1 Introduction

This guide describes the data structure definition facilities that have been developed by the ExBed group for the implementation of the XE compiler.

The XE compiler has been written in Common Lisp because we had good experience in using a Lisp machine in the rapid development of prototype systems. However, plain Common Lisp has some drawbacks that cause problems when it is used in the development of large systems. It also lacks some facilities that would be beneficial in the implementation of a compiler. Therefore we have implemented various "small languages" on top of Common Lisp to be used in the implementation of the XE compiler. A LALR parser generator facility is described in [1] and this document describes the data structure definition facilities of the XE compiler.

The main problem with Lisp is that it is not strongly typed. When a large program is being implemented by several programmers one can be sure that much time will be spent tracking down "deletion errors", i.e., missing access forms in an expression, and other type errors that would be detected at compilation time when using a strongly typed language. In many cases these kinds of errors will not be correctly detected even at run-time. In the Symbolics Common Lisp, for example, if we define two struct types A and B with `defstruct` and use an object of type A as an object of type B, no error will be detected if the indices of the fields that we access are not greater or equal to the number of fields of A.

A compiler has usually a large number of data structures. However, these structures can be divided into a small number of categories that have much in common. For example, there may be many types of abstract syntax tree nodes, but they all have some common structure and some common functions that may be applied on them. Defining these data structures and corresponding functions directly with the facilities of Common Lisp, e.g., with `defstruct` and `defun`, would be rather tedious. Also, if plain Common Lisp were used, we would always have to find out the type of the node, e.g., with `case`, before we know how to access it. Inevitable modifications of the data structure definitions would cause changes to many places in the compiler.

These were the reasons why we decided to implement two data structure definition languages of our own, *AST* and *DG*. We chose the object oriented paradigm with methods and inheritance as a basis for these languages. AST supports the concise definition of abstract syntax tree node classes and functions (actually methods), that manipulate abstract syntax trees. DG is a general purpose object class definition language that contains multiple inheritance and methods. As a speciality, both these languages contain *structural methods* that support the efficient definition of a set of similar structure oriented methods, e.g., methods that copy abstract syntax trees.

In Sections 2 and 3 we describe the AST and DG languages and their implementation. In Section 4 we discuss our experiences of using these languages and explain some problems that we have found.

2 AST

AST (Abstract Syntax Tool) is an object oriented language for the definition of abstract syntaxes. Abstract syntax tree nodes are objects and methods can be applied to these objects. The structure of the abstract syntax defines a simple three level inheritance hierarchy.

2.1 Syntax definition

An abstract syntax consists of a set of *productions*. A production defines a *domain* and a set of possible *options* of that domain. The options consist of a *name* and a sequence of *fields*. Every field has a name and a type. The type is either *domain*, `&opt` *domain*, `&list` *domain*, or `&list1` *domain*. If it is *domain* the field must have a value and the value must be a node of *domain*. If it is `&opt` *domain*, if the value is given it must be a node of *domain*. If it is `&list` *domain* the value must be a list of nodes of *domain*. If it is `&list1` *domain* the value must be a non-empty list of nodes of *domain*. There may also be `&aux` fields that may contain data that is not of any abstract syntax domain, e.g., attribute values.

Following is a simple example of an AST definition:

```
(define-abstract-syntax syn
  :productions
  ((stmt (:if (condition expr) (then stmt) (else &opt stmt))
         (:begin-end (body &list stmt))
         (:assign (var idn) (value expr)))
   (expr (:add (op1 expr) (op2 expr))
         (:sub (op1 expr) (op2 expr))
         (:minus (op expr))
         (:var (var idn))
         (:int &aux (int integer))
         &aux (value integer))
   (idn (&aux (name string)))
   &aux (point point)))
```

This defines an abstract syntax called `syn` that has three domains, `stmt`, `expr`, and `idn`. The first production that defines domain `stmt` consists of three options. The first option, with the name `:if`, has three fields with names `condition`, `then`, and `else`. The types of the fields are `expr`, `stmt`, and `&opt stmt`. The `&opt` definition means that the `else` field may be missing. The value of the field `body` of the second option (`:begin-end`) must be a list of nodes of domain `stmt`. In the production that defines domain `idn` there is only one option and the option name may therefore be omitted. The name of the field is defined as an `&aux` field. The value of the field must be a Common Lisp string. In the production that defines domain `expr` there is also an `&aux` definition. It defines an auxiliary field for every option of that domain. There is also an `&aux` definition after the productions. It defines an auxiliary field for all options of all productions.

