

2.3 Examples

As an introduction to the exception handling mechanism we consider the following simple function abstraction:

```
def f := λx.if  x<0  then (signal I  x)>1
          signals I;   else true fi
```

Then, the function applications (a)-(c) yield the following results

(a) f(-5) **handle** I := λx.x+4 **resume**
The handler value $(\lambda x.x+4)(-5) = -1$ is calculated, and then the function f is resumed where it left off . Hence the result is *false*.

(b) f(-5) **handle** I := λx.x+4 **retry**
The handler value $(\lambda x.x+4)(-5) = -1$ is calculated, and then the function f is invoked again with -1 as new argument. This again leads to a signalling. Finally the result is *true* where two retries are done.

(c) f(-5) **handle** I := λx.false **terminate**
The handler value $(\lambda x.false)(-5) = false$ is calculated and is used as the value of the function application. Hence the result is *false*.

As a second, more instructive example the well-known curried while-functional

```
def while:= fix while.λp.λf.λx.if p(x) then (while p f) (f x)
                                 else x fi
```

may also be rewritten as a (functional) expression using the exception handling constructs:

```
def while1:= λp.(λx.if p(x) then (signal I x)
                      else x fi
                 signals I);
```

Assume that *pa* denotes an arbitrary predicate and *fa* an arbitrary function. Then, for all arguments x the evaluation of the expression

(while pa fa) (x)

yields the same result as the evaluation of the expression

(while1 pa) (x) **handle** I:= fa **retry**

As a third, more practical example, we develop a recursive ALEX function which converts a given sequence of integer numbers into a sequence of ASCII-characters [21,22]. The function takes as its input a list of integer numbers. For every list element the function tests whether there exists a corresponding ASCII-character. If the test yields *true* then the number's character representation is appended to the result list; otherwise the exception *Bad_code* is signalled:

```
def Convert :=
    fix Convert.
        λl.if null(l)  then <>  /*empty list */
            else
                if  (0≤head(l))  and  (head(l)≤127)
                    then chr(head(l)) /* chr is the transfer function in ALEX */
                    else (signal Bad_code head(l))
                fi :: Convert(tail(l)) handle Bad_code := λy.(signal Bad_code y) resume
            fi
        signals Bad_code;
```

Since for a function with a *signals-clause* a handler has to be given for every application of that function in this prototype language, at least a "dummy" handler for propagating the exception *Bad_code* has to be added in *Convert's* body. (This is forced by the typing rules, see section 4.)

To complete this example *Convert* is applied to a list *ll* twice.

- Convert(ll) **handle** Bad_code := λi.if i<0 then '-' else '+' fi **resume**
  The handler for *Bad_code* specifies that negative numbers are represented as '-' and numbers greater then 127 as '+'.

- Convert(ll) **handle** Bad_code := λi.<> **terminate**
  The handler for *Bad_code* specifies that the result of the application is the empty list.

3. Operational Semantics for ALEX

An operational semantics specifies a language by defining an interpreter for the abstract syntax of the language. For ALEX we develop a variant of the SECD machine proposed by Peter J. Landin in 1964 [11] as an interpreter for its abstract syntax. Our SECD machine supports lazy evaluation since our ALEX implementation is lazy. But that is not essential. Here only those parts concerning exception handling are

described. A full description of the interpreter is given in [1]. If the function *NEXT_STATE* yields no longer a new state, then *head(S)* is the result.

```
def NEXT_STATE:=
  λ[S,E,C,D].
      if is_empty (C) and not is_empty (D) then
      /* an intermediate evaluation has been terminated */
          let S == res::S0 and
              D == [S1, E1, C1]::D1 in
          if is_su(res) then
              let res == [su, expr, E2] in
              [<>,E2,<expr>,D]
          elsif is_sr(res) then
              let res == [sr, closure, [su, rand, E2]] in
              [<[su, rand, E2],  closure>,<>,<ap>,D]
          else
              [res::S1, E1, C1, D1]
          fi
      elsif not is_empty (C) then
          let C == X::C1 in
      (s1) if is_constant (X) or is_su (X) or is_sr (X) then
              [X::S, E, C1, D]
      (s2) elsif is_identifier (X) then
              [value(X, E)::S, E, C1, D]
          ...
      (s5) elsif is_application_with_handler (X) then
              letrec X == rator rand "handle" handler and
                  handler == exc_id ":="  "λ"bv"."body  stat in
              [<>,  E1,<rator,    [su, rand, E],  ap>,  D1]
              whererec D1 == [S, E, C1]::D and
                  E1 == (exc_id ← [stat,[cl,bv,body,E], D1, rator])::E
          ...
      (s7) elsif is_abstraction_with_exception (X) then
              let X ==  "λ"bv"."body  "signals" exc_id in
              [[cl, bv, body, E]::S, E, C1, D]
          ...
      (s9) elsif is_signal (X) then
              letrec X == "signal" exc_id rand and
                  [stat, closure, D1, rator] == value(exc_id, E) and
                  D1 == [S0, E0, C0]::D0 in
              if stat == "resume" then /* resumption */
                  [<[su, rand, E],  closure>,  E, ap::C1, D]
              elsif stat == "retry" then /* retrying */
                  [<>,  E1, <rator,[sr, closure, [su, rand, E]],  ap>,  D1]
                  where E1 == (exc_id ← [stat, closure, D1, rator])::E0
              else /* termination */
                  [<[su, rand, E],  closure>,<>,<ap>,D1]
              fi
          ...
          fi
      else
          /* no new state */
      fi
```

