ACM 2012. This is the author's version of the work. It is posted here by permission of the ACM for your personal use. Not for redistribution. The definitive version was published in Onward! 2012 Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software, pp. 159–178, http://doi.acm.org/10.1145/2384592.2384608

Modular, Statically Typed, Flexibly Extensible Programming Language

Christian Heinlein
University of Applied Sciences, Aalen, Germany
christian.heinlein@htw-aalen.de

Abstract

Even though extensible programming languages have been around for decades, they have not received much attention so far. To obtain a more attractive solution, where extending the language is almost as easy as writing normal programs, extensibility should not be provided as a separate add-on, but rather as the very heart of the language. Furthermore, syntactic flexibility should not only allow to extend, but also to completely change the syntax when desired. MOSTflexiPL follows this approach by allowing users to define new operators, control structures, type constructors, and even declaration forms almost as easily as functions without sacrificing static type safety. This is achieved by encoding all constructs as generalized operators possessing any number of names and operands in an arbitrary order, where users have full control over associativity, precedence, and even scoping rules. Even though the language is still under development, there is a working compiler that translates MOSTflexiPL programs to equivalent C++ code.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Extensible Languages

Keywords Programming Language, Syntactic Extensibility, Static Type System

1. Introduction

Humans wish to abbreviate lengthy statements and to avoid repetition; e. g., experts extend the *vocabulary* of natural language with new words and jargon to optimize their conversations with other experts. Natural language *syntax* is usually fixed, however, and so we also develop synthetic languages such as mathematical notation providing tailored syntax (e. g., for limits and integrals in calculus) to express our thoughts even more concisely. If their standard syntax appears insufficient for a particular domain, it can simply be extended with new, more appropriate forms.

Similar to natural languages, however, programming languages typically force their users into a fixed syntactic corset with only a few convenient notations for frequent idioms, e. g., arithmetic operators and basic control structures. Their extensible vocabulary of functions, types, etc. cannot really compensate for their lack of extensible syntax; for example, iterating over a list with a tailored `for` loop is more convenient than calling a `for` function with a body closure; likewise, a tailored notation such as `s[3..7]` conveys the notion of subsequence much more clearly than a call to an aptly named method.

Of course, it would be possible to include the convenient notations just mentioned in our programming languages, but it is obviously impossible – and presumably not even desirable – to predefine all the notations that any programmer in the world might ever consider useful. Instead it seems more appropriate to give programmers the possibility to define their own notations when they see fit, i. e., to *extend or customize the syntax* of their programming language, similar to the way mathematicians extend or customize their notation on demand.

Even though this idea of syntactically extensible programming languages is by no means new, it does not have attracted much attention in the past. It seems that many programmers either do not know very much about extensible languages at all or do not consider them particularly useful for improving their everyday work. Maybe, this is due to the fact that these languages are often split into two separate parts: a "normal" programming language for writing "normal" programs, plus a special machinery (or "magic") such as a macro system for performing syntactic extensions, where the latter is considerably more complicated to use than the former. Furthermore, many extensible languages still impose more or less significant syntactic restrictions, i. e., they only allow their users to stretch the syntactic corset a bit, but not to completely take it off.

To address these shortcomings and to provide a more attractive extensible programming language, syntactic extensibility should not be offered as a separate add-on, but

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Onward! 2012, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1562-3/12/10. . . $10.00

rather as the very heart of the language, i. e., the entire language design should be based on the concept of extensibility. Consequently, there should not be much difference between writing "normal" programs and performing syntactic extensions, and doing the latter should be just as easy as the former. Furthermore, the syntactic flexibility should be as high as possible (as embodied in the MOSTflexiPL logo used in the paper title), so that users can not only extend the syntax of the language, but completely change it (or "turn it upside down" as in the logo) when desired.

This paper presents MOSTflexiPL[1] based on this approach: users can define new operators, control structures, type constructors, and even declaration forms almost as easily as functions. Such broad extensibility is enabled by encoding all constructs as generalized operators possessing any number of names and operands in an arbitrary order. Furthermore, users have full control over operator associativity and precedence as well as scoping rules. Despite its tremendous flexibility, the language possesses a static type system with bounded parametric polymorphism, where types might also depend on constant values.

flexiPL can be used amongst others as an extensible general purpose programming language, where broadly useful extensions can be easily distributed as operator libraries, as a host language for embedding domain-specific languages, and as a playground for experimenting with new language constructs. Because syntactic extensibility might not only be used adequately to make programming simpler and programs easier to understand by carefully designing reasonable new language constructs, but also inadequately to confuse programmers and to obfuscate code by unthoughtfully overwhelming a program with useless new language constructs, it might be advisable in practice to agree on "language extension conventions" or "syntax guides" similar to well-known coding conventions and style guides.

The remainder of the paper is structured as follows. Section 2 unfolds the approach that has been taken to design an "extensibility-centered" programming language such as flexiPL and introduces its fundamental language constructs. Afterwards, Sec. 3 shows flexiPL in action by presenting some instructive examples of user-definable language extensions. Section 4 describes the basic structure of the flexiPL compiler and explains some current limitations of the language. Related work is discussed in Sec. 5, and Sec. 6 concludes the paper.

2. Overview

How can we design a programming language with virtually unlimited syntactic flexibility, which is at the same time easy to use and statically type-safe? With such a language, it should be easily possible to simulate language constructs found in other languages. Therefore, it is advisable to study many existing languages and to carefully analyze their constructs with the aim of unifying and generalizing them to obtain a minimal set of orthogonal language features that can be successfully combined and employed for many different purposes. When studying the grammar of programming languages, one can usually identify four major areas – expressions, statements, types, and declarations – that shall be discussed in turn in the following subsections, because in a flexible language each of them should be fully extensible and customizable.