The AST compiler compiles the abstract syntax definition into a set of Common Lisp functions. For every option of a production a creator function with name *syntax*`$create-`*domain-option* is created. For every field of each option an accessor function with name *syntax*`$`*domain-option-field* is created. Also a set of type predicates is created. For every domain a type predicate with name *syntax*`$`*domain*`-p` is created, and for every option of each domain a type predicate *syntax*`$`*domain-option*`-p` is created. The accessor and creator functions check that the types of their arguments are legal.

In our example, the first production is compiled into the following Common Lisp functions:

```
;;; Creators
(defun syn$create-stmt-if (&key point condition then else) ...)
(defun syn$create-stmt-begin-end (&key point body) ...)
(defun syn$create-stmt-assign (&key point var value) ...)
;;;Accessors
(defun syn$stmt-if-condition (obj) ...)
(defun syn$stmt-if-then (obj) ...)
(defun syn$stmt-if-else (obj) ...)
(defun syn$stmt-begin-end-body (obj) ...)
(defun syn$stmt-assign-var (obj) ...)
(defun syn$stmt-assign-value (obj) ...)
;; Predicates
(defun syn$stmt-p (obj) ...)
(defun syn$stmt-if-p (obj) ...)
(defun syn$stmt-begin-end-p (obj) ...)
(defun syn$stmt-assign-p (obj) ...)
```

In the following example these and other functions that are compiled from the definition of the abstract syntax `syn` are used.

```
(defun make-inc-node (var-name increment)
  (let* ((expr1 (syn$create-expr-int :int increment))
```

```
        (var (syn$create-idn :name var-name))
        (expr2 (syn$create-expr-var :var var))
        (expr3 (syn$create-expr-add :op1 expr1 :op2 expr2)))
   (syn$create-stmt-assign :var var :value expr3)))

(let ((x (make-inc-node "a" 1)))
  (syn$idn-name (syn$stmt-assign-var x)))
;;; Result is "a"
```

The values of the fields may be modified with the Common Lisp `setf` form. E.g., the `var` of a `stmt-assign` node of the previous example may be modified in the following way:

```
(let ((x (make-inc-node "a" 1)))
  (setf (syn$idn-name (syn$stmt-assign-var x)) "b")
  (syn$idn-name (syn$stmt-assign-var x)))
;;; Result is "b"
```

2.2 Methods

The abstract syntax definition facility that we described above is useful per se. It facilitates the concise and clear definition of abstract syntaxes and gives a type checking mechanism. For example, the definition of the node types of our example using the `defstruct` facility of Common Lisp would be at least three times longer than the AST definition. In the Symbolic Common Lisp we would also have no type checking if we use `defstruct`.

However, manipulating abstract syntax trees only with the functions that the AST compiler generates would be tedious in some situations. In all structure oriented functions, e.g., functions that walk through syntax trees, we must find out the type of the node and then branch according to the type. When a new node type is implemented all branching expressions must be changed so that they reflect the new situation.

Methods provide a solution to this problem. A *generic function* is defined and then, for every node type, a method is written that implements the generic function. The generic function is called whenever we must branch according to the type of the node. When a new node type is defined we only have to write methods for that node type. Thus, the effects of defining (or deleting or changing) a node type are local.

A generic function can be defined with macro
`define-abstract-syntax-generic-function` . The form of the definition is:

(`define-abstract-syntax-generic-function` *function-name syntax-name*)

The methods are defined with macro `define-abstract-syntax-method` . The form of the definition is:

(`define-abstract-syntax-method` (*generic-function node-type*) *args*
body)

where *node-type* is the name of a syntax, a domain, or an option, *args* is a list of extraneous arguments to the generic function and *body* is a sequence of lisp forms that implement the method.

The form of the generic function call is:

(*generic-function node form1 form2* ...)

In the body of the method that implements *generic-function* for the type of *node* the variable `self` will be bound to *node*. The extraneous arguments of that method will be bound to the values of evaluating *form1*, *form2*, etc.