The following explanations point out how the SECD machine processes the exception handling constructs. The exception handling mechanism leads to two different kinds of activities at run time:

case (s5):

When an application with a handler is executed, the handler body has to be bound to the exception identifier together with the information needed to compute associated signal-expressions appropriately. Thus, execution continues with an enlarged environment. Since for the terminate- and retry-actions the computation has to continue from the current state, we associate the state in the form of a dump *D1* with the identifier. Retry-actions also need the current operator *rator*. Thus, we decided to associate also this information to *exc_id*.

case (s7):

An abstraction with exception does not lead to additional actions at execution time. The signals-clause is only used for type inference.

case (s9):

When an exception *exc_id* is signalled, different actions have to be done depending on the handler's status *stat*:

(s9a) resume means: the handler body should be applied and with that result execution continues.

(s9b) retry means: the handler body should be applied, but then execution should continue with the handler's application using the new result value as the new rand.

(s9c) terminate means: the handler body should be applied and the execution should continue with that result as the result of the handler's application.

Note also, that to make the re-application at retry really lazy we need a structure to suspend cl/su-pairs. This is done by a new intermediate result, a sr-suspension, in the implementation.

As another example, note that the overall discipline of the SECD machine could be described using the exception handling constructs of ALEX, as well.

```
SECD_MACHINE:=
    λexpr.  NEXT_STATE1([<>,<>,<expr>,<>])
                handle finish:=  λ[S,E,C,D].head(S)  terminate
```

In this case the state-transition function *NEXT_STATE1* should call itself recursively and signal an exception *finish*, if there is no successor state.

```
def NEXT_STATE1:=
    fix NEXT_STATE1.
        λ[S,E,C,D].
            NEXT_STATE1(
                        if is_empty (C) and not is_empty (D) then
                            /* an intermediate evaluation has been terminated */
                            ...
                        elsif not is_empty (C) then
                            ...
                        else
                            /* no new state */
                            signal finish [S,E,C,D]
                        fi)
            handle finish:= λx.(signal finish x) resume
        signals finish;
```

4. Type Inference in ALEX

Since there are higher order functions in functional languages, in an expression ($expr_1$ $expr_2$) the operator $expr_1$ can itself be an application expression whose evaluation yields a function which can signal an exception *exc_id* (cf. example 2 above). In this case there must a handler definition for *exc_id* be attached to the expression ($expr_1$ $expr_2$) because for our language construct we have required that the immediate invoker of a function handles that function's exceptions. We propose to check this by a polymorphic typechecking algorithm [3,14,17]. The type system of ALEX is formalized as a type deduction system [2] that prescribes how to establish the type of an expression from the types of its subexpressions. The following is a list of the inference rules ordered to parallel the abstract syntax of ALEX.

Let e be an expression and let π be a type assignment (i.e. a mapping from the identifiers occurring free in e into type expressions). The notation $\pi \vdash e : \tau$ means that given π we can deduce that e has type τ. The horizontal bar reads as "implies"; # is the overwrite operator for mappings. To force the correct use of the exception handling constructs, all neccessary information is collected in an exception type, which is constructed from

1) the exception identifier *exc_id*
2) the resume-handler type
3) the retry-handler type
4) the terminate-handler type

by using the type constructor *exc*.

Since exceptions are associated with abstractions, we also use a type constructor to model functions of the type "from τ_1 to τ_2 with an exception type τ_3" and write in mixfix notation "$\tau_1 \otimes \tau_3 \rightarrow \tau_2$".