2.1 Expression Syntax

2.1.1 Simple Operators and Exclude Declarations

C++, Ada, and other languages support *operator overloading*, i. e., to define the meaning of *predefined* operators (such as `- _` and `_ * _`, where the underscores indicate the positions of their operands) being applied to *user-defined* types (e. g., complex numbers or matrices), which is just as easy as defining functions. Even though this is not true syntactic extensibility, it allows to use familiar operator syntax such as `-A * B` where clumsy function or method syntax such as `A.negate().multiply(B)` is required in other languages.

C+++ [9], Haskell, and other languages go a significant step further by allowing users to define *new* operator symbols, e. g., `_ ^ _` and `_ !` to denote exponentiation and factorial, respectively, which can then be used in expressions such as `- 2 ^ 3 ^ 4!` just like predefined operators. Defining the meaning of these operators is again straightforward; for example, in flexiPL the factorial operator can be (recursively) defined as follows:

```
["n" : int]   ← parameter list          implementation
n "!" : int   ← signature and result type      ↓
{ if n > 1 then (n-1)! * n else 1 end }
```

Static type safety is achieved by specifying both the parameter or operand types and the result type of each operator. Then, the application of an operator is *type-correct* (and its type is equal to the operator's result type), if its operands are in turn type-correct and their types match the corresponding parameter types.

Apart from defining the meaning of new operators, it is also desirable to specify their *associativity* and *precedence*. The latter is usually accomplished by allowing users to specify a numerical precedence value (e. g., from 0 to 9) for infix operators and by assigning a fixed precedence to prefix and postfix operators which is either higher or lower than that of all infix operators. Obviously, this approach suffers from a number of limitations:

- It is impossible to define a precedence between two adjacent values. Even though this can be mitigated in practice by extending the range of possible values and trying to always leave gaps for future extensions, this does not solve the problem in principle.

[1] Or just flexiPL, which is pronounced like "flexible" with a "hard" (unvoiced) p instead of a "soft" (voiced) b.

- While it is possible to carefully design the precedence relationships within a single operator library, operators from different libraries will have an accidental relationship which might or might not be appropriate.
- A fixed precedence for prefix and postfix operators with respect to infix operators disallows, for example, that negation binds stronger than addition, but weaker than exponentiation, as accustomed from mathematics.

An effective solution for these problems is to define an operator's precedence only *relative* to other operators, i. e., to establish only a *partial* instead of an (artificial) total ordering, and to extend this ordering from infix to prefix and postfix operators. Furthermore, it must be possible to extend the ordering later, e. g., to define precedence between operators from different libraries.

In flexiPL, both associativity and precedence can be expressed in terms of very general *exclude declarations*. For example, the well-known "× and ÷ before + and −" rule can be expressed by forbidding additive operators as *top-level operators* of the operands of multiplicative operators. This implies, for example, that of the two basically possible interpretations of `2 + 3 * 4` only the one having the multiplication below the addition will be correct. To declare a binary operator (or a group of such operators) left- or right-associative, it is sufficient to exclude the operator itself as top-level operator of its right or left operand, respectively. For example, the exclude declarations for the predefined multiplication operator for `int` values read as follows:

```
[
  "x" : int; "y" : int;   ← parameters
  x :- additive;   ← exclude declarations
  y :- additive|multiplicative ↵
]
x "*" y : int   ← signature and result type
{ ... }         ← predefined implementation
```

If the precedence relationships constitute only a partial ordering, incomparable operators might clash in an expression, causing it to become syntactically ambiguous. This can either be remedied by the user with explicit bracketing or possibly by the compiler by automatically ruling out possibilities which are not type-correct, i. e., to use *semantic* information (the types of operands) to break *syntactic* ambiguities. By incorporating type information into the parsing process, it is even possible to make the precedence of an overloaded operator depend on the types of its operands. For example, addition might have a higher priority than string concatenation, even though both operations might be denoted by an infix plus operator.

2.1.2 Generalized Operators with Special Cases

Apart from unary and binary operators, programming languages often provide specialized constructs such as `_ ? _ : _` (the ternary conditional operator in C et al.) and `_ [ _ ]` (the array subscripting operator in many languages). To allow users to define similar constructs, the notion of operators should be generalized to *multi-part operator combinations* (sometimes also called distfix or mixfix operators) with any number of names (or operator symbols) and any number of operands in an arbitrary order, e. g.:

```
[
  "x" : bool;
  "T" : type; "y" : T; "z" : T
]
x "?" y ":" z : T
{ if x then y else z end }
```

If such a multi-part operator possesses *outer operands* appearing before the first or after the last operator name, its application is analogous to that of a prefix, infix, or postfix operator and therefore it should also be possible to specify associativity and precedence. For example, the conditional operator in C has the same associativity and precedence as the infix assignment operators, while the subscripting operator is usually treated like a postfix operator with very high precedence.