In the body of a method the fields of the node must be accessed with the functions that the AST compiler has created. It would be convenient if the fields could be accessed as normal variables, like in the bodies of the Zetalisp flavor methods, but this was too complicated to implement (because Common Lisp does not have symbol macros).

Here is an example of the use of methods.

```
(define-abstract-syntax-generic-function evaluate syn)

;;; 'env' contains (name . value) pairs
(define-abstract-syntax-method (evaluate syn$idn) (env)
  (let ((binding (assoc (syn$idn-name self) env
                        :test #'string-equal)))
    (cond ((null binding) 0)
          (t (rest binding)))))

(define-abstract-syntax-method (evaluate syn$expr-add) (env)
  (+ (evaluate (syn$expr-add-op1 self) env)
     (evaluate (syn$expr-add-op2 self) env)))

(define-abstract-syntax-method (evaluate syn$expr-sub) (env)
  (- (evaluate (syn$expr-add-op1 self) env)
     (evaluate (syn$expr-add-op2 self) env)))
```

```
(define-abstract-syntax-method (evaluate syn$expr-minus) (env)
  (- (evaluate (syn$expr-minus-op self) env)))

(define-abstract-syntax-method (evaluate syn$expr-var) (env)
  (evaluate (syn$expr-var-var self) env))

(define-abstract-syntax-method (evaluate syn$expr-int) (env)
  (syn$expr-int-int self))

(let* ((expr1 (syn$create-expr-int :int 1))
       (var (syn$create-idn :name "a"))
       (expr2 (syn$create-expr-var :var var))
       (expr3 (syn$create-expr-add :op1 expr1 :op2 expr2)))
  (evaluate expr3 '(("a" . 2))))
;;; Result is 3
```

Methods may be inherited. The abstract syntax definition implies a three-level class hierarchy (figure 1). The domains may be considered as abstract superclasses and the options as instantiable subclasses of their domain class. All domains of a syntax have a single abstract superclass, namely the whole syntax. Because the option classes are the only instantiable classes every node belongs to three classes: one option class, one domain class, and one syntax class.

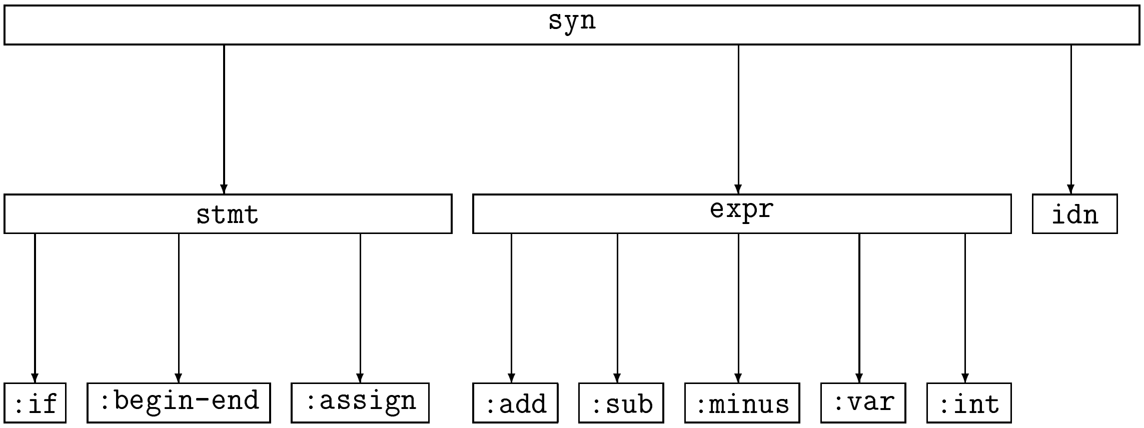

Figure 1: The abstract syntax inheritance hierarchy of `syn`.

A method can be defined for the whole syntax, for a whole domain, or for a single option. The methods that are defined for the whole syntax may be redefined for a domain or for an option, and the methods that are defined for a domain may be redefined for some options of that domain.