Typing rules:

(r1) /* constant; c is a constant of type b */

$\pi \vdash c : b$

(r2) /* identifier */

$\pi \vdash id : \pi(id)$

(r3) /* conditional */

$$\frac{\pi \vdash e_1 : \text{bool},\ \pi \vdash e_2 : \tau,\ \pi \vdash e_3 : \tau}{\pi \vdash \text{“if”}\ e_1\ \text{“then”}\ e_2\ \text{“else”}\ e_3 : \tau}$$

(r4) /* application */

$$\frac{\pi \vdash e_1 : \tau_1 \rightarrow \tau_2,\ \pi \vdash e_2 : \tau_1}{\pi \vdash e_1\ e_2 : \tau_2}$$

(r5a) /* application with handler */

$$\frac{\pi \vdash e_1 : \tau_1 \otimes \text{exc}(\{id_1\},\tau_3,\tau_4,\tau_5) \rightarrow \tau_2 \qquad \pi \vdash e_2 : \tau_2,\ \pi \vdash \text{“}\lambda\text{”}\ id_2\ \text{“.”}\ e_3 : \tau_3}{\pi \vdash e_1\ e_2\ \text{“handle”}\ id_1\ \text{“:=”}\ \text{“}\lambda\text{”}\ id_2\ \text{“.”}\ e_3\ \text{“resume”} : \tau_2}$$

(r5b)

$$\frac{\pi \vdash e_1 : \tau_1 \otimes \text{exc}(\{id_1\},\tau_3,\tau_4,\tau_5) \rightarrow \tau_2 \qquad \pi \vdash e_2 : \tau_2,\ \pi \vdash \text{“}\lambda\text{”}\ id_2\ \text{“.”}\ e_3 : \tau_4}{\pi \vdash e_1\ e_2\ \text{“handle”}\ id_1\ \text{“:=”}\ \text{“}\lambda\text{”}\ id_2\ \text{“.”}\ e_3\ \text{“retry”} : \tau_2}$$

(r5c)

$$\frac{\pi \vdash e_1 : \tau_1 \otimes \text{exc}(\{id_1\},\tau_3,\tau_4,\tau_5) \rightarrow \tau_2 \qquad \pi \vdash e_2 : \tau_2,\ \pi \vdash \text{“}\lambda\text{”}\ id_2\ \text{“.”}\ e_3 : \tau_5}{\pi \vdash e_1\ e_2\ \text{“handle”}\ id_1\ \text{“:=”}\ \text{“}\lambda\text{”}\ id_2\ \text{“.”}\ e_3\ \text{“terminate”} : \tau_2}$$

(r6) /* abstraction */

$$\frac{\pi\ \#\ [id{:}\tau_1] \vdash e : \tau_2}{\pi \vdash \text{“}\lambda\text{”}\ id\ \text{“.”}\ e : \tau_1 \rightarrow \tau_2}$$

(r7) /* abstraction with exception */

$$\frac{\pi\ \#\ [id_1{:}\tau_1,\ id_2{:}\text{exc}(\{id_2\},\tau_3,\ \tau_4 \rightarrow \tau_1,\tau_4 \rightarrow \tau_2)] \vdash e : \tau_2}{\pi \vdash \text{“}\lambda\text{”}\ id_1\ \text{“.”}\ e\ \text{“signals”}\ id_2 : \tau_1 \otimes \text{exc}(\{id_2\},\tau_3,\ \tau_4 \rightarrow \tau_1,\tau_4 \rightarrow \tau_2) \rightarrow \tau_2}$$

(r8) /* recursive functions */

$$\frac{\pi\ \#\ [id{:}\tau] \vdash e : \tau}{\pi \vdash \text{“fix”}\ id\ \text{“.”}\ e : \tau}$$

(r9) /* signal */

$\pi \vdash$ id : exc({id}, $\tau_1 \rightarrow \tau_2, \tau_1 \rightarrow \tau_3, \tau_1 \rightarrow \tau_4$)

$\pi \vdash$ e : τ_1

$\pi \vdash$ "signal" id e : τ_2

A bottom-up typing algorithm can be extracted from the rules in a straightforward way [2].

5. A Detailed Example

Suppose l is a sorted list. We develop a recursive function which inserts an element x into l in case that x is not already a member of l; otherwise the exception *Multiple* is signalled. (The reader might excuse that we use multi-argument functions without introducing them formally, too.):

```
def Insert:=
    fix Insert.
        λ[x,l].
            if null(l) then <x>
            else
                if  x<head(l)  then x :: l
                elsif x=head(l) then (signal Multiple (x, head(l)))  ||  l
                else
                    head(l) :: Insert(x, tail(l))
                                handle Multiple:= λ[x,y].(signal Multiple(x,y)) resume
                fi
            fi
        signals Multiple;
```

We proceed as follows: first we focus on the bottom-up derivation of the type of *Insert* (or more correctly of the type of the ALEX-expression which we have named *Insert*); second we present some reasonable applications of *Insert*.

