Report on the EvE Programming Language

Wolfgang Schreiner
Parallel Computation Laboratory
RISC-Linz

November 1989

Abstract

EvE is a functional language designed for the research on programming parallel computers, especially dataflow machines. The language combines a simple syntax and semantics with most of the features widely used in functional programming: strong typing, higher order functions, delayed evaluation, non-strict functions and data types. This report defines the syntax of the language and informally describes its semantics on the basis of examples. A major program shows the capabilites of the language.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Research on parallel computing has a long tradition at RISC-Linz, especially the research on **languages for parallel symbolic computation**. This research has its roots in the seventies (see (Buchberger 1978)) and resulted in the concept of the parallel L-Machine and the associated L-Language (see (Buchberger 1985)). An experimental prototype of this machine and the language has been operative for several years now. One of the latest highlights in our research has been the design of the **new L-Language**, a parallel general purpose programming language designed to meet the needs of symbolic computation (see (Kusche 1988)).

For reasons of optimum hardware usage and highest possible efficiency, the new L-Language was decided to rely on **explicit coarse-grain parallelism**. The programmer himself has to specify the parallelism on the level of whole functions or even larger parts of the program.

In contrast to this approach, world-wide there has also been much research on how to exploit the **implicite fine-grain parallelism** inherent in most algorithms. Hardware architectures have been developed that allow to use this kind of parallelism in a very efficient way, where **dataflow machines** seem to be the most promising approach (see (Arvind and Iannucci 1987)). Especially for **functional languages**, which provide a very large amount of implicit fine-grained parallelism, these kinds of architectures are of great interest (see (Arvind *et al.* 1988)).

However, also for conventional machines the techniques used for exploiting implicit parallelism (e.g. dataflow analysis) are very important, since they allow the **automatic parallelization** of basically sequential programs (see (Boyle 1989)).

Therefore in the frame of the parallel programing project at RISC-Linz, we have started to investigate the usefulness of this approach for applications in symbolic computation: The goal of these investigations is the design and implementation of an *"Abstract Dataflow Machine"* (called ADAM) that will be programmed in a functional language and will simulate the architecture of a dataflow computer on a conventional parallel machine (see (Schreiner 1990)).

The extensive research on functional (respectively dataflow) programming has already resulted in a number of well-known functional languages (Miranda ((Turner 1985)), ML

((Harper *et al.* 1988)), ID ((Nikhil 1988)), ...) that correspond in many essential features. This shows that the functional programming community has reached a well-founded consensus on the state of the art. It is not our aim to enrich this variety of languages by a new language with extraordinary new facilities. In fact, any of the languages mentioned above would do the job of programming the ADAM at least as well as the language that we are going to propose. However, these languages have obtained a complexity that makes it rather difficult to implement an experimental prototype (of even a language subset) in a rapid way.

Since the prevailing purpose of our investigations is the research on dataflow machines and (for the moment) not the research on compiler construction, we have decided to design a small functional language providing the main features of the languages mentioned above while omitting those features which are not really essential but rather complicate an experimental implementation.

EvE[1] is the name of the language that we have designed to program the ADAM. The main intentions we pursue with the construction of this language are as follows:

1. We want to investigate the implementation of functional languages on parallel computers.
2. We want to experiment with symbolic algorithms programmed in a functional language and executed on parallel architectures.

EvE is small and simple, but it is certainly not a toy language. In particular, the language shows the following **characteristics**:

- EvE is *purely functional*, i.e. all functions are free of any kind of side-effects. Result values only depend on the values of the given arguments (and globally defined constants), but not on any other influences. The evaluation of any kind of expression (e.g. a function call) returns a value and has no effect besides that. This property, called *"Referential Transparency"* imposes the following restrictions on the language:
  - EvE has *no variables* (in the meaning of storage locations containing variable values) and therefore *no assignment statement* that can alter a variable's value.
  - Therefore, EvE also has no data structures whose components can be altered, especially it has *no mutable arrays.*
  - Since every kind of communication with the outside world is a side-effect, EvE has *no input/output statements*[2].
- EvE is *strongly typed.* It forces the programmer to give explicit type annotations to all objects defined and is therefore able to check the correct typing of all definitions and expressions. The type system of the language includes various structured types and allows the construction of recursively defined data types.

---

[1] "EvE" is a condensation of the term *"Evaluated Expressions"*, which is a hint on the functional character of the language.

[2] Of course, the system has to communicate with the human user by inputs and outputs, but this is the task of a dialogue shell and not part of the language itself.

- Functions are first-order objects, i.e. functions can receive other functions as parameters and return functions as results (*"Higher Order Functions"*).
- EvE has a special construct for the *delayed evaluation* of expressions and the construction of *infinite data structures ("Streams").*
- Functions and data structures are totally *non-strict*, i.e. functions may return results, although arguments have not yet been computed, and data structures may be used, although components of them are still unevaluated.
- EvE is a *safe language* in the sense, that programs never produce runtime errors which force a halt of the execution. Errors like "Division by Zero" are handled by the use of special *"Error Values"* which are returned as the results of invalid computations.

The advantages of the language for parallel execution are obvious:

- Any program written in EvE is fully *deterministic*, i.e. it always returns the same result, if it is given the same input. Runtime-dependent errors like deadlocks can not occur.
- EvE is implicitly parallel down to the lowest level. Any implementation of the language may vary the granularity of parallelism from fine-grain parallelism (on the level of primitive operators) up to coarse-grain parallelism (on the level of whole program blocks).
- The language is independent of the underlying hardware, the number of processors, and their configuration. Hardware changes will not cause any changes of the source code. Every program that correctly works on some kind of computer architecture will also do so on any other one.

In order to keep the language small, the following features have been explicitly omitted, either because they just represent less important *"Syntactic Sugar"* or because they make an experimental implementation essentially more complex[3]:

- Pattern Matching,
- Type Inference and Polymorphism,
- Lambda Expressions and Curried Functions,
- List Comprehensions,
- Abstract Data Types and Modules.

The **structure of the report** is as follow:

[3] Further versions of the language will probably include some of them.

- After this introduction, a **Reference Manual** describes the syntax and semantics of the language on the basis of small examples.
- Then a major **Programming Example** illustrates some of the most important features of the language.
- The appendix includes the formal **Syntax Definiton** of the language and a list of its **Predefined Identifiers** (i.e. types, constants and functions).

Chapter 2

Reference Manual

2.1 Hints for Further Reading

Before we start the description of the EvE language, some hints on notational conventions will help to understand the following sections:

Syntax Notation

The exact syntax rules of the language are given in Section A.3. For the moment, it suffices to distinguish two kinds of syntactic entities:

Terminal symbols, i.e. symbols that can appear in the source code of a program, are written in `Typewriter` type style. All program examples are written in this style, e.g.

```
F[x]: INT[INT] == x*x;
```

Nonterminal symbols, i.e. symbols that represent classes of tokens, but can not appear in a program text themselves, are written in *Italic* type style, e.g.

Identifier

Layout

EvE does not impose any restrictions upon the format of source texts (*"Free Format Style"*). Just the sequence of characters in the text is significant, indentations or other format features do not have any effects. As the only exception to this rule, line breaks denote the end of comments, which is described in the following subsection.

Comments

Comments start with the token `--` and continue to the end of the line (denoted by the ⟨*LF*⟩ character). Everything between this token and the end of the line is ignored. In fact, a line containing a comment is treated, as if it had ended before the `--`. Comments may not start within text constants (see Section 2.4.3).

```
-- This is a comment!
```

Identifiers

consist of a sequence of letters and digits, but have to start with at least one letter. EvE is case-sensitive, i.e. it distinguishes between lower and capital letters, therefore

`ident`, `Ident` and `IDENT`

denote three different identifiers. The end of an identifier is recognized by the first character that is neither a letter nor a digit, therefore two identifiers have to be separated by at least one such character.

Keywords and Predefined Identifiers

are written in capital letters, e.g.

```
IF NULL[l] THEN l ELSE m.
```

Keywords (`IF, THEN, ELSE`) are reserved and may not be redefined, while predefined identifiers (`NULL`) may[1]. A list of the keywords and predefined identifiers is given in Appendix A.5 and B.

2.2 Overall Program Structure

A program is a sequence of *definition statements* (also called *bindings*), which are separated by semicolons, i.e.

$$\begin{array}{c} \mathit{Definition}_1 \ ; \\ \mathit{Definition}_2 \ ; \\ \ldots \\ \mathit{Definition}_n \end{array}$$

[1] Of course, it is a very bad programming style to redefine predefined identifiers, therefore don't do it!

Each of these statements binds a name (an *identifier*) to some kind of object. The collection of bindings sets up the environment, in which a user-given expression is evaluated.

Typically, the file containing the program's source code is submitted to a compiler and translated into the machine code of the underlying hardware. Using an interactive dialogue shell, the user may enter expressions, which are evaluated within the environment generated by the compilation. The results of these evaluations are printed on the screen or written into an output file.

According to the kind of object an identifier is bound to, there are three kinds of definition statements:

- Type definitions,
- Constant definitions,
- Function definitions.

The order of definition statements within the source text is significant, the programmer has to obey two simple rules:

1. The definition of an identifier must textually precede its use: The scope of the identifier starts at the point of its definition and extends over the right hand sides of its own definition statement and all the following ones, as well as over the expression to be evaluated[2].
2. Within a level of scope[3], an identifier may be defined at most once, i.e. redefinitions are not allowed[4].

These rules require some programming discipline but are not really limitations of EvE. Their only purpose is to facilitate the construction of a compiler for the language.

2.3 Type Definitions

EvE is a *strongly typed* language, i.e. all operators and functions demand arguments of certain types. Typing errors are detected on compilation time and incorrectly typed programs are rejected. All types are *named*, i.e. they have to be introduced by a type definition statement of the form

$$Ident \texttt{ == } Type$$

where *Ident* is the name of the type to be introduced and *Type* its definition.

EvE knows four predefined types (see Table 2.1) and three constructors that build new types from already existing ones[5] (see Table 2.2).

Type Name	*Description*
`INT`	Integer numbers
`BOOL`	Boolean values
`CHAR`	Characters
`STR`	Strings

Table 2.1: Predefined Types

Type Constructor	*Description*
`LIST OF`	Sequence of elements of equal type
`TUPLE OF`	Fixed size collection of elements
`UNION OF`	Element of some selected type

Table 2.2: Type Constructors

A type definition may simply rename a previously defined type:

```
Number  ==  INT;
Word    ==  STR;
Letter  ==  CHAR
```

But it may also introduce a new type that is constructed from other types:

```
Sequence  ==  LIST OF INT;
Location  ==  TUPLE OF x: INT, y: INT, z: INT;
Datum     ==  UNION OF Num == INT, Word == STR
```

For strong type checking, it is important to know, in which cases two types are considered to be equal. As a rule, two types are equal, if and only if

1. Both types have the *same type name*, or if
2. One type is derived from the other by a sequence of *renamings*.

In the following example,

```
Number   ==  INT;
Integer  ==  INT
```

[2]As we will see, there are some relaxations to this rule in order to allow mutually recursive functions (see Section 2.10), data types (see Section 2.5.4) and constants (see Section 2.6).

[3]This term already indicates, that EvE knows local definitions (see Section 2.9).

[4]The possibility of redefining predefined identifers is not affected by this rule, since these identifiers are defined in an outer level of scope than the user definitions take place.

[5]A fourth type constructor, the *function constructor*, is described in Section 2.11.

the type `Number` equals `INT`, as well as `Integer` equals `INT`. But `Number` *does not equal* `Integer`, since both have different type names and no type is a renaming of the other one!

The scope of a user-defined type starts at the right hand side of the definition statement itself, therefore directly recursive type definitions like

```
Foo == TUPLE OF x: INT, y: Foo;
```

are principially allowed, but do not make sense. In contrast to directly recursive types, mutually recursive type definitions can be reasonable in some cases but violate the rule that types have to be defined before they can be used (see Section 2.2). EvE therefore provides a special kind of definition, the *forward type definition*. Its general form is simply

Ident

which makes public the name *Ident* of a new type but not its construction. From the point of this definition on, references to *Ident* are allowed, but the actual type definition has to take place in the same level of scope. So mutually recursive type definitions like

```
Subtree;
Tree      ==  UNION OF Node == INT, Inner == Subtree;
Subtree   ==  TUPLE OF value: INT, left: Tree, right: Tree
```

are possible. However, a sensible recursive type definition must contain at least one union type (see Section 2.5.4), but the compiler may not be able to check whether type definitions are reasonable or not[6].

In the following sections, we will provide a detailed description of the different kinds of types and type constructors.

2.4 Atomic Types

A type is called *atomic*, if objects of this type can not be partioned into smaller semantic entities. EvE knows three predefined atomic types,

- Integers (`INT`),
- Booleans (`BOOL`),
- Characters (`CHAR`).

---

[6] Even if objects of a senseless type are constructed, they can not cause any serious harm but are just not very useful.

2.4.1 Integers

The data type `INT` provides signed, fixed-size integer values in a range limited by the predefined integer constants

`MININT` and `MAXINT`

denoting the lower bound and the upper bound of the range, respectively. The values of these constants depend on the particular implementation but the assertion `MININT` $< 0 <$ `MAXINT` can be expected to hold[7].

Numeric expressions, i.e. expressions of type `INT`, look like common mathematical expressions, e.g

`a*(b+1)/c-d`.

EvE provides five binary integer operators and one unary operator (see Table 2.3) that obey the usual priority rules (the unary `-` binds stronger than `*`, `/`, and `%` that again bind stronger than `+` and the binary `-`).

Operator	*Function Name*	*Description*
`+`	`ADD`	Addition
`-`	`SUB`	Subtraction
`*`	`MUL`	Multiplication
`/`	`DIV`	Division
`%`	`MOD`	Remainder
`-`	`NEG`	Negation

Table 2.3: Integer Operators

The result of a remainder operation may be either `0` or it has the same sign as the result of the corresponding division, e.g.

`-17 % 5 = -2`.

The predefined function associated to each operator represents the prefix version of this operator[8], therefore the expression

`(-a)*(b-c)`

is equivalent to

`MUL[NEG[a],SUB[b,c]]`.

[7] Typically, these constant values equal -2^{32} and $2^{32}-1$ on 32-bit machines.

[8] The neccessity for the existence of these functions is explained in Section 2.11

The use of fixed-size integer numbers raises some problems, since integer operations may return values that are outside the range of type `INT` (*"Overflow"*) or not defined *("Zero Division")*. EvE handles these errors by the use of special *error values*, a detailed description of the error behaviour can be found in Section 2.13.

Two more predefined functions operating on integer values are listed in Table 2.4.

Function	*Description*
`ABS`	Absolute Value
`ODD`	Is number odd?

Table 2.4: Integer Functions

- `ABS[i]` returns the absolute value of the integer `i`. This function may return the error value, if `i` is negative and has no positive counterpart within the range of `INT` or vice versa.
- `ODD[i]` returns `TRUE`, if `i % 2 = 0` does not hold.

2.4.2 Booleans

The data type `BOOL` provides two predefined constants, the values

`TRUE` and `FALSE`.

For expressions of type `BOOL`, three operators (and associated functions) have been defined:

Operator	*Function Name*	*Description*
`&`	`AND`	Conjunction
|	`OR`	Disjunction
`~`	`NOT`	Negation

Table 2.5: Boolean Operators

In addition to the usual priority rules (`NOT (~)` binds stronger than `AND (&)` which binds stronger than `OR` (|)), the binary operators follow the *"Short Circuit"* discipline:

If a conjunction or disjunction is evaluated, the evaluation starts with the left operand. If this partial result already determines the result of the whole operation (i.e. if it is TRUE in disjunctions or FALSE in conjunctions) the second operand is not evaluated. We say:

`AND (&)` *and* `OR` (|) *are both strict in their first argument.*

The strictness of a function or data type can have a strong influence on the runtime behaviour and even on the termination of a program. Therefore we will discuss this important point in Section 2.12 in greater detail.

2.4.3 Characters

The data type `CHAR` provides the characters of the US-ASCII character set[9].

Character constants consist of a single character enclosed by single quotation marks ('). Valid characters between these marks are

- Any *printable character* of the ASCII character set except the backslash character (\) and the single quotation mark ('), e.g.

  `'a', 'b', 'Z', '␣', '*', '$'.`

- An *abbreviation* of the form `\l`, where `l` is a character code representing a special character (see Table 2.6), e.g.

  `'\n'` *(New Line)*, `'\''` *(' itself)*,

- A *code sequence* of the form *ddd*, where *ddd* is a three digit decimal ASCII code number from 000 to 255, e.g.

  `'\036'` *($)*, `'\065'` *(A)*, `'\127'` *(DEL)*.

Within a character constant, it is illegal to use the backslash character in another way than just described. A character constant (i.e. the sequence `'`*c*`'`, where *c* denotes a character) must not be splitted over several lines, therefore the $\langle LF \rangle$ character and the comment token `--` are not allowed to appear within it.

Special Character	*Description*	*ASCII code*
`\'`	' in character constants	044
`\"`	" in string constants	034
`\\`	\ itself	092
`\b`	Backspace	008
`\f`	Form Feed	009
`\n`	Line Feed	010
`\r`	Carriage Return	013
`\t`	Horizontal Tab	009
`\_`	String Interrupt	—

Table 2.6: Special Characters

EvE has three predefined functions handling the conversion between characters, from characters to integers, and vice versa (see Table 2.7).

- `CAP[ch]` returns the capital of `ch`, if `ch` is a letter, and `ch` itself, otherwise.

[9] A table of this character set can be found in Appendix A.2.

Function	*Description*
CAP	Capitalize a letter
CHR	Character from ASCII code
ORD	ASCII code of a character

Table 2.7: Character Functions

- CHR[int] returns the character with ASCII code int, if int is in the range 0–255, and the error value[10] otherwise.
- ORD[ch] returns the integer value denoting the ASCII code of character ch.

2.4.4 Comparison Operators

Table 2.8 shows the usual comparison operators that have been predefined on all three atomic types.

Operator	*Function*	*Description*
<	*LT	Less Than
<=	*LE	Less or Equal
=	*EQ	Equal
#	*NE	Not Equal
>=	*GE	Greater or Equal
>	*GT	Greater Than

Table 2.8: Comparison Operators

These operators are overloaded in the sense, that each of them is a placeholder for one of three functions, one function for each atomic type. The asterisk * in the function names of Table 2.8 has actually to be replaced by I (INT), B (BOOL), or C (CHAR) to get the particular comparison function for each data type, e.g.

```
ILT[2,5], BNE[TRUE,FALSE], CEQ['f','G'].
```

The comparison order for each type is defined as follows:

- INT: The comparison order of integer values is obvious.
- BOOL: Boolean values are ordered by FALSE < TRUE.
- CHAR: Characters are compared according to their ASCII values.

[10] For further explanation on error handling, see Section 2.13.

2.5 Structured Types

EvE knows three *type constructors* that generate new structured types from previously defined types,

- the *List Constructor* (`LIST OF`),
- the *Tuple Constructor* (`TUPLE OF`),
- the *Union Constructor* (`UNION OF`),

and one predefined structured type,

- the *String* type (`STR`).

2.5.1 Lists

Lists are variable-size sequences of elements of equal type. List types are defined by statements of the form

Ident `== LIST OF` *SimpleType*

where *SimpleType* denotes the name of a previously defined type.

Some examples of list types are

```
NumSeq    ==  LIST OF INT;
Word      ==  LIST OF CHAR;
Sentence  ==  LIST OF Word;
```

EvE does not allow nested type definitions, so statements like

```
Sentence == LIST OF LIST OF CHAR;
```

are invalid and have to be replaced by the introduction of auxiliary type definitions (as shown above).

Every list type possesses its own *constructor function*, which has the same name as its associated type and builds lists of this type. E.g. if we define

`NumSeq == LIST OF INT`,

then the expression

```
NumSeq[2,3,5,7,11,13,17]
```

constructs a list of type `NumSeq` which contains the first seven prime numbers. These elements are entered into the list from right to left, therefore 2 is the first element (the *"head"*) of the constructed list.

The empty number sequence (i.e. the list without any elements) is constructed by

`NumSeq[ ].`

The generic list operators and functions[11] that have been predefined are listed in Table 2.9.

Function	*Description*
`::`	Construction of a new list (infix)
`CONS`	Construction of a new list (prefix)
`HEAD`	Selection of the list head
`TAIL`	Selection of the list tail
`NULL`	Test a list for emptiness

Table 2.9: List Operators and Functions

- The operator `::` is the infix equivalence of the generic function `CONS`. It inserts its first argument of type *Any* into its second argument which is a list of type *AnyList*, where

  AnyList `== LIST OF` *Any*.

  E.g. both of the expressions

  ```
  2 :: NumSeq[3,5,7,11],
  CONS[2,NumSeq[3,5,7,11]]
  ```

  are equivalent to

  `NumSeq[2,3,5,7,11].`

- The components of a list can be extracted by the function `HEAD` and `TAIL` that return the first element of a list (its *"head"*) and the list without its first element (the *"tail"*). As a simplification, both of these selector functions (and only these!) may be applied to a list expression in the form

  List`.`*Selector*

  E.g., both of the expressions

[11] In this case, *"generic"* means *"operating on all kinds of lists"*.

```
HEAD[NumSeq[2,3,5,7]]
NumSeq[2,3,5,7].HEAD
```

return the value 2, and both of the expressions

```
TAIL[NumSeq[2,3,5,7]]
NumSeq[2,3,5,7].TAIL
```

return the list `NumSeq[3,5,7]`.

It is illegal to apply `HEAD` or `TAIL` to an empty list. How this error is handled, is described in Section 2.13.

- `NULL[list]` returns `TRUE`, if and only if `list` is empty, i.e. does not contain any element.

2.5.2 Strings

A *string* is simply a list of characters. Therefore, the predefined standard identifier `STR` can be defined by[12]

```
STR == LIST OF CHAR.
```

Because of this interpretation, all predefined list selectors and functions can also be applied on strings.

String constants consist of a (possibly empty) sequence of characters enclosed by double quotation marks ("), e.g.

```
"A string",
```

which is a simplified version of the list construction

```
STR['A',' ','s','t','r','i','n','g'].
```

The rules for valid characters within string constants are basically the same as for character constants, with two exceptions:

- *Double quotation marks* (") have to be encoded by the sequence `\"`, while single quotation marks stand for themselves. So the string constant

```
"He said \"I don't like programming\"."
```

[12] In spite of this fact, it is strongly recommended not to define ones own private string type by `WORD == LIST OF CHAR`, because any implementation may handle the type `STR` in a special way. It is a better strategy to rename the predefined type by `WORD == STR`, since renamings do not create a new type.

generates the string value

```
He said "I don't like programming".
```

- String constants may be *split over several lines* by the use of the special character `\_`. The first occurrence of this character within a string constant denotes an interruption of the string, after the next occurrence of `\_` (which may happen several lines afterwards) the string is continued. Between an string interruption and its continuation only white space (see Apendix A.2) and comments are allowed. E.g. the string constant

```
 "This is a stri\_  -- Break
\_ng interruption."
```

generates the string value

```
This is a string interruption.
```

2.5.3 Tuples

Tuples are fixed-size collections of elements with equal or different types (i.e. they represent the counterpart to the concept of *cartesian products* in denotational semantics). Tuple types are defined by statements of the form

$$\mathit{Ident}\ \texttt{==}\ \texttt{TUPLE OF}\ \mathit{Id}_1\ :\ \mathit{Type}_1\texttt{,}\ \mathit{Id}_2\ :\ \mathit{Type}_2\texttt{,}\ \ldots\ \mathit{Id}_n\ :\ \mathit{Type}_n$$

where Id_1 to Id_n are the *field names* of the tuple and Type_1 to Type_n their corresponding types, e.g.

```
Coordinate  ==  TUPLE OF x: INT, y: CHAR;
Name        ==  TUPLE OF first: Word, last: Word;
Address     ==  TUPLE OF street: Word, number: INT
```

Similar to lists, the components of a tuple must have explicit type names, anonymous type components, such as in

```
Name == TUPLE OF first: LIST OF CHAR, last: Word;
```

are not allowed.

Tuples are always constructed as a whole by the use of *constructor functions*. These functions have the same name as their associated types, e.g.

```
Coordinate[3,'A']
```

creates a new tuple with `x`-field set to 3 and `y`-field set to `'A'`.

The field names of a tuple are always local to the tuple, so definitions like

```
First == TUPLE OF x: INT,  y: INT;  -- Fields x and y
Secnd == TUPLE OF x: CHAR, y: CHAR; -- Fields x and y, too!
```

are perfectly valid.

Components of a tuple can be selected by use of the *selectors* that are associated to each tuple type and have the names of the fields to be selected.

The application of these selectors is (similar to lists) of the form

$$Tuple\texttt{.}Selector$$

E.g. if `name` is of type `Name` (as defined above), then the expressions

```
name.first,
name.last
```

return the first and the second component of the tuple, respectively.

2.5.4 Unions

Sometimes, the type of an object is not known at compile time, but there is only a fixed set of possibilites. In these cases, the *union constructor* may be used. A union type is rather a collection of types than a single type itself (it represents the equivalence to the concept of *disjoint unions* in denotational semantics). A *variant* of a union type is one of the types collected in the union. The definition of a union type is of the form

$$Ident \texttt{ == UNION OF } Id_1 \texttt{ == } Type_1\texttt{,}\ Id_2 \texttt{ == } Type_2\texttt{,}\ \ldots\ \texttt{,}\ Id_n \texttt{ == } Type_n$$

where Id_1 to Id_n are the variants of the union, i.e. renamings of the types $Type_1$ to $Type_n$. Although definitions of union types look similar to definitions of tuple types, it is important that one does not mix them up: In contrast to tuples, unions are not collections of constants but *collections of types*. Elements of a union type have as their actual type exactly *one variant* of the union.

The following statements are examples for the definition of tuple types:

```
Data        ==  UNION OF Num == INT, Word == STR;
Person      ==  UNION OF Man == Male, Woman == Female;
Coordinate  ==  UNION OF Pol == Polar, Cart == Cartesian;
```

An object of type `Data` may now be either an integer or a string, as well as a `Coordinate` value may be given either in polar or in cartesian form.

A union value is constructed by the application of a *conversion function* which converts an expression from a type that equals one variant of the union to the union type itself. This function has the name of its associated union type but is augmented by the name of the variant. Its general form is

UnionType`$`*Variant*`[`*expression*`]` .

where the $ separates the name of the union type from the name of the variant. So the expressions

```
Data$Num[317], Data$Word["Hello"]
```

create values that are both of type `Data`, but hold different variants of this type.

As with tuples, the variant names of a union type are local to the union and can therefore be used for definitions outside the union.

As already mentioned, the definition of a name must textually precede all of its applied occurrences. In order to alllow mutually recursive type definitions, EvE knows *forward type definitions* of the form

Ident

A forward type definition makes public only the name *Ident* of a type, but not its construction. From the point of this definition on, references to this type are allowed, but the actual type definition has to take place within the same level of scope. E.g. in the definition sequence

```
Subtree;
Tree      ==  UNION OF node == INT, inner == Subtree;
Subtree   ==  TUPLE OF leaf == INT, left == Tree, right == Tree;
```

the forward reference of `Tree` to `Subtree` is resolved by the use of a forward type definition.

Section 2.8 describes in detail how the actual variant of a union value can be determined and how a union value can be reconverted to its original type.

2.6 Constant Definitions

Constant definitions introduce names for anonymous values. Their general form is

Ident : *SimpleType* `==` *Expression*

Ident is the constant name to be introduced, *SimpleType* denotes the name of a previously defined type and *Expression* provides the value of the constant. The result value of *Expression* has to be of type *SimpleType*.

The following statements define some constant names:

```
i:  INT   ==  (2+3)*5;
s:  STR   ==  "The art of programming";
b:  BOOL  ==  (1 > 0) & ("test".HEAD = 't')
```

The scope of a constant name starts at the point of its definition, so the following chain of definitions is perfectly valid:

```
n:  INT  ==  -217;
m:  INT  ==  ABS(m);
a:  INT  ==  n+m;
```

Of course, constant definitions are not restricted to constants of atomic types, but every structured type also has to be given an explicit type name, e.g.

```
IntSeq      ==  LIST OF INT;
Pair        ==  TUPLE OF c: CHAR, i: IntSeq;

s:  IntSeq  ==  IntSeq[3,5,7,9];
p:  Pair    ==  Pair['A',2::s]
```

Since the scope of a constant also extends over the right hand side of its own definition, recursive constant definitions, such as

```
x: IntSeq == 1 :: x
```

are possible. This definition introduces a list `x` consisting of an infinite sequence of the value `1` . In order to allow mutually recursive constant definitions, EvE provides as a special form the *forward constant definition*. Its general form is

Ident : *SimpleType*

This definition makes public the name and the type of a new constant but not its value. From the point of this definition on, references to the constant are allowed. However, the actual constant definition has to take place within the same level of scope. Using this feature, mutually recursive constant definitions like

```
x2: IntSeq;
x1: IntSeq   ==  1 :: x2;
x2: IntSeq   ==  2 :: x1
```

are possible. These definitions introduce a list `x1` consisting of a infinite sequence of alternating values `1` and `2` starting with `1` and a list `x2` consisting of a infinite sequence of alternating values `2` and `1` starting with `2`.

A recursive constant definition only makes sense, if the expression on the right hand side of the definition contains *non-strict* operators or functions, e.g. the list constructor `CONS` (`::`). On the contrary, a definition like

```
undef: INT == undef+1
```

introduces a name `undef` for a value that will never be computed, since the strict operator `+` has to wait until all of its arguments have become available. For a detailed description of the strictness properties of the language, see Section 2.12.

Although the lists `x`, `x1` and `x2` defined above are logically inifinite, they can be represented in a finite (and actually very small) amount of computer memory. Altogether their construction only requires three applications of `CONS` (`::`), their infinite behaviour is a result of their (mutually) recursive definitions.

How "really infinite" list structures can be constructed and handled, will be shown in Section 2.14.

2.7 Conditional Expressions

A conditional expression is of the form

`IF` *Cond* `THEN` $Expr_1$ `ELSE` $Expr_2$

where *Cond* is a boolean expression, the *test condition*, and $Expr_1$ and $Expr_2$ are expression of equal type, e.g.

```
IF n = 0 THEN 1 ELSE n * Fib[n-1],
a + (IF a = 1 THEN b ELSE c) * d,
(IF 1 = 1 THEN 1 ELSE 0) > 0.
```

The if-then-else construct is *strict* in the first of its arguments, i.e. if its value is computed, first the boolean test condition is evaluated to `TRUE` or `FALSE`. Depending on the result, the expression of one branch of the construct *(but not the other one!)* is evaluated and returned as the result of the whole expression.

For a more detailed discussion on the strictness, respectively non-strictness of functions and data types, see Section 2.12.

2.8 Case Expressions

The case construct is used to select the actual type of a union-typed value (see Section 2.5.4) and to extract its actual value. A *case expression* is an expression of the form

`CASE`	*UnionExp*	
	$\{c_1 : Variant_1\}$	Exp_1
	$\{c_2 : Variant_2\}$	Exp_2
	...	
	$\{c_{n-1} : Variant_{n-1}\}$	Exp_{n-1}
	`ELSE`	Exp_n

where *UnionExp* is an expression that returns a union value, $Variant_1$ to $Variant_{n-1}$ represent different variants of this union and c_1 to c_n are identifiers used to define new constants. Exp_1 to Exp_n are expressions of equal type.

Each of the branches of the case construct tests the value returned by *UnionExp* for its actual variant (i.e. for the actual type of the union value). A variant must not be tested by different branches, i.e. $Variant_1$ to $Variant_{n-1}$ have to be distinct. If $Variant_i$ matches the actual variant of the union value, the identifier c_i is bound to the value that the variant holds, where c_i has type $Variant_i$. The scope of the constant name c_i is Exp_i, i.e. c_i may be freely used in Exp_i but is neither known in the other expressions Exp_j with $i \neq j$ nor outside the case expression. The expression Exp_i is evaluated and returned as the result of the whole case expression.

If no $Variant_i$ matches the actual variant of the union value, the expression Exp_n in the else branch of the construct is evaluated and returned as the result of the case expression. If the variants tested by the case construct are *exhaustive*, i.e. if each of the variants of the union type is tested by one branch, the else branch of the construct has to be omitted.

E.g., using the definitions

```
-- Types
Data == UNION OF Num == INT, Letter = CHAR, Word == STR;
Node == UNION OF Flag == BOOL, Value == INT;

-- Constants
d: Data == Data$Word["Foo"];
n: Node == Node$Value[317];
```

the expression

```
CASE d
  {n: Num}    n*2
  {l: Letter} ORD[l]+1
  ELSE        -1
```

returns the value `-1` and the expression

```
CASE n
 {f: Flag}  ~f
 {v: Value} v=0
```

returns the value `FALSE`. The second example shows a case expression with exhaustive variants, i.e. each of the variants of type `Node` is handled by one branch, therefore the else part has to be omitted.

There is one point that might be misleading: Every identifier c_i defined by

$$c_i \ : \ Variant_i$$

is of type $Variant_i$ which is a renaming of type $SimpleType_i$ in the definition of the underlying union type:

$$UnionType \texttt{ == UNION OF } \ldots\ Variant_i \texttt{ == } SimpleType_i \ldots$$

Therefore in the first case expression shown above, `n` is a constant of type `Num`. However it may be freely used in integer expressions, since `Num` is merely a renaming of `INT`.

The case construct is strict in its first argument, i.e. in the expression *UnionExp*. Depending on the actual variant of the union value, one of the expressions Exp_i (*but not the other ones!*) is evaluated and returned as the result of the whole expression.

A more detialed discussion on the strictness properties of EvE can be found in Section 2.12.

2.9 Block Expressions

A *block* is an expression of the form[13]

$$\begin{array}{l} \{ \\ \qquad Definition_1\ ; \\ \qquad Definition_2\ ; \\ \qquad \quad \ldots \\ \qquad Definition_n \\ \} \\ \qquad Expression \end{array}$$

The value of a block is the value of the trailing expression, which is evaluated in the global environment augmented by the bindings within the block, e.g.

```
{
   a:INT == 3;
   b:INT == a+2;
}
   (a + b) * 7
```

evaluates to 56.

Every definition statement binds an identifier to a value and augments the environment by this new binding. The scope of the binding starts at the point of the definition statement and extends over the right hand side of this definition and all the following ones (up to the end of the block), as well as over the expression to be evaluated. Any binding to an identifier `x` outside the block is overridden by a redefinition of `x` within the block. The other way round, the binding of an identifer within a block is invisible outside the block.

Blocks may be nested and follow the *lexical scoping* discipline. The value of an identifier that has not been defined within a block is taken from the surrounding block. If it has

[13]The curly braces {} replace the more known `LET IN` construct of other functional languages.

not been introduced there, the value is taken from the block on the next outer level of nestings, and so on, up to the global level of definition statements.

In contrast to many other functional languages, in EvE the scope of an identifier starts at the point of its definition, previous occurrences of the identifier take their values from the surrounding environment. E.g. the nested block expression

```
{   a:INT == 3   }
{
    b:INT == a;
    a:INT == 5
}
    a + b
```

evaluates to 8 and not to 10, since the reference to `a` within the definition of `b` takes its value from the binding of the surrounding block.

2.10 Function Definitions

Functions are defined by a defintion statement of the form

Ident`[`*Ident*`,` ...`]` : *SimpleType*`[`*SimpleType*`,` ...`]` `==` *Expression*

The first *Ident* is the name of the function to be defined, the *Idents* within the brackets define the names of the formal parameters. The first *SimpleType* denotes the type of the function's result value, the following ones the types of the parameters in the order of their appeareance within the brackets. The *Expression* specifies the result value of the function.

There is also a second form of function definitions for functions with *named type signature*:

Ident`[`*Ident*`,` ...`]` : *SimpleType* `==` *Expression*

In this case, *SimpleType* denotes the name of a *function type* that specifies the types of the parameters as well as the type of the result. For further explanation on function types, see Section 2.11.

The following statements present some of the not very exciting but inevitable examples of function definitions how they can be found in any textbook on functional programming:

```
Fac[n]: INT[INT] ==  -- Factorial
  IF n = 0
    THEN 1
    ELSE n * Fac[n-1];

Fib[n]: INT[INT] ==  -- Fibonacci, Sledge-Hammer Version
  IF n <= 2
```

```
     THEN 1
     ELSE {
           a: INT == Fib[n-1];
           b: INT == Fib[n-2]
          }
           a + b;

Fibo[n]: INT[INT] == -- Fibonacci, Tail Recursive Version
  {
   -- Local Auxiliary Function
   Iter[a,b,n]: INT[INT,INT,INT] ==
     IF n = 1
      THEN a
      ELSE Iter[b,a+b,n-1];
  }
   Iter[1,1,n];
```

As shown in these examples, the function *"bodies"* (i.e. the expressions on the right hand sides of function definitions) may include local definitions, also function definitions themselves. The formal parameters of a function can be interpreted as constant names that are locally defined, which overrides any binding of these names outside the function.

As usual, function definitions follow the lexical scoping discipline (see Section 2.9). Names that are used within the body of a function, but have neither been introduced as parameters nor by local bindings, take their values from the lexical enclosing environment. E.g. in the definition statements

```
a: INT == 7;

F[m,n]: INT[INT,INT] ==
 {
  G[i,j]: INT[INT,INT] == i*j+n
 }
  G[m,a]
```

the reference to `a` within the body of `F` takes its value 7 from the global environment, while the reference to `n` within the body of `G` takes its value from the formal parameter of `F`.

As already mentioned several times, the scope of any name starts at the point of its definition, therefore functions have to be defined, before they can be used. In order to allow mutually recursive functions, EvE knows as a special form the *forward function definition*:

Ident`[`*Ident*`, ...] :` *SimpleType*`[`*SimpleType*`, ...]`

respectively

Ident[*Ident*, ...] : *SimpleType*

for functions with named type signature (see Section 2.11).

A forward definition statement makes public just the signature of a function, i.e. its head and its type, but not the body of the function itself. From the point of this statement on, references to this function are allowed but the actual function definition has to take place within the same level of scope.

E.g. in the definition sequence

```
G[x]: INT[INT];    -- Forward Definition of G

F[x]: INT[INT] ==  -- Definition of F
  IF ODD[x]
    THEN F[x-1]
    ELSE G[x/2];   -- Use of G

G[x]: INT[INT] ==  -- Actual Definition of G
  IF x = 0
    THEN 1
    ELSE F[x+2];   -- Use of F
```

the forward reference to `G` is resolved by the use of a forward definition statement.

2.11 Higher Order Functions

In EvE, functions are first order citizens like any other types of objects, i.e. they can be stored in data structures, passed to other functions as arguments, returned as results of functions, etc.

For this purpose, EvE provides as a fourth type constructor[14], the *function constructor*. Its general form is

SimpleType[*SimpleType*, ..., *SimpleType*]

This constructor creates the type that includes all those functions, whose signatures (i.e. the number and the types of parameters as well as the result type) equal the given signature (this is known as *function domain* in denotational semantics). Examples for the definition of function types are

```
IntFun   ==  INT[INT];  -- Unary functions on integers, e.g. INEG
IntTest  ==  BOOL[INT]; -- Test functions on integers, e.g. ODD
CharInt  ==  CHAR[INT]; -- Conversion functions, e.g. CHR
```

[14]The three other ones have been described in Section 2.3.

The function constructor enables us to write higher order functions, i.e. functions that take other functions as arguments, e.g.

```
IntFun  == INT[INT];
IntList == LIST OF INT;

IMap[fun,list]: IntList[IntFun,IntList] ==
  IF NULL[list]
    THEN list
    ELSE fun[list.HEAD]::IMap[fun,list.TAIL];
```

This function `IMap` takes a function and a list as arguments and applies the function to every element of the list (it "maps the function to the list"). The generated result values are successively entered into a list, which is returned as the result of the function.

Using the following function definition with a named type signature,

```
Twice[x]: IntFun == x+x
```

the expression

```
IMap[Twice,IntList[1,2,3,4,5]]
```

maps the function `Twice` to a list of integers and returns the list of the doubled values. But now, if we want to use `IMap` to apply the function

```
Square[x]: INT[INT] == x*x
```

to a list of integers, we have to take care. In contrast to our policy of giving all objects explicitely *named types*, `Square` has, like most functions, an *anonymous type* specified by its signature[15]. Our definition of type equality (see Section 2.3) forces us to convert a function with anonymous type signature to a named type signature. This may be done by the application of a *conversion function* that has the name of its associated type. E.g. the expression

```
IMap[IntFun[Square],IntList[1,2,3,4,5]]
```

maps the converted function `Square` to a list of integers and returns the list of the squared values.

Functions can also be generated as results of function calls, e.g. the statement

```
Mult[n]: IntFun[INT] ==
  {M[i]: INT[INT] == n*i}
  IntFun[M];
```

[15]The reason for this is mere practicality, since it would be very tedious to define explicit type names for every kind of function signature.

defines a function `Mult`, which takes an integer `n` as argument and returns a unary function, which multiplies a given integer `i` with `n`.

Functions can also be renamed or stored in data structures, e.g. the statements

```
FunPair == TUPLE OF a: IntFun, b: IntFun;

A: IntFun == IntFun[IADD]; -- Renaming of IADD
B: FunPair ==              -- Pair of ISUB and IDIV
  FunPair[IntFun[ISUB],
          IntFun[IDIV]];
```

define the name `A` as a synonym for `IADD`[16], and `B` as a tuple containing the functions `ISUB` and `IDIV`.

Now the existence of the prefix versions of the integer operators and comparison operators becomes clear. Their only purpose is to serve as the first order counterparts of the infix operators when using higher order functions.

2.12 Strictness

The strictness properties of functions and data structures may have strong impacts on the runtime behaviour of a program, especially on its termination and on its reaction on runtime errors.

In this section, we use the special value $\perp$ (in words *"bottom"*) for two purposes:

1. It denotes a non-terminating value, i.e. an expression, whose result value will never be computed.
2. It denotes the error value, i.e. the result of the illegal application of an operator[17].

We say that a function $fun[arg_1, arg_2, ..., arg_n]$ is *strict in* arg_i $(1 \leq i \leq n)$, if the following property holds:

$$fun[arg_1, ..., arg_{i-1}, \perp, arg_{i+1}, ..., arg_n] \; = \perp .$$

Otherwise we call fun *non-strict in* arg_i.

In other words, a function fun is strict in an argument arg_i, if

- whenever the evaluation of arg_i does not terminate and return a result, fun also does not terminate and return a result, and

[16] The term "synonym" already indicates that any reference to `A` is actually a reference to `IADD`. Function renamings do *not* create new functions!

[17] See Section 2.13 for more details.

- whenever the evaluation of arg_i results in the error value, fun also results in the error value[18].

In EvE there exist a few simple rules for the strictness of functions:

- User-defined functions are *non-strict* in all of their arguments, unless the strictness in an argument is forced by the body of the function. This is the case, if the evaluation of the body requires calls of operators or functions that are strict in this argument.
- Predefined functions and operators are *strict* in all of their arguments. This rule may be overridden by some of the following rules.
- The boolean operators & and | (and their associated functions `AND` and `OR`) are only *strict in their first argument.*
- The infix list construction operator `::` and its prefix version `CONS` are *non-strict* in both of their arguments.
- All list, tuple, and union constructor functions are *non-strict* in all of their arguments.
- The if-then-else construct is only *strict in its first argument*, i.e. the boolean test condition.
- The case construct is only *strict in its first argument*, i.e. the union expression.

Some examples may help to understand these rules: we define the non-terminating function

```
Forever[x]: INT[INT] == Forever[x]
```

and the following functions

```
AddO[x]: INT[INT] == 1+0;   -- Not strict in x

AddX[x]: INT[INT] == 1+0*x; -- Strict in x

Branch[x]: INT[INT] ==      -- Strict in x
  IF x=0                       -- = is strict in x
    THEN 1                     -- THEN-branch not strict
    ELSE Forever[x];           -- ELSE-branch strict in x

ConsL[x]: IntList[INT] ==   -- Not strict in x
  x :: IntList[0];             -- :: not strict in first arg

ConsR[x]: IntList[INT] ==   -- Not strict in x
  1 :: IntList[x];             -- :: not strict in secnd arg
```

[18] A detailed description of error handling can be found in Section 2.13.

Then the following propositions hold:

$$
\begin{array}{lcl}
\texttt{Add0[Forever[0]]} & = & \texttt{1} \\
\texttt{AddX[Forever[0]]} & = & \perp \\
\texttt{Branch[0]} & = & \texttt{1} \\
\texttt{Branch[Forever[0]]} & = & \perp \\
\texttt{ConsL[Forever[0]]} & = & \texttt{IntList[}\perp\texttt{,0]} \\
\texttt{ConsR[Forever[0]]} & = & \texttt{IntList[1,}\perp\texttt{]}
\end{array}
$$

where $\perp$ denotes a value that will never be computed.

But be careful, although the expression

```
ConsL[Forever[0]]
```

returns a result (namely the list with the first value undefined and the second value 0), the expression

```
ConsL[Forever[0]].HEAD
```

will not return a value, since the head of the constructed list will never be computed!

In EvE, all *compound data structures* (lists, tuples and also unions) are *non-strict* in the sense, that the structure as a whole may be used, while components of the structure may still be undefined.

The consequences of the strictness property on the error behaviour of a program will be discussed in the next section.

2.13 Error Behaviour

EvE is a language especially designed for the execution on parallel computers. In this context, the treatment of runtime errors (like zero division) is given special importance, since it is not clear what happens if one processor triggers an error, while others may continue working.

EvE handles runtime errors by the use of *error values*. Whenever some operation is not able to return a proper result, instead of the result the predefined constant `ERROR` is returned. `ERROR` is a generic constant, i.e. it is element of every data type that can be defined in EvE and therefore compatible with every type.

`ERROR` may only be produced in those cases that are listed in Table 2.10.

If a function or operator is applied to `ERROR`, the result of the operation depends on the strictness, respectively non-strictness, of the function[19]:

- If the function is strict in the `ERROR` argument, the result will also be the `ERROR` value.

[19] See Section 2.12 for a detailed discussion on this topic.

Operator	*Error*	*Description*
`+, -, *, /, ABS`	Overflow	Result outside the range `MININT–MAXINT`
`/, %`	Zero divide	Division (resp. Remainder) by zero
`CHR`	No ASCII value	Argument outside the range 0–255
`HEAD, TAIL`	Empty list	Application to an empty list

Table 2.10: Runtime Errors

- If the function is not strict in this argument, it will compute some sensible result value.

The generic function `FAILED` is used to detect, whether the evaluation of some value value has been successful or whether the result is the `ERROR` constant, i.e.

```
FAILED[ERROR] = TRUE.
```

The following examples demonstrate some cases, in which `ERROR` may be returned as result:

```
FAILED[1/0]                          =   TRUE
FAILED[IntList[1/0]]                 =   FALSE
FAILED[CHR[1234]]                    =   TRUE
FAILED[IF 0=0 THEN 1 ELSE 1/0]       =   FALSE
FAILED[IF 1=0 THEN 1 ELSE ERROR]     =   TRUE
FAILED[(1/0)::IntList[1]]            =   FALSE
```

Take care, although `1/0` is the error value, `IntList[1/0]` returns a proper result, namely a list whose only element is the error value!

User-defined functions are basically non-strict, i.e. they may return proper results, although one of their arguments is invalid, e.g. using the definition

```
F[x,y]: INT[INT,INT] ==
  IF x=0
    THEN 1
    ELSE y;
```

the expression `F[0,1/0]` returns the result value 1, although the second argument to the function is the `ERROR` value.

2.14 Delayed Evaluation

In some situations, it is desirable not to start the evaluation of an expression immediately since one is not sure, whether this value will be really required. In this case it is better to *delay the evaluation* until the value of the expression is really (if at all) needed. E.g. consider the function

```
Infinite[x]: IntList[INT] == x::Infinite[x+1]
```

constructing the infinite[20] list of integers starting with the value x. Using this kind of list, we can solve a big number of problems in a very intuitive and elegant fashion. E.g. taking the function `IMap` as defined in Section 2.11 and the function `Square` computing the square of an integer number, the expression

```
IMap[Infinite[1], Square]
```

generates a table of all square numbers.

Of course, it makes no sense to start the computation of an infinite list eagerly, as this would flood the computer's processor and its memory with a lot of useless work and results. Actually only a small amount of these results will be needed, so it is be reasonable to delay the computation of the list elements, until they are really required.

For this purpose, EvE provides the generic function `DELAY`, which takes an expression as argument and returns an unevaluated representation of the expression. So the definition above would change to

```
Infinite[x]: IntList[INT] == DELAY[x::Infinite[x+1]].
```

It is not explicitly neccessary to force the evaluation of a delayed expression. As soon as its value is required[21], the evaluation is triggered *automatically*, without any assistance of the programmer. To him, `DELAY` behaves just like a pseudo-function defined by

`DELAY[x]:` *AnyType*`[`*AnyType*`] == x.`

Therefore, any application of `DELAY` may be omitted without any change of a program's semantics. The real difference will become obvious at runtime: Without `DELAY`, processor and memory load will dramatically increase, and the computer will possibly fail to generate the result.

Using delayed evaluation, the recursive constant definitions mentioned in Section 2.6 may become very useful. E.g. the statement sequence

```
Twice[list]: IntSeq[IntSeq] == -- Double a list of integers
  DELAY[ 2*list.HEAD :: Twice[list.TAIL] ];

exp: IntSeq ==                 -- List of all powers of 2
  DELAY[1::Twice[exp]];

PowerOf2[n]: INT[INT] ==       -- Compute the n-th power of 2
```

[20] In contrast to the recursively defined constants of Section 2.6, a list generated by this function is "really" infinite, since it requires an unbounded number of constructor operations to be computed.

[21] This is the case, if the expression is operand of a function that is strict in this argument (a detailed discussion on the strictness of functions can be found in Section 2.12).

```
{
Nth[l,n]: INT[IntSeq,INT] ==
  IF n=0
    THEN l.head
    ELSE Nth[l.TAIL,n-1]
}
Nth[exp,n];
```

constructs the function `Powerof2` that returns the n-th power of 2 by the use of the recursively defined infinite list `exp`. `exp` serves as a table containing all powers of 2, in which the required value is looked up by the local function `Nth`. Every element of `exp` is computed at most once, further references to an element are handled simply by looking up the table. This technique of tabulating the result values of functions (sometimes called "memoization") helps to avoid redundant computations and in many cases essentially increases the performance of a program (see (Abelson and Sussman 1985)).

Chapter 3

A Programming Example

As an illustration of some of the facilities of EvE, we show the design of a function that performs symbolic differentiation of algebraic expressions. This example is taken from (Abelson and Sussman 1985), where the program has been written in Scheme, a functional dialect of Lisp.

We want to write a function `Derive` that takes as arguments an algebraic expression and a variable and returns the derivative of the expression with respect to the variable. The algorithm itself is rather simple, but we want to direct our attention to the way EvE supports the solution of the problem.

First of all, how can we represent algebraic expressions? Algebraic expressions may be constants or variables, as well as sums or products of previously constructed expressions[1]. This description already gives us some hints on how to define expressions:

- Constants may be simply encoded as integers.
- Variables may be represented by character strings.
- Sums and Products are pairs, i.e. tuples, of expressions.
- Since an expression may be either a constant, a variable, a sum or a product, we will represent it as a union consisting of one of these four variants.

Figure 3.1 shows the several statements needed for these type definitions.

Let's take a look at the order of these statements: `SumExp` and `ProdExp` both refer to `Exp`, therefore `Exp` has to made public by a forward type definition (see Section 2.3). Vice versa, `Exp` refers to `SumExp` and `ProdExp` too (mutual recursion), but both of these types have been introduced before.

Now we have to introduce some functions doing auxiliary work in the process of computing the derivative of an expression.

[1]We restrict ourselves to these kinds of expressions. It is rather simple to extend them to expressions that also contain subtractions, divisions, etc.

```
----------------------------------------------------------
--
-- Symbolic Differentiation (Abelson & Sussman, page 104)
--
----------------------------------------------------------

Constant == INT;       -- Constants

Variable == STR;       -- Variables

Exp;                   -- Forward Definition Exp

SumExp == TUPLE OF     -- Sum Expressions
  addend: Exp,         -- addend + augend
  augend: Exp;

ProdExp == TUPLE OF    -- Product Expressions
  multiplicand: Exp,   -- multiplcand * multiplier
  multiplier:   Exp;

Exp == UNION OF        -- Expressions
  Const == Constant,
  Var   == Variable,
  Sum   == SumExp,
  Prod  == ProdExp;
```

Figure 3.1: Symbolic Derivation (Type Definitions)

- First, we need a function `VarEq` that compares two variables. This task is not very difficult, since variables are represented by character strings, which can be compared by the sequential comparison of their elements.
- Then we need two functions `MakeSum` and `MakeProd` constructing the sum and the product of two expressions. This task gets a little bit complicated, because we want these functions to perform some simplifications. E.g. the expression $0 + x$ should not remain a sum expression, but be simplified to x itself. Similar simplifications can be done for $0 * x$ and $1 * x$. In addition, if both parts of a sum or a product are constants, the result value should be computed directly without the construction of a compound expression.

Figure 3.2 shows, how these functions can be programmed.

An interesting point is the definition of the function `VarEq`. `VarEq` does not call the function `StrEq` that performs the actual comparison of strings. On the contrary, `VarEq` *is* the function `StrEq`, or in other words, it is a synonym of this function, so any reference to `VarEq` is actually a reference to `StrEq`. Renamings of functions are (like renamings of types) a very powerful tool for program and data abstraction, since any changes in the representation of an object will affect just some small parts of the program.

```
-- Test Strings for Equality
StrEq[s1,s2]: BOOL[STR,STR] ==
  IF NULL[s1]
    THEN NULL[s2]
  ELSE IF NULL[s2]
    THEN NULL[s1]
  ELSE
    (s1.HEAD = s2.HEAD) & StrEq[s1.TAIL,s2.TAIL];

-- Test Variables for Equality
BoolVar == BOOL[Variable,Variable];
VarEq: BoolVar == BoolVar[StrEq];

-- Construct the Sum of Two Expressions
MakeSum[exp1,exp2]: Exp[Exp,Exp] ==
  CASE exp1
    {c1: Const} CASE exp2
      {c2: Const} Exp$Const[c1+c2]   -- exp1 and exp2 constant
      ELSE IF c1 = 0                 -- only exp1 constant
        THEN exp2                    --   exp1 = 0
        ELSE Exp$Sum[exp1,exp2]      --   exp1 # 0
    ELSE CASE exp2
      {c2: Const} IF c2 = 0          -- only exp2 constant
        THEN exp1                    --   exp2 = 0
        ELSE Exp$Sum[exp1,exp2]      --   exp2 # 0
      ELSE Exp$Sum[exp1,exp2];       -- No constants

-- Construct the Product of Two Expresssions
MakeProd[exp1,exp2]: Exp[Exp,Exp] ==
  CASE exp1
    {c1: Const} CASE exp2
      {c2: Const} Exp$Const[c1,c2]   -- exp1 and exp2 constant
      ELSE IF c1 = 0 THEN
        Exp$Const[0]                 -- exp1 = 0
      ELSE IF c1 = 1 THEN
        exp2                         -- exp1 = 1
      ELSE
        Exp$Prod[exp1,exp2]          -- exp1 # 0,1
    ELSE CASE exp2
      {c2: Const} IF c2 = 0 THEN
        Exp$Const[0]                 -- exp2 = 0
      ELSE IF c2 = 1 THEN
        exp1                         -- exp2 = 1
      ELSE
        Exp$Prod[exp1,exp2]          -- exp2 # 0,1
      ELSE
        Exp$Prod[exp1,exp2];         -- No constants
```

Figure 3.2: Symbolic Derivation (Auxiliary Functions)

The actual derivation function, shown in figure 3.3, can now be written straightforward. Depending on the type of the expression, we perform one of the well-known differentiation rules for constants, variables, sums and products.

```
--------------------------------------------------------------
-- Derive an expressions exp with respect to the variable var
--------------------------------------------------------------

Derive[exp,var]: Exp[Exp,Var] ==

  CASE exp

    {c: Const}             -- Constants
      Exp$Const[0]

    {v: Var}               -- Variables
      IF VarEq[v,var]
        THEN Exp$Const[1]
        ELSE Exp$Const[0]

    {s: Sum}               -- Sums
     {
      first: Exp == Derive[s.addend,var];
      secnd: Exp == Derive[s.augend,var]
     }
      MakeSum[first,secnd]

    {p: Prod}              -- Products
     {
      first: Exp == p.multiplicand;
      secnd: Exp == p.multiplier;
      prod1: Exp == MakeProd[first,Derive[secnd,var]];
      prod2: Exp == MakeProd[Derive[first,var],secnd]
     }
      MakeSum[prod1,prod2];
```

Figure 3.3: Symbolic Derivation (Main Function)

Our derivation program is still very simple with respect to the simplification of algebraic expressions. This is a very complex algorithmic problem, because, among other reasons, a form that is simplest for one purpose may not be for another. Therefore in “real” applications, combinations of different strategies have to be applied.

Appendix A

Syntax

A.1 Notation

We define EvE's lexical structure and syntax using an *Extended Backus-Naur Notation*:

Terminal Symbols (i.e. the tokens of EvE) are written in `Typewriter` style.

Nonterminal Symbols (i.e. syntactic entities defined by syntax rules) appear in *Italic* style.

Meta symbols (i.e. symbols that structure syntax rules) appear in simple Roman style:

- = seperates a nonterminal symbol that is being defined from its definition.
- ( ) group syntactic items in order to form a larger item[1].
- [ ] enclose syntactic items that are optional, i.e. that may appear once or not at all in that place[2].
- { } enclose syntactic items that may be repeated zero or more times in that place[3].
- | seperates two alternative syntactic items, one of which must appear in that place.
- . denotes the end of a syntax rule.

Each nonterminal symbol is defined by a **Syntax Rule** of the form

$$\textit{Nonterminal} = \textit{Definition}\ .$$

[1] Do not mix up the meta symbols ( ) with the symbols `(` `)` of the EvE language!
[2] Do not mix up the meta symbols [ ] with the symbols `[` `]` of the EvE language!
[3] Do not mix up the meta symbols { } with the symbols of the EvE language!

	0	*16*	*32*	*48*	*64*	*80*	*96*	*112*
0	⟨*NUL*⟩	⟨*DLE*⟩	␣	0	@	P	`	p
1	⟨*SOH*⟩	⟨*DC1*⟩	!	1	A	Q	a	q
2	⟨*STX*⟩	⟨*DC2*⟩	"	2	B	R	b	r
3	⟨*ETX*⟩	⟨*DC3*⟩	#	3	C	S	c	s
4	⟨*EOT*⟩	⟨*DC4*⟩	$	4	D	T	d	t
5	⟨*ENQ*⟩	⟨*NAK*⟩	%	5	E	U	e	u
6	⟨*ACK*⟩	⟨*SYN*⟩	&	6	F	V	f	v
7	⟨*BEL*⟩	⟨*ETB*⟩	'	7	G	W	g	w
8	⟨*BS*⟩	⟨*CAN*⟩	(	8	H	X	h	x
9	⟨*HT*⟩	⟨*EM*⟩	)	9	I	Y	i	y
10	⟨*LF*⟩	⟨*SUB*⟩	*	:	J	Z	j	z
11	⟨*VT*⟩	⟨*ESC*⟩	+	;	K	[	k	{
12	⟨*FF*⟩	⟨*FS*⟩	,	<	L	\	l	|
13	⟨*CR*⟩	⟨*GS*⟩	-	=	M	]	m	}
14	⟨*SO*⟩	⟨*RS*⟩	.	>	N	^	n	~
15	⟨*SI*⟩	⟨*US*⟩	/	?	O	_	o	⟨*DEL*⟩

Table A.1: The US-ASCII Character Set

A.2 Input File

Input files may consist of all printable characters of the *US-ASCII character set* (see Table A.1). In addition, the following invisible ASCII characters *(white space)* are allowed:

- ␣ (ASCII code 032) is the *space character*.
- ⟨*TAB*⟩ (ASCII code 009) is translated to a *single space* in all places it occurs (even in text constants).
- ⟨*LF*⟩ (ASCII code 010) denotes the *end of a line*.
- ⟨*FF*⟩ (ASCII code 012) is translated into the ⟨*LF*⟩ character in all places it occurs (also in text constants).
- ⟨*CR*⟩ (ASCII code 013) is translated into the ⟨*LF*⟩ character in all places it occurs (also in text constants).

All other characters are *illegal* (even within text constants or comments).

A.3 Lexical Structure

The following syntax rules define the lexical structure, i.e. the structure of the tokens of EvE:

Letter	=	`a` | `b` | ... | `z` | `A` | `B` | ... | `Z` .
Digit	=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` .
Ident	=	*Letter* { *Letter* | *Digit* }.
Integer	=	*Digit* { *Digit* }.
Number	=	*Integer* .
SpecialChar	=	*Any character of Table 2.6 except* `\'` *and* `\"` .
SingleChar	=	*Any printable character of the ASCII set including* ␣ *except* `'` *and* `\` | *SpecialChar* | `"` | `\'` .
StringChar	=	*Any printable character of the ASCII set including* ␣ *except* `"` *and* `\` | *SpecialChar* | `'` | `\"` .
Character	=	`'` SingleChar `'` .
String	=	`"` { *StringChar* } `"` .

A.4 Context-Free Syntax

The following syntax rules define the context-free syntax of the EvE language in four sections:

- Types,
- Definitions,
- Expressions,
- Programs.

Types

IdentList	=	*Ident* { `,` *Ident* } .
SimpleType	=	*Ident* .
TypeList	=	*SimpleType* { `,` *SimpleType* } .
Field	=	*Ident* `:` *SimpleType* .
FieldList	=	*Field* { `,` *Field* } .
Variant	=	*Ident* `==` *SimpleType* .
VariantList	=	*Variant* { `,` *Variant* } .
ListType	=	`LIST OF` *SimpleType* .
TupleType	=	`TUPLE OF` *FieldList* .
UnionType	=	`UNION OF` *VariantList* .
FunctionType	=	*SimpleType* `[` *TypeList* `]` .
Type	=	*SimpleType* | *ListType* | *TupleType* | *UnionType* | *FunctionType* .

Definition Statements

TypeDefinition = *Ident* [`==` *Type*].
ConstantDefinition = *Ident* `:` *SimpleType* [`==` *Expression*].
FunctionDefinition = *Ident* `[` *IdentList* `]` `:` (*FunctionType* | *SimpleType*) [`==` *Expression*].
Definition = [*TypeDefinition* | *ConstantDefinition* | *FunctionDefinition*].
DefinitionSequence = *Definition* { `;` *Definition* }.

Expressions

Expression = *ListExpr* | `{` *DefinitionSequence* `}` *Expression* | `IF` *Expression* `THEN` *Expression* `ELSE` *Expression* | `CASE` *Expression* `{` *Field* `}` *Expression* { `{` *Field* `}` *Expression* } [`ELSE` *Expression*].
ListExpr = *CompoundExpr* { `::` *Factor* }.
CompoundExpr = *SimpleExpr* [*Relation* *SimpleExpr*].
Relation = `=` | `#` | `<` | `<=` | `>` | `>=` .
SimpleExpr = [`+` | `-`] *Term* { *AddOp* *Term* }.
AddOp = `+` | `-` | `|` .
Term = *Factor* { *MulOp* *Factor* }.
MulOp = `*` | `/` | `%` | `&` .
Factor = *Number* | *Character* | *String* | *Designator* { *ArgumentList* | *Selector* } | `(` *Expression* `)` | `~` *Factor* .
Designator = *Ident* [`$` *Ident*].
ArgumentList = `[` [*Expression*] { `,` *Expression* } `]` .
Selector = `.` *Ident* .

Programs

Program = *DefinitionSequence* .

A.5 Keywords

There are eight keywords that are reserved and must not be redefined:

```
CASE  ELSE  IF     LIST
OF    THEN  TUPLE  UNION
```

Appendix B

Predefined Identifiers

Predefined identifiers may be redefined by the user, although it is strongly recommended not to do so. There are three classes of predefined identifers:

- Types,
- Constants,
- Functions.

B.1 Predefined Types

EvE has four predefined types, which are shown in Table B.1.

Name	*Description*
INT	Integer Numbers
BOOL	Boolean Values
CHR	Characters
STR	Strings

Table B.1: Predefined Types

B.2 Predefined Constants

Table B.2 shows the predefined constants of EvE.

The constant ERROR is a generic constant, i.e. it is element of any type and therefore compatible with every type.

Name	*Type*	*Description*
MININT	INT	Lower bound of integer numbers
MAXINT	INT	Upper bound of integer numbers
TRUE	BOOL	Boolean constant
FALSE	BOOL	Boolean constant
ERROR	*Any*	Error value

Table B.2: Predefined Constants

B.3 Predefined Functions

B.3.1 Functions on Atomic Types

Tables B.3, B.4 and B.5 show the predefined functions working on integers, booleans and characters, respectively.

Name	*Type*	*Description*
ABS	INT[INT]	Absolute Value
ADD	INT[INT,INT]	Addition
SUB	INT[INT,INT]	Subtraction
MUL	INT[INT,INT]	Multiplication
DIV	INT[INT,INT]	Division
MOD	INT[INT,INT]	Remainder
NEG	INT[INT]	Negation
ODD	BOOL[INT]	Is number odd?

Table B.3: Predefined Integer Functions

Name	*Type*	*Description*
AND	BOOL[BOOL,BOOL]	Conjunction
OR	BOOL[BOOL,BOOL]	Disjunction
NOT	BOOL[BOOL]	Negation

Table B.4: Predefined Boolean Functions

Name	*Type*	*Description*
CAP	CHAR[CHAR]	Capitalize a letter
CHR	CHAR[INT]	Character from ASCII code
ORD	INT[CHAR]	ASCII code of a character

Table B.5: Predefined Character Functions

B.3.2 Comparison Functions

Table B.6 describes the functions that can be used to compare objects of atomic type (i.e. integers, booleans and characters). For each kind of comparison operator, there exist *three functions*, one for each atomic type. In order to get the actual function names, one has to replace the asterisk * in the names by `I (INT)`, `B (BOOL)`, or `C (CHAR)`.

Function	*Type*	*Description*
*LT	BOOL[*Atomic,Atomic*]	Less Than
*LE	BOOL[*Atomic,Atomic*]	Less or Equal
*EQ	BOOL[*Atomic,Atomic*]	Equal
*NE	BOOL[*Atomic,Atomic*]	Not Equal
*GE	BOOL[*Atomic,Atomic*]	Greater or Equal
*GT	BOOL[*Atomic,Atomic*]	Greater Than

Table B.6: Predefined Comparison Functions

B.3.3 List Functions

Table B.7 refers to the generic type definition

AnyList == LIST OF *Any*.

Function	*Type*	*Description*
CONS	*AnyList*[*Any,AnyList*]	Construct a list
HEAD	*Any*[*AnyList*]	Select the head of a list
TAIL	*AnyList*[*AnyList*]	Select the tail of a list
NULL	BOOL[*AnyList*]	Is a list empty?

Table B.7: Predefined List Functions

B.3.4 Miscellaneous Functions

The generic functions shown in Table B.8 can be applied to expressions of *any* type.

Function	*Type*	*Description*
`DELAY`	*Any*[*Any*]	Delay an expression
`FAILED`	`BOOL`[*Any*]	Test for `ERROR`

Table B.8: Miscellaneous Functions

References

[Abelson and Sussman 1985]
Harold Abelson and Gerald Jay Sussman.
Structure and Interpretation of Computer Programs.
The MIT Electrical Engineering and Computer Science Series. The MIT Press, 1985.

[Arvind and Iannucci 1987]
Arvind and Robert A. Iannucci.
Two Fundamental Issues in Multiprocessing.
Computation Structures Group Memo 226-6, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1987.
Also in: Proceedings of the DVLR-Conference on Parallel Processing in Science and Engineering, Bonn/Bad Godesberg, June 25–26, 1987.

[Arvind *et al.* 1988]
Arvind, David E. Culler, and Gino K. Maa.
Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs.
Computation Structures Group Memo 279, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988.
Revised June 1988. To appear in the *International Journal of Supercomputer Applications.*

[Bird and Wadler 1988]
Richard Bird and Philip Wadler.
Introduction to Functional Programming.
Series in Computer Science. Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

[Boyle 1989]
James M. Boyle.
Deriving Parallel Programs from Pure Lisp Specifications by Program Transformation.
Lecture held at RISC-Linz, Johannes Kepler University, Linz, Austria, June 1989.

[Buchberger 1978]
Bruno Buchberger.
Computer-Trees and their Programming.
In Dauchet, editor, *4th Colloquium "Trees in algebra and programming"*, pages 1–18, University of Lille, February 16–18, 1978.

Also: Technical Report 78-11, RISC-Linz, Johannes Kepler University, Linz, Austria, 1978.

[Buchberger 1985]
Bruno Buchberger.
The L-Machine: An Attempt at Parallel Hardware for Symbolic Computation.
In J. Calmet, editor, *3rd International Conference on Algebraic Algorithms and Error-Correcting Codes*, volume 229 of *Lecture Notes in Computer Science*, pages 333–347, Grenoble, France, July 15–19, 1985. Springer.
Also: Technical Report 85-24, RISC-Linz, Johannes Kepler University, Linz, Austria, 1985.

[Harper *et al.* 1988]
R. Harper, R. Milner, and M. Tofte.
The Definition of Standard ML — Version 2.
Technical Report LFCS-88-62, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, 1988.

[Kusche 1988]
Klaus Kusche.
LL User Manual.
Master's thesis, Johannes Kepler University, Linz, Austria, 1988.
Also: Technical Report 88-25, RISC-Linz, Johannes Kepler University, Linz, Austria, 1988.

[Nikhil 1988]
Rishiyur S. Nikhil.
ID (Version 88.0) Reference Manual.
Computation Structures Group Memo 284, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988.

[Peyton Jones 1987]
Simon L. Peyton Jones.
The Implementation of Functional Programming Languages.
Prentice-Hall, New York, 1987.

[Schreiner 1990]
Wolfgang Schreiner.
ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language.
Master's thesis, Johannes Kepler University, Linz, Austria, October 1990.
Also: Technical Report 90-42.0, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990. Also: Technical Report 91-1, Austrian Center for Parallel Computation, January 1991.

[Turner 1985]
D. A. Turner.
Miranda: A Non-strict Functional Language with Polymorphic Types.
In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, September 16–19, 1985. Springer, Berlin.