There is also a *wrapper* facility that can be used for the redefinition of a method. A wrapper is a piece of code that surrounds the actual method code. The wrapper code may, for example, examine the values of the arguments of the method, modify some of them, and then call the actual method. A wrapper is defined with macro `define-abstract-syntax-wrapper`. The form of the definition is:

(`define-abstract-syntax-wrapper` (*method-name node-type inner-function-variable*)
        args
    body)

where *inner-function-variable* is the name that the user can use when calling the actual method. Other arguments are as in `define-abstract-syntax-method`. Here is an example:

```
(define-abstract-syntax-wrapper (evaluate syn$idn inner) (env)
  (format t "~%Evaluating ~A" (syn$idn-name self))
  (funcall inner self env))
```

2.3 Structural Methods

There are situations where the method facility that we described above is not very useful. For example, using methods to implement a program that copies abstract syntax trees is not very much easier than implementing it with normal functions. We must write copy method for every node type. Although all copy methods are rather similar we cannot use inheritance effectively. Maintaining these methods after changes to the abstract syntax definition is also tedious.

To solve these problems *structural methods* were included in AST. A structural method is a macro that expands to a set of methods, one for each node type. In a structural method definition the programmer defines abstractly how nodes should be processed according to their structure. For example, the definition of a structural method that copies nodes of the abstract syntax that was defined in our first example is:

```
(define-abstract-syntax-structural-method (syn$copy syn) ()
  `(,option-creator
    ,@(mapcan #'(lambda (arg)
                 `(,(to-keyword (ast$arg-name arg))
                   ,(let ((self-access-form
                           `(,(ast$arg-accessor arg) self)))
                      (case (ast$arg-kind arg)
                        (&plain `(syn$copy ,self-access-form))
                        (&opt `(when ,self-access-form
                                 (syn$copy ,self-access-form))
```

```
                        ((&list &list1)
                         `(mapcar #'syn$copy ,self-access-form))))))
          args)
  ,@(mapcan #'(lambda (arg)
                `(,(to-keyword (ast$aux-name arg))
                  (,(ast$aux-accessor arg) self)))
          (append option-aux-defs
                  domain-aux-defs
                  global-aux-defs))))
```

When this definition is evaluated a `syn$copy` method will be created for every option of each production. The body of each method is obtained by evaluating the body of the structural method definition with the following variable bindings:

`option-creator`	The name of the creator function of the option.
`args`	The definitions of the non-auxiliary fields of the option.
`option-aux-defs`	The definitions of the auxiliary fields of the option.
`domain-aux-defs`	The definitions of the auxiliary fields of the domain.
`global-aux-defs`	The definitions of the auxiliary fields of the whole syntax.

The field definitions are accessed with the following functions:

`ast$arg-name`	The name of a non-auxiliary field.
`ast$arg-accessor`	The name of the accessor function of a non-auxiliary field.
`ast$arg-kind`	The type of a non-auxiliary field.
`ast$aux-name`	The name of an auxiliary field.
`ast$aux-accessor`	The name of the accessor function of an auxiliary field.

As we see, the definition of `syn$copy` is very dense. Whenever the abstract syntax definition is changed, we only have to recompile this definition, there is no need to edit any methods.

In some situations a simple structural method definition is not sufficient, because there are some special cases that should be handled differently. The special cases can then either be included in the structural method definition or, more elegantly, the programmer can write own methods or wrappers for those node types that need special treatment.

3 DG

DG (Descriptor Generator) is a general object class definition language that contains multiple inheritance and methods.

3.1 Class definitions

As opposed to most object oriented languages, a DG definition defines a set of classes, not just a single class. This permits an easy implementation, because all classes can be compiled at the same time. However, the compilations after modifications last longer.

Here is an example of a DG definition:

```
(define-descriptors windows
   :structures
   ((window
     :fields ((name string)
              (top int)
              (bottom int)
              (left int)
              (right int)
              (superior (&opt window))
              (inferiors (&list window))))
    (process
     :fields ((top-level-function (&union symbol lambda))))
    (bordered-window-mixin
     :instantiable nil
     :fields ((borders (&union int (&list int) symbol))))
    (bordered-window-process
     :parents (window process bordered-window-mixin))
    (:extern int :test fixnump)
    (:extern string :test stringp)
    (:extern symbol :test symbolp)
    (:extern lambda :test (lambda (x)
                            (and (listp x)
                                 (eq (first x) 'lambda))))))
```