The bottom-up derivation of the type of *Insert* can be understood by looking at the following principal snapshots of the derivation, where an a priori typing of the list handling operations is assumed:

1. The type of the else-part of the inner conditional is derived using the typing rules (r5a) and (r9), together with some other rules. The resulting type of it is

   "αlist"

   and the type assignment π contains the following types for the free identifiers:

x: α'
l: αlist
Multiple: exc({*Multiple*}, $\beta\times\beta'\to\gamma$, $\beta\times\beta'\to\delta$, $\beta\times\beta'\to\zeta$)
Insert: ($\alpha\times\alpha$list) $\otimes$ exc({*Multiple*}, $\beta\times\beta'\to\gamma$, κ, ι) $\to$ αlist

2. Next, we type the body of the abstraction with exception by using the rules (r3) and (r9) especially. Thus,

"αlist"

is also the type of the abstraction's body and the free identifiers have types as follows:

x: α
l: αlist
Multiple: exc({*Multiple*}, $\alpha\times\alpha\to\alpha$list, $\alpha\times\alpha\to\delta$, $\alpha\times\alpha\to\zeta$)
Insert: ($\alpha\times\alpha$list) $\otimes$ exc({*Multiple*}, $\beta\times\beta'\to\gamma$, κ, ι) $\to$ αlist

3. According to rule (r7), then the abstraction with exception has type

"($\alpha\times\alpha$list)$\otimes$exc({*Multiple*}, $\alpha\times\alpha\to\alpha$list, $\alpha\times\alpha\to\alpha\times\alpha$list, $\alpha\times\alpha\to\alpha$list)$\to\alpha$list"

where π contains for *Insert* the type

Insert: ($\alpha\times\alpha$list) $\otimes$ exc({*Multiple*}, $\beta\times\beta'\to\gamma$, κ, ι) $\to$ αlist

4. Finally, the rule for recursive function definitions (r8) yields for the whole expression the type

"($\alpha\times\alpha$list)$\otimes$exc({*Multiple*}, $\alpha\times\alpha\to\alpha$list, $\alpha\times\alpha\to\alpha\times\alpha$list, $\alpha\times\alpha\to\alpha$list)$\to\alpha$list"

where the type assignment π is empty.

There are several possibilities for handlers for an application of *Insert*. Assume, *nl* is a sorted list and n is a new entry. The following applications with handler are only some significant examples out of a variety of possibilities:

(1) Insert(n,nl) **handle** Multiple:= λ[new,old].<new> **resume**
/* replaces old entry by new entry */
(2) Insert(n,nl) **handle** Multiple:= λ[new,old].<old> **resume**
/* keeps old entry only */

(3) Insert(n,nl) **handle** Multiple:= λ[new,old].<new,old> **resume**
/* stores new entry in front of old one */

(4) Insert(n,nl) **handle** Multiple:= λ[new,old].<> **resume**
/* deletes both entries */

(5) Insert(n,nl) **handle** Multiple:= λ[new,old].nl **terminate**
/* same as case (2) */

(6) Insert(n,nl) **handle** Multiple:= λ[new,old].[modify(new),nl] **retry**
/* tries to insert a modified entry into nl */

Conclusion

An exception handling proposal for applicative languages which seems quite powerful and conceptually simple has been presented. The appropriate language construct was defined by introducing an ISWIM-like language, called ALEX. Its abstract syntax and its operational semantics were specified, for the latter using a variant of the SECD maschine. A type system for ALEX has been developed to restrict the use of the exception handling constructs, whereby security can be achieved.

Finally, a brief comparison between the ML exception handling mechanism [18] and our mechanism may show the benefits of our proposal:

(a) ML only supports the handler response "terminate the signaller". The signalling expression is terminated and the handler's result replaces the result of the (signalling) expression. ALEX supports with resume, retry and terminate three different possible handler responses.

(b) In ML exceptions are propagated automatically along the dynamic invocation chain as long as no handler is found. As was pointed out in [15] multilevel mechanisms are in contrast to the hierachical program design methodology. In ALEX exceptions must be propagated explicitly along the dynamic invocation chain and handlers are bound statically to exceptions.