If an operator possesses only *inner operands*, i. e., if it starts and ends with an operator name, its application is analogous to a bracketed expression. In fact, if it is possible to define arbitrary multi-part operators, the parentheses `( _ )`, which almost always constitute a very special built-in language construct, become an ordinary user-definable *circumfix* operator, whose application simply returns the value of its single operand:

```
["T" : type; "x" : T]
"(" x ")" : T
{ x }
```

To provide a maximum of flexibility, operator names should be composable of arbitrary characters, in contrast to many other extensible languages, where they are frequently restricted to either identifiers (i. e., sequences of letters and digits starting with a letter) or sequences of special characters, where parentheses are usually excluded due to their special, predefined meaning.

Remarkable special cases of this generalized concept of operators are:

- *Nullary operators*, possessing only names but no operands, can be used to represent literals, constants, and variables (even with unusual multi-part names such as `2nd last char`), i. e., these need not be provided as separate language constructs.
- *Anonymous operators*, possessing only operands but no names, can be used, amongst others, to denote multiplication (e. g., `2 a` instead of `2 * a`, as accustomed from mathematics) or string concatenation (e. g., `firstname " " lastname`, as in AWK), for example:

```
["x" : int; "y" : int]
x y : int
{ x * y }
```

- As a very special case, an *anonymous unary operator* with parameter type X and result type Y happens to represent an *implicit type conversion* from X to Y, because it can ("invisibly" because it has no names) be applied to any subexpression with type X and returns a value of type Y. (This also implies that conversions can be applied transitively.)

Of course, *functions* are also just a special case of operators now, whose calling syntax can be defined at will by choosing an appropriate operator signature, e. g., `max _ _` (as in many functional languages), `max ( _ , _ )` (as in many imperative languages), or even `_ _ max` (as in PostScript).

It should be emphasized that the apparently simple generalization of traditional operators to multi-part operator combinations actually provides the key to flexiPL's flexibility, because defining an operator – which is still all but easy as defining a function in other languages – now means to perform a *syntax extension* along the way, and therefore extending the syntax of the language becomes the same as "ordinary" programming, as demanded in the introduction.

2.1.3 Parameter Passing and Variable Types

In a language with imperative features, where expressions can have side effects, *call by value* seems to be the appropriate mode for passing parameters to operators. Furthermore, operands should be evaluated in their "natural" order from left to right, as, e. g., in Java, in contrast to C and C++ where the order might be arbitrarily chosen by the compiler to allow for a maximum of optimizations.

Some operators, however, e. g., Boolean operators with "shortcut evaluation" and the aforementioned conditional operator, cannot be implemented faithfully if all operands are evaluated in advance. Instead it is necessary to pass individual operands *unevaluated* (or "by expression") and to allow the operator implementation to decide whether and when they should be evaluated. Technically, this can be achieved by wrapping these operands into parameterless local functions (or closures), but the user should not be burdened with doing that explicitly. Because an operator might well have both call by value and call by expression operands, it should be possible to mark individual parameters instead of an entire operator as "call by expression;" e. g., to give a more realistic implementation of the conditional operator:

```
[
  "x" : bool;
  "T" : type; "y" : T {}; "z" : T {}
]
x "?" y ":" z : T
{ if x then y else z end }
```

Here, x is passed by value, while y and z are passed by expression (indicated by {}) and will be evaluated only if the `then` resp. `else` part gets executed.

Call by reference is another mode occasionally found in imperative languages that actually allows *variables* instead of their values to be passed (where variables might also include array elements and the like). Because sometimes it is also necessary to *return* variables (e. g., the prefix ++ operator of C returns the variable it increments as an "L-value"), it is more effective to provide a general notion of *variable types* (denoted `T?` in flexiPL) with accompanying operators `? _` to explicitly obtain the value of a variable (plus an implicit type conversion from `T?` to `T` that does the same more conveniently) and `_ << _` (or `_ >> _`) to assign to a variable of type `T?` a value of type `T` (or vice versa). This is similar to reference types in C++ (`T&`, not to be confused with pointer types and references in other languages), but without their irregularities, e. g., that references to references are forbidden, and `int x` actually declares `x` with type `int&` (or L-value of type `int`) which must be implicitly converted to `int` (or R-value of type `int`) on demand. Using a variable type as a parameter or result type of an operator effectively allows call or return by reference:

```
["x" : int?]
"++" x : int?
{ x << ?x + 1; x }
```

2.2 Statement Syntax

Imperative languages usually distinguish between expressions and statements, a distinction that is partially discretionary and artificial. C and C++, for example, provide both *statement* sequences and sequential executions within *expressions* (comma operator); similarly, there is a conditional *statement* (`if`) as well as the already mentioned conditional *operator*. This replication could be avoided by unifying statements and expressions, similar to functional languages. For example, by subsuming statements beneath expressions, an operator `if ( _ ) _ else _` (with four operator names and three operands) can in fact be used just like a conditional statement in C.

As a consequence, if the syntax of expressions can be extended by defining new operators, the syntax of statements immediately becomes extensible, too, since control structures such as `if`, `while`, `for`, etc. are nothing else but operators then, e. g.:

```
[
  "x" : int?; "x1" : int; "x2" : int;
  "B" : type; "body" : B {}
]
"for" x "from" x1 "to" x2 "do" body "end"
                                     : int
```

```
{
  x << x1;
  while x <= x2 do body; ++x end
}
```

Again, to provide a maximum of flexibility, operator syntax should be overloadable in an arbitrary fashion. For example, it should be possible to define both `if _ then _ end` and `if _ then _ else _ end` (in fact, both of these operators are predefined), even though this might complicate parsing of expressions starting with `if ... then ...`

When comparing operator definitions with grammar productions, this means that the grammar defined by all operators together can be an arbitrary context-free one, i. e., the user should not be burdened with restrictions such as LL(k) or LR(k), even though they could simplify the parser (which then has to perform backtracking, cf. Sec. 4.1.1). This view is confirmed by the fact that parser generators typically provide workarounds for such restrictions, e. g., JavaCC provides "syntactic and semantic lookahead specifications" to circumvent the LL(1) requirement, while Bison provides "generalized LR parsing" to overcome the LALR(1) restriction.