The name `windows` identifies the definition. It is used as a prefix in the creation of operation and class names. The list after keyword `:structures` contains the class definitions. In the example eight classes are defined, namely `window`, `process`, `bordered-window-mixin` , `bordered-window-process` , `int`, `string`, `symbol`, and lambda. A class definition consists of a set of field definitions and an optional parent class definition. A parent class definition contains one or more class names, thus multiple inheritance is permitted. A field consists of the name and the type of the field. The type is either a class or a type expression built of class names and type expression operations. Type expression operations are: `&opt` meaning an optional field, `&list` meaning a list field, and `&union` meaning a disjoint union. For example, the value of the field `inferiors` of an object of the class `window` is a list of objects belonging to the class `window` or to some of its descendant classes. Type expressions may be nested. For example, the type of the field `borders` defines that the value of this field must be either an integer (defining the same border thickness for all borders), a list of integers (defining different thicknesses for different borders), or a symbol (a name of a variable that should be bound to either an integer or to a list of integers when an instance is created).

An object of a class has the fields that are defined in the definition of that class and all the fields that are defined in the ancestors of the class. Fields cannot be redefined in subclasses.

Because DG is not a self contained language there must be a facility for using objects of other kinds as field values of DG objects. This is accomplished with the `:extern` class definition. It contains the name of the extern class and optionally a type predicate. Extern classes may be used in field type definitions in the same way as normal DG classes. However, an extern class may not be the parent of some other class. In our example the classes `int`, `string`, `symbol`, and `lambda` are extern.

Some classes represent behavior that is only reasonable when that class is combined with some other class. In our example, the class `bordered-window-mixin` is such a class. When we combine this class with some window class we can make windows that have border lines of variable width. It is not reasonable to have objects of class `bordered-window-mixin` and therefore the class definition contains the definition `:instantiable nil` which prohibits instantiation.

The DG compiler compiles the class definitions to a set of Common Lisp functions. For every class a creator function with name *dg-name*`$create-`*class-name* is created, where *dg-name* is the name that identifies the definitions. Also a type predicate with name *dg-name*`$`*class-name*`-p` is created. For every field name that occurs in the class definitions a field accessor function with name *dg-name*`$`*field-name* is created. The accessor functions are polymorphic so that the same functions can be applied to objects of different classes.

Here is the result of the compilation of our example definitions.

```
;;; Predicates
(defun windows-p (x))
(defun windows$window-p (obj))
(defun windows$process-p (obj))
```

```
(defun windows$bordered-window-mixin-p (obj))
(defun windows$bordered-window-process-p (obj))
(defun windows$int-p (obj))
(defun windows$string-p (obj))
(defun windows$symbol-p (obj))
(defun windows$lambda-p (obj))
;;; Creators
(defun windows$create-window (&key name top bottom left right
                                   superior inferiors) ...)
(defun windows$create-process (&key top-level-function) ...)
(defun windows$create-bordered-window-process
  (&key name top bottom left right
        superior inferiors
        top-level-function
        borders) ...)
;;; Accessors
(defun windows$name (obj))
(defun windows$top (obj))
(defun windows$left (obj))
(defun windows$bottom (obj))
(defun windows$right (obj))
(defun windows$superior (obj))
(defun windows$inferiors (obj))
(defun windows$top-level-function (obj))
(defun windows$borders (obj))
```

The values of the fields of the objects may be modified with the Common Lisp `setf` form. In the following example the functions produced by the DG compiler are used.

```
;;; Create window w1.
(setf w1 (windows$create-window :name "Window-1"
                                :left 0
                                :top 0
                                :right 100
                                :bottom 100))
;;; Create window w2 inside window w1.
(setf w2 (windows$create-window :name "Window-2"
                                :left 10
                                :top 20
                                :right 40
                                :bottom 70
                                :superior w1))
(push w2 (windows$inferiors w1))
;;; Create a bordered window process bwp inside w1.
```

```
;;; The corners of the windows w1 and w2 are used in the creation.
(setf bwp (windows$create-bordered-window-process
            :name "Bordered Window Process-1"
            :left (windows$right w2)
            :top (windows$top w2)
            :right (windows$right w1)
            :bottom (windows$bottom w2)
            :superior w1
            :top-level-function #'zap
            :borders (list 3)))
(push bwp (windows$inferiors w1))
```

3.2 Methods

Methods can be written for the manipulation of the DG objects. The DG methods resemble the AST methods. First a generic function is defined and then for every class that should have the corresponding behavior a method is written that implements the behavior. When a method should be applied to an object the generic function is called. The function finds out the correct method and calls it.