(c) In both, ML and ALEX, exceptions can be parameterized.

On the other hand, the algorithmic language Scheme [20] provides the possibility to program with continuations, which allows management of control in a general and powerful manner. Scheme's "call-with-current-continuation" feature is useful for implementing a wide variety of control structures, including exception handling. To us this possibility seems to general whereas the proposal in this paper was meant to be specific to exception handling. We hope, it could become a practical and versatile help for programming.

References:

[1] **Bretz, M.:**
Exception Handling in Functional Programs,
in: W.-M. Lippe (Hrsg.),
"4. Workshop - Alternative Konzepte für Sprachen und Rechner",
Universität Münster, Schriftenreihe "Angewandte Mathematik und Informatik", Band 2/87-I

[2] **Bretz, M.; Ebert, J.:**
Type Inference for Exception Handling,
Internal Report, EWH Koblenz, 1987

[3] **Cardelli, L.:**
Basic Polymorphic Typechecking,
Science of Computer Programming, 8(1987), pp. 147-172

[4] **Cristian, F.:**
Robust Data Types,
Acta Informatica, 17(1982), pp.365-397

[5] **Cristian, F.:**
Dependable Programs: Concepts and Terminology,
IBM Research Laboratory, San Jose, CA, 1986 (Technical Report)

[6] **Ebert, J.:**
Graph Implementation of a Functional Language,
in: H. Noltemeier (ed.),
Proceedings of the WG' 85,
Trauner, Linz, 1985, pp. 73-84

[7] **Ebert, J.:**
Ein SECD-artiger Graphenauswerter,
in: W.-M. Lippe (Hrsg.),
"4. Workshop - Alternative Konzepte für Sprachen und Rechner",
Universität Münster, Schriftenreihe "Angewandte Mathematik und Informatik", Band 2/87-I

[8] **Ebert, J.:**
A Versatile Data Structure for Edge-Oriented Graph Algorithms,
Comm. ACM, 30(6, 1987) (June 1987), pp. 513-519

[9] **Goodenough, J.B.:**
Exception Handling: Issues and a Proposed Notation,
Comm. ACM, 18(12, 1975), (Dec. 1975), pp. 683-696

[10] **Johnson, S.C.:**
YACC - Yet Another Compiler Compiler,
Bell Laboratories, Murray Hill, NJ, 1975 (CSTR 32)

[11] **Landin, P.J.:**
The Mechanical Evaluation of Expressions,
Computer Journal 6, 1964, pp. 308-320

[12] **Landin, P.J.:**
The Next 700 Programming Languages,
Comm. ACM, 9(3, 1966) (March 1966), pp. 157-166

[13] **Lesk, M.E.; Schmidt E.:**
LEX - A Lexical Analyzer Generator,
Bell Laboratories, Murray Hill, NJ, 1975 (CSTR 39)

[14] **Letschert, T.:**
Type Inference in the Presence of Overloading, Polymorphism and Coercions,
in: Tagungsband der 8ten Fachtagung "Programmiersprachen und Programmentwicklung", Zürich 1984, pp. 58-70

[15] **Liskov, B.H.; Snyder, A.:**
Exception Handling in CLU,
IEEE Trans. on Soft. Eng., 5(6, 1979) (Nov. 1979), pp. 546-558

[16] **Luckham, D.C.; Polak W.:**
Ada Exception Handling - An Axiomatic Approach,
ACM Trans. on Prog. Lang. Syst., 2(2, 1980) (April 1980), pp. 225-233

[17] **Milner, R:**
A Theory of Type Polymorphism in Programming,
Journal of Computer and System Sciences, 17(1978), pp. 348-375

[18] **Milner, R.:**
A Proposal for Standard ML,
ACM Conf. Record of the 1984 Symposium on Lisp and Functional Programming, 1984, pp. 184-197

[19] OS and DOS PL/I Language Reference Manual,
IBM Corporation, 1981

[20] **Rees, J.; Clinger W. et. al.:**
Revised Report on the Algorithmic Language Scheme,
SIGPLAN Notices, 21(12, 1986) (Dec. 1986), pp. 37-79

[21] **Yemini, S.; Berry, D.M.:**
A Modular Verifiable Exception Handling Mechanism,
ACM Trans. on Prog. Lang. Syst., 7(2, 1985) (April 1985), pp. 214-243

[22] **Yemini, S.; Berry, D.M.:**
An Axiomatic Treatment of Exception Handling in an Expression-Oriented Language,
ACM Trans. on Prog. Lang. Syst., 9(3, 1987) (July 1987), pp. 390-407