Another issue that needs consideration in the context of statements are *visibility* or *scoping rules*. For example, C++ and Java allow variables to be declared in the initialization part of a `for` loop, whose scope is limited to the loop. Similarly, `let` forms can be used in functional languages to introduce names with limited scope. To allow users to define similar constructs, the visibility of operators must be user-controllable. More specifically, it should be possible to specify for an operator:

- Which operators are *exported* from an application of this operator? This is typically specified recursively in terms of the operators exported by the individual operands. For example, an application of the predefined sequential composition operator `_ ; _` exports the union of the operators exported by its operands, while most other operators do not export anything (which terminates the recursion). As another important recursion base case, a declaration (cf. Sec. 2.4) exports the operator declared by it.
- Which operators are *imported* by the individual operands of an application of this operator? This is also typically specified recursively in terms of the operators imported by the entire operator application and possibly the operators exported by preceding operands[2]. For example, the first operand of the sequential composition operator simply imports the operators imported by the entire operator application (which is also the most frequent default case for operands of other operators), while the second operand additionally imports the operators exported by the first operand to make them visible there. As the single recursion base case, a top-level expression (actually denoting a flexiPL *program*) imports the set of all predefined operators.

[2] As a special rule, a call-by-expression operand does not export anything, because it effectively constitutes the implementation of an anonymous, parameterless local operator (cf. Sec. 2.1.3).

As an example, the following operator `use _ in _ end` behaves identical to the predefined sequential composition operator, except that it does not export operators declared in its first operand:

```
[
  "X" : type; "x" : X;
  "Y" : type; "y" : Y;
  x :+ ^;      ← import declaration for 1st operand
  y :+ ^|x;    ← import declaration for 2nd operand
  ^ :+ y       ← export declaration
]
"use" x "in" y "end" : Y
{ y }
```

This can be used to achieve information hiding, e. g.:

```
use
  "n" : int?; n << 0
in
  "inc" : int
  { n << n + 1 }   ← OK, n is visible here
end;

inc;      ← OK, inc is visible here
n << 5    ← error: n is not visible here
```

2.3 Type Syntax

2.3.1 Type Constructors and Static Operators

The next area that needs consideration are types. Usually, a programming language provides a couple of *basic types* (such as `int` or `bool`), plus a few generic *type constructors* that can be used to construct new types. For example, `_ *` constructs pointer types in C, while `_ [ _ ]` constructs array types.

From a purely syntactic point of view, type constructors are operators (as already indicated by the underscore notation) that are applied to types and possibly non-type values and which return new types. For example, `int*` can be interpreted as an application of the postfix operator `_ *` to the basic type `int` (which might again be interpreted as an application of the nullary constant operator `int`, cf. the special cases in Sec. 2.1.2), while `bool [10]` is an application of the operator `_ [ _ ]` to the type `bool` and the `int` value `10`. Using this analogy, C++ class templates and Java generic classes correspond to functions on the type level, whose arguments are enclosed in angle brackets instead of parentheses.

If types can actually be used as operands and result values of operators, the syntax of types becomes immediately extensible, too, simply by defining type-valued operators. For example, one might define an operator `[ _ ]` to construct list types such as `[int]` in Haskell (which is more convenient than `list<int>`) or an operator `_ { _ x _ }` to construct matrix types such as `float{3x4}` (looking more familiar to mathematicians than `matrix<float, 3, 4>`). To actually define such operators, a meta-type called `type` is required whose values are types and which can be used as a parameter and result type of operators. In fact, types become just expressions whose type is equal to `type`.

Nevertheless, types remain something special in a statically typed language, because the compiler must be able to reason about them. In particular, it must be decidable at compile time whether two types are equal. This implies immediately, that operators whose behaviour cannot be statically determined, must not be used in type expressions, even if they return a value of type `type`.

When looking back at type constructors in conventional languages, these actually do not perform any computations to construct new types, but merely perform *mappings* from types and possibly other values to types according to the following rules:

- Every type constructor embodies an *injective* function, i. e., its applications to the same values produce the same result, while applications to different values produce different results.
- The domains of these functions are pairwise *disjoint*, i. e., the types returned by a particular type constructor are different from all types returned by other type constructors.

To make sure that user-defined type constructors obey the same rules, they must be defined as so-called *static operators*, which do not possess a user-defined implementation that could be executed at run-time to compute its result value, but only an (optional) parameter list, a signature, and a result type[3], e. g.:

```
["T" : type; "M" : int; "N" : int]
T "{" M "x" N "}" : type
```

Their implementation, which is automatically provided by the run time system, guarantees the "injective and disjoint mapping behaviour" described above. As a special case, a parameterless static operator simply returns a unique constant value. As another special case, literals and constants, i. e., nullary operators initialized with an arbitrary value, are also treated as static operators, even if the initialization value is not known at compile time – it suffices to know that the operator will always return the same value.

Based on this notion of static operators, a type can now be defined as a *static expression*, i. e., an expression that is solely made up of static operators. (Operators which are not static, are called *dynamic* in the sequel.) According to the above rules, this implies that static expressions which are *structurally equal* – a property that can easily be examined at compile time – are guaranteed to evaluate to the same value at run time. Therefore, equality of types can simply be defined as structural equality of the corresponding expressions, just as in conventional languages.

Restricting types to static expressions implies that they cannot contain computations of values (because computations are performed by dynamic operators), but only constant values. In contrast to, e. g., C++ templates, however, these constants need not be compile-time constants nor need they have an integral type. It is in fact possible to initialize a constant with a dynamically computed value and use that constant in type expressions afterwards, e. g.:

```
"M" := ...;          ← compute values of M
"N" := ...;          ← and N at run time
"A" : float{MxN};    ← and use them in type
"B" : float{MxN};    ← expressions afterwards
```

Even though the compiler cannot know the particular value of the constant, it knows that it will always be the same and this is sufficient to faithfully determine equality of types. More precisely: It implies that structurally equal static expressions will evaluate to the same value at run time, as already mentioned before, i. e., types regarded equal by the compiler will actually be equal at run time, too. It might happen, however, that structurally different expressions also evaluate to the same value, for example if they contain different constants having the same value. That means, that types considered different by the compiler might happen to be equal at run time, but this kind of "false alarms" cannot break static type safety.

The actual value of a type expression is rarely used at run time, because currently flexiPL does not provide any reflective language features. However, the non-type values contained in a type, e. g., the dimensions of a matrix type, might well be useful (cf. the example given in Sec. 3.6).

2.3.2 Type Aliases and Virtual Operators

Because parametric types such as lists and matrices can be combined, e. g., to form a list of list of matrices, the resulting type expressions can get rather complex. Therefore, it should be possible to abbreviate them, similar to using `typedef` in C, i. e., to define *type aliases*. Going a step further, it should also be possible to define *parametric* aliases, e. g., to define `T{N}` as an abbreviation or synonym for `T{Nx1}` for all types `T` and integer values `N`.

At first sight, one might simply define an operator `_ { _ }` with result type `type`, a `type` parameter `T`, and an `int` parameter `N`, whose implementation computes and returns the type `T{Nx1}`. This will not work, however, because operators possessing a user-defined implementation are classified as dynamic operators which cannot be used in types because these must be static expressions.

To make the compiler accept an expression such as `float{3}` as a type and to actually consider it equal to `float{3x1}`, an appearance of the former should be im-

[3] Even though the primary reason for introducing static operators is to use them as type constructors, it turns out that they can be successfully employed for other purposes, too (cf. Sec. 3.1). Therefore, their result type is not restricted to the meta-type `type`.

mediately replaced by the latter at compile time, i. e., the operator `_ { _ }` should be treated similar to a macro. To remain within the bounds of static type safety, however, and to seamlessly integrate this kind of macros into the remaining language, `_ { _ }` should still be defined and used like an operator:

```
["T" : type; "M" : int]
T "{" M "}" = T{Mx1};

"u" : float{3};
```

This means in particular:

- The macro has a parameter list, a signature, and a result type (which is not given explicitly, though, but is deduced as the type of the right hand side), just like a normal operator definition.
- Its right hand side must be a type-correct expression, just like the implementation of a dynamic operator.
- Its applications look identical to and are type-checked in the same way as applications of other operators.

Only after an application has been confirmed as a type-correct expression – which implies that its operands are in turn type-correct and their types match the types of the corresponding parameters (cf. Sec. 2.1.1) –, it is replaced by the right hand side, where the parameters are in turn replaced with the corresponding operands. Because the right hand side has already been proven type-correct and the operand types match the corresponding parameter types, the finally resulting expression is necessarily type-correct, too. That is, it is only necessary to type-check the definition and the applications of a macro, not the expanded expressions, which implies that the expansion cannot produce unexpected errors.

Because of the striking similarities with "real" operators, these macros will be called *virtual operators* in the sequel and their right hand side is called their *realization*.

2.4 Declaration Syntax

Given the general notion of operators that has emerged as the cornerstone of the language, even declarations can be seen as a kind of expressions, i. e., as applications of special predefined *declaration operators* to appropriate operands. For example, a Pascal variable declaration such as `x : int` is syntactically an application of an infix operator `_ : _` to the identifier `x` and the type `int`. Likewise, a C function declaration can be interpreted as an application of a declaration operator `_ _ ( _ ) { _ }` whose operands represent the functions's result type, its name, its parameter list (which is in turn a sequence of parameter declarations), and its implementation (an arbitrary expression).

In fact, flexiPL provides several similarly shaped declaration operators such as `[ _ ] _ : _ { _ }`, which can be used to define the various kinds of operators introduced so far, i. e., static, dynamic, and virtual operators, each with or without parameters. Because operator names might be arbitrary character sequences (cf. Sec. 2.1.2), they must be enclosed in quotation marks to avoid potential ambiguities when parsing an operator declaration.[4] Afterwards, however, when applying an operator, the bare names without quotation marks can be used.

Even though declarations are just expressions from a syntactic point of view, they play a very special role for the compiler, because they influence the subsequent compilation process by defining new operators that can be applied afterwards. Therefore, after having successfully processed a declaration, the compiler must add the operator declared by it to its internal table of operators that governs the subsequent parsing process (cf. Sec. 4.1.1).

If a user wants to define a new declaration operator, for example to declare variables in a C-like notation, he is faced with similar problems as with defining type aliases in Sec. 2.3.2: Using a normal dynamic operator for that purpose, whose implementation contains the declaration of the variable, will not work, because the variable declared that way will be local to the implementation, i. e., a subsequent application of this operator will not have any effect at compile time.

However, the basic solution to this problem is also similar to that for type aliases: If the new declaration operator is defined as a virtual operator, whose realization consists of the declaration of a variable, a subsequent application will be immediately replaced with this realization, causing the variable declaration to actually appear at this point of application:

```
["T" : type; "name" : string]
T name = name : T?;

int "x"    ← equivalent to: "x" : int?
```

In general, the matter is more subtle, however: Declarations appearing in the realization of a virtual operator shall typically depend on the parameters of this operator, e. g., in the previous example the name and the type of the variable are passed as parameters. This implies that the actual name of the declared variable is not known when the virtual operator is defined, which makes it difficult to type-check its realization in more complex cases (e. g., if the variable should be assigned an initial value afterwards). On the other hand, type-checking virtual operators already at their definition shall in no way be compromised. In fact, it seems possible to reconcile the flexibility needed for more advanced user-defined declaration operators with complete static type checking, but the details have not yet been worked out completely (see also Sec. 4.2.1).

Because every operator declaration effectively constitutes a syntax extension (cf. Sec. 2.1.2), defining user-

[4] It is planned to relax this restriction in the future, e. g., by allowing standard identifiers without quotation marks, too. In fact, users should be able to freely define the set of character sequences that can be used without quotation marks.

defined declaration operators means in the end to extend or customize even the syntax that is used to define new syntax.

2.5 Summary

Reviewing expression, statement, type, and declaration syntax has revealed that a general notion of operators with any number of names and operands in an arbitrary order is sufficient to achieve virtually unlimited syntactic flexibility simply by defining new operators. This can be summarized as the most fundamental design principle of MOSTflexiPL: *Everything is an expression,* i. e., the application of an operator to operands which are in turn expressions.

A second important principle could be termed: *One law for all,* i. e., there is no difference whatsoever between user-defined and predefined operators. They are used in exactly the same way, obey exactly the same rules, etc. The only special thing with (some) predefined operators is the fact that they *must* be predefined, because they cannot (or not sensibly) be defined in terms of other operators.

3. Examples

To give the reader an impression of the possibilities of flexiPL, some instructive examples of user-definable language extensions shall be presented in the sequel. Their complete source code as well as more extensive examples can be found on the website `flexipl.info`.

3.1 Open Types

flexiPL provides predefined basic types `char`, `bool`, `int`, and `float` with the usual arithmetic and logic operators as well as two predefined type constructors, `_ ?` and `_ *`, to construct variable and sequence types, respectively. While variable types `T?` have been introduced in Sec. 2.1.3, the type `T*` represents sequences of elements of type `T` of any (finite) length. For such a sequence `s`, the expression `#s` returns the length of `s`, i. e., the number of its elements, while `s[i]` returns the `i`-th element of `s` counted from 1 (because humans start counting at 1). Sequences and individual elements can be concatenated with the operator `_ , _` to construct longer sequences.

flexiPL does not provide a direct means to construct compound types, however, because this can be achieved in a rather flexible way by combining other language features. In fact, after defining a few additional operators below, the following code will work:

```
"person" : type;
"name" : person -> string;

"John" : person;
John.name << "John Smith";

"spouse" : person -> person;

"Jane" : person;
John.spouse << Jane;
Jane.spouse << John;

print the name of Jane's spouse
```

First of all, `string` can be defined as a type alias (cf. Sec. 2.3.2) for a `char` sequence:

```
"string" = char*;
```

The predefined declaration operator `_ : _` declares parameterless static operators, i. e., constants possessing a uniquely generated value (cf. Sec. 2.3.1). Thus, `person` is a unique new type, while `John` and `Jane` denote unique objects or values of that type.

The arrow is the following user-defined type constructor that maps each pair of types `X` and `Y` to a unique new type `X -> Y` that is intended to represent an attribute of type `X` with target type `Y`:

```
["X" : type; "Y" : type]
X "->" Y : type;
```

Accordingly, `name` and `spouse` denote attributes of type `person`.

The dot is also a user-defined static operator mapping an object `x` of any type `X` and an attribute `a` of a corresponding type `X -> Y` to a unique variable of type `Y?` that can be used to store the object's value of the attribute:

```
[
  "X" : type; "Y" : type;
  "x" : X; "a" : X -> Y
]
x "." a : Y?;
```

Besides the *explicit parameters* `x` and `a` appearing in its signature, this operator possesses *deduced parameters* `X` and `Y`, which appear in the types of other parameters and therefore can be deduced from the types of the corresponding operands. For example, because `John` is an object of type `person` and `name` is an attribute of type `person -> string`, the expression `John.name` is a type-correct application of the dot operator deducing `X` and `Y` as `person` and `string`, respectively, and therefore returning a unique variable of type `string?` that is used to store the `name` of `John`. Likewise, the `spouse` of `John` is set to `Jane` and vice versa.

The operators `_ 's _` and `the _ of _` are just alternative syntactic forms of the dot operator that make the natural language-like expression `print the name of Jane's spouse` (using the predefined output operator `print _`) equivalent to `print Jane.spouse.name`:

```
[
  "X" : type; "Y" : type;
  "x" : X; "a" : X -> Y
]
x "'s" a : Y?
{ x.a };
```

```
[
  "X" : type; "Y" : type;
  "x" : X; "a" : X -> Y
]
"the" a "of" x : Y?
{ x.a };
```

Apart from these natural language gimmicks, the above concept of attributes is very flexible, because it allows attributes of a type to be defined modularly and incrementally (e. g., `spouse` is defined later), similar to inter-type declarations in aspect-oriented programming. As outlined in [11], the concept can be generalized to bidirectional relationships, which (when combined with implicit type conversions) can be used as a superior replacement for single, multiple, and even repeated inheritance.

3.2 Construction of Subsequences

Using the predefined operators for sequences (cf. Sec. 3.1) plus some other predefined and user-defined operators introduced earlier, it is easily possible to define an operator `subseq` _ _ _ that constructs and returns a subsequence of a given sequence:

```
[
  "T" : type;
  "s" : T*; "i" : int; "j" : int
]
"subseq" s i j : T*
{
  "t" : T*?; "k" : int?;
  for k from max i 1 to min j #s do
    t << t, s[k]
  end;
  t
};
```

When looking at the implementation (or an accompanying documentation), one can see that the result sequence `t` will contain all elements of `s` whose index is between `i` and `j`, both inclusively, even in cases where these indices are out of range. But a mere application of the operator such as `subseq s 3 7` does not give any clue whether the border elements 3 and 7 are included into the result or not, and it could just as well mean a subsequence starting at the third element and having at most seven elements.

On the other hand, a specifically designed notation such as `s[3..7]` seems to convey the intended meaning much more clearly, and by playing around with different combinations of square and round brackets according to mathematical habits, e. g., `s[3..7)` and `s(3..7]`, the inclusion or exclusion of the border elements can be expressed quite naturally, too.[5] Additional abbreviations such as `s[3..]` (meaning `s[3..#s]`), `s(..7)` (meaning `s(1..7)`), or even `s[..)` (meaning `s[1..#s)`) should also be rather self-explanatory. By combining the different notations, the subsequence starting at the third element and having (at most) seven elements can then be written as `s[3..][..7]`.

All the operators just introduced can easily be defined as syntactic wrappers of the original operator `subseq` _ _ _, e. g.:

```
["T":type; "s":T*; "i":int; "j":int]
s "[" i ".." j "]" : T*
{ subseq s i j };

["T":type; "s":T*; "i":int; "j":int]
s "[" i ".." j ")" : T*
{ subseq s i j-1 };

["T":type; "s":T*; "i":int]
s "[" i ".." "]" : T*
{ subseq s i #s };
```

3.3 Iteration over a Sequence

Many programming languages provide a tailored `for` statement to conveniently iterate over containers. In flexiPL, it is a simple exercise to define a corresponding user level operator `for _ in _ do _ end` possessing the following explicit parameters:

- a variable `var` with type `T?`, where `T` might be any type;
- a sequence `seq` with type `T*`;
- a loop body `body` with an arbitrary type `B`.

Consequently, the operator also needs deduced parameters `T` and `B` with type `type`. Its definition could read as follows:

```
[
  "T" : type; "B" : type;
  "var" : T?; "seq" : T*; "body" : B {}
]
"for" var "in" seq "do" body "end" : int
{
  "i" : int?;
  for i from 1 to #seq do
    var << seq[i];
    body
  end
};
```

The operator possesses result type `int`, since every operator must have a result type and, by convention, a loop returns the number of iterations it has performed. Therefore, the value returned by the (also user-defined) loop operator `for _ from _ to _ do _ end` can directly be used as the result value of the new operator.

The empty curly brackets at the end of the declaration of the parameter `body` indicate that the corresponding operand shall not as usual be passed by value, but rather as an *unevaluated expression*, similar to lazy evaluation in functional languages and “call by name” in Algol (cf. Sec. 2.1.3).

[5] This implies, by the way, that bracket symbols can be used just like any other character, in particular they are not required to be balanced.

The actual evaluation of the operand – whose effect typically depends on the current value of the variable var – is performed each time the parameter body is used (i. e., applied) inside the implementation of the for operator, i. e., once per iteration.

Frequently, list comprehensions are even more convenient to use than for loops, e. g.:

```
"persons" : person* = ...;
"p" : person?;
"long" "names" := select p.name from
  p in persons where #p.name >= 20 end;
```

Defining the necessary generic operator is also quite easy:

```
[
  "X" : type; "Y" : type;
  "var" : X?; "seq" : X*;
  "cond" : bool {}; "expr" : Y* {}
]
"select" expr "from" var "in" seq
"where" cond "end" : Y*
{
  "res" : Y*?;
  for var in seq do
    if cond then
      res << res, expr
    end
  end;
  res
};
```

3.4 Iteration over Multiple Sequences

It is a bit more difficult to generalize the for operator introduced in Sec. 3.3 to iterate over multiple sequences simultaneously, stopping at the end of the shortest one, e. g.:

```
"s1" : int* = ...; "v1" : int?;
"s2" : person* = ...; "v2" : person?;
for v1 in s1 and v2 in s2 do ... end
```

The particular challenge is to deal with the combination of a previously unknown number of sequences and the fact that they might have different types: An unknown number of homogeneous sequences could be captured with a sequence of sequences, while a fixed number of sequences with different types could be handled with a corresponding fixed number of deduced parameters. The solution presented in the sequel makes use of some interesting features of flexiPL not introduced so far, in particular local operators and closures.

As a preliminary, the following virtual operator (cf. Sec. 2.3.2) can be used to make available at compile time the type T of an arbitrary expression x:

```
["T" : type; "x" : T]
"typeof" x "end" = T;
```

For example, "x" : typeof 1 end would be equivalent to "x" : int, because the type of the subexpression 1 is int. A more useful application exploits the fact that a declaration returns the operator declared by it, i. e., its type is equal to the corresponding *operator type*. Therefore, the following declaration defines v as a variable whose value is an operator with parameter and result type int, similar to a function pointer variable in C:

```
"v" : typeof ["x" : int] "f" x : int end ?
```

As another preparation, it is usually helpful to phrase the envisaged syntactic form as a set of grammar productions in, e. g., BNF, which can be transformed to a set of operator declarations afterwards:

```
loop    : "for" inits "do" body "end"

inits   : init "and" init
        | inits "and" init

init    : var "in" seq
```

Here, body can be an arbitrary expression of some type B, while var and seq can be arbitrary expressions of type T? and T*, respectively, for some T, which might be different for each pair of var and seq.

To abstract from these different variable and sequence types, the auxiliary operator _ in _ takes a variable var and a sequence seq and returns a locally defined operator assign _ with a parameter i of type int and result type bool, i. e., the type of this operator does not depend on the type T. This operator can then be applied to an index i in order to assign the i-th element of seq to var (if such an element exists; otherwise, the operator does nothing and returns false). Since the operator _ in _ returns an operator, its result type must be an appropriate operator type that is obtained with the above operator typeof _ end and abbreviated with the name Init:

```
"Init" =
typeof ["i" : int] "assign" i : bool end;

["T" : type; "var" : T?; "seq" : T*]
var "in" seq : Init
{
  ["i" : int]
  "assign" i : bool
  {
    if i <= #seq then
      var << seq[i]; true
    else
      false
    end
  }
};
```

Because the operator _ in _ always returns an operator of type Init, no matter to which variable and sequence types it has been applied, the results of multiple – even heterogeneous – applications can now easily be joined together into a sequence of type `Init*` with the following operators _ `and` _:

```
["x" : Init; "y" : Init]
x "and" y : Init*
{ x, y };

["x" : Init*; "y" : Init]
x "and" y : Init*
{ x, y };
```

Therefore, an expression such as `v1 in s1 and v2 in s2 and v3 in s3` actually returns a sequence of three different `assign` _ operators, one for each variable/sequence pair.

After these preparations, the actual loop operator `for` _ `do` _ `end` can be defined as follows:

```
[
  "B" : type;
  "inits" : Init*; "body" : B {}
]
"for" inits "do" body "end" : int
{
  "i" : int?; i << 1;
  while
    "f" : bool?; f << true;
    "init" : Init?;
    for init in inits do
      ["i":int] "assign" i : bool = init;
      f << f && assign i
    end;
    f
  do
    body;
    i << i + 1
  end
};
```

The outer while loop (whose condition part is a more complex expression than usual) constitutes the actual iteration over the sequences, while the inner for loop (using the `for` operator for a single sequence defined in Sec. 3.3) iterates over the sequence `inits` in order to execute the `assign` _ operators it contains. For that purpose, the current value of the iteration variable `init` is used to initialize a local operator `assign` _ that is afterwards applied to the current index value `i`. The result values of these operators are conjunctively combined in the Boolean variable `f` in order to detect the end of either sequence and to terminate the outer loop.

3.5 Iteration over Other Containers

As another exercise, the previously developed `for` operator can be generalized to arbitrary container types `C` which provide the basic operators `#` _ to query their length and _ `[` _ `]` to obtain individual elements, by passing these operators as *implicit parameters* to the auxiliary operator _ `in` _ (which is the only operator that actually needs them):

```
[
  "T" : type; "var" : T?;
  "C" : type; "cont" : C;
  ["c" : C] "#" c : int;
  ["c" : C; "i" : int] c "[" i "]" : T
]
var "in" cont : Init
{
  ["i" : int]
  "assign" i : bool
  {
    if i <= #cont then
      var << cont[i]; true
    else
      false
    end
  }
};
```

The parameters `#` _ and _ `[` _ `]` do neither appear in the operator's signature (and therefore they are not explicit) nor in the type of other parameters (therefore they are not deduced). This third kind of parameters is called implicit.

When an operator with implicit parameters is applied, each of these parameters is automatically initialized with an operator that is (i) visible at the point of application, (ii) has the same syntax as the parameter, and (iii) is able to *implement* the parameter. Generally, an operator o_1 can be implemented by another operator o_2, if an application of o_2 to the explicit parameters of o_1 would be a correct expression whose type is equal (or implicitly convertible) to the result type of o_1. For example, any unary operator with parameter type `person*` and result type `int` can be implemented by the generic length operator `#` _ for sequences, because an application of this operator to an operand with type `person*` is a correct expression with type `int`.

If no suitable operator is found for an implicit parameter, the respective operator application is rejected by the compiler. This means, that implicit parameters can be used to specify constraints for deduced parameters, as in the example above: An application of _ `in` _ for particular types `T` and `C` will only be accepted by the compiler, if it finds visible operators `#` _ and _ `[` _ `]` that can implement the corresponding implicit parameters for these types.

If, for example, `v` is a variable with type `person?` and `s` is a sequence with type `person*`, the expression `v in s` is correct, because after deducing `T` and `C` as usual to be equal to `person` and `person*`, respectively, the implicit parameters `#` _ and _ `[` _ `]` can be implemented by the corresponding predefined sequence operators. If, however, `ls` is a list with type `[person]` (cf. Sec. 2.3.1), the expression `v in ls` is not correct, because after deducing `T` and `C` as `person