A generic function is defined with the macro `dg$define-method-function`. The form of the definition is

`(dg$define-method-function` *generic-function-name arg1 arg2 ...*`)`

The methods are defined with macro `dg$define-method`. The form of the definition is

`(dg$define-method (`*dg-name method-name class name*`)` *args*
    body`)`

As in AST methods the variable `self` is bound to the object that the method is applied to.

Wrappers can be also defined for DG methods. The form of the definition is

`(dg$define-wrapper (`*dg-name method-name class-name previous-method-fn-var*`)`
        args
    body`)`

where *previous-method-fn-var* is a symbol chosen by the user. It is used when the original method is called.

3.3 Strutural methods

Like the AST language the DG language contains a structural method facility. The user interface of the facility is rather cumbersome and it is described in [2].

3.4 Access oriented functions

Functions that act like daemons reacting to accesses to a given field are often called access oriented. Access oriented functions are needed when the access of a field of an object should cause a side effect, for example, the call of some trace function. Access oriented functions can also be used for the implementation of lazy evaluation. Because this is the main use for access oriented functions in the XE compiler we use the term *lazy field* when referring to fields that are associated with access oriented functions.

A field can be declared lazy by listing the name of the field and a name of a function in the class definition after the keyword `:accessors`. This function is called when the accessors of the fields are created. It receives the following four arguments:

`self`	A variable that refers to the object whose field this is.
`direct`	A variable that refers to an array of the fields of the object.
`index`	The index of the lazy field in `direct`.
`field-name`	The name of the field.

Using these arguments the function should return a piece of code that implements the lazy behavior.

Here is an example. Let the function `superior-accessor` be defined in the following way.

```
(defun superior-accessor (self direct index field-name)
  (let ((value-var (make-symbol "VALUE")))
    `(let ((,value-var (svref ,direct ,index)))
       (format t "~%~A of ~A is ~A"
          ',field-name
          (windows$name ,self)
          (and ,value-var (windows$name ,value-var)))
       ,value-var)))
```

If we now replace the definition of class `window` of our class definition example with the following class definition

```
(window
 :fields ((name string)
          (top int)
          (bottom int)
          (left int)
          (right int)
          (superior (&opt window))
          (inferiors (&list window)))
 :accessors (((superior) superior-accessor)))
```

then every time the superior field of an object of type `window` or some of its descendant classes is accessed a trace message will be printed:

```
(windows$superior w2)
;;; Prints: "SUPERIOR of Window-2 is Window-1"
;;; Returns: w1
```

4 Experiences, Problems and Further Improvements

The AST and DG languages helped the development of the XE compiler in many ways. The AST and DG definitions are compact. In the case of AST this is achieved because AST is a special purpose language. In the case of DG this is mainly achieved with inheritance. The method definition facilities localize changes to the abstract syntax and class definitions, because the code that uses objects only calls generic functions. This kind of polymorphism is very beneficial and one might wonder whether this should also have been included in the XE language.

Maybe the biggest savings in programming and maintenance work were achieved with structural methods. The structural method definitions are compact. The XE compiler contains 10 structural method definitions. The size of a definition varies from 10 lines to 94 lines. The total size of the structural method definitions is 349 lines. The size of the macro expansion results is 18800 lines, thus on the average the expansion ratio is 54. Hand coding would have saved some of the 18800 lines, but anyway the savings have been thousands of lines. We do not know any other languages that contain structural methods, but this kind of facility would be very useful in other languages too.

The connection between the AST and DG languages and other data structure definitions, e.g. `defstructs`, that we used in the XE compiler, was a big problem. The `&aux` fields of the AST definitions and the `:extern` classes in the DG definitions did not solve the problem well. The problem would have been avoided if we had implemented AST on top of DG which would have been rather easy. We should also not have used any other data structure definition facilities. If we had had one uniform data structure definition language it would have been possible to implement a disk save facility for the XE environment. In the current situation it turned out to be a too complicated a task and could not be performed efficiently.

References

[1] ROCC Programmer's Guide. Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, Markku Tamminen, Helsinki Univ. of Tech., Dept. of Comp. Sci. report TKO-C38, 1989.

[2] XE Programming Environment Implementation Notes. Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, Markku Tamminen, Helsinki Univ. of Tech., Dept. of Comp. Sci. report TKO-C40, 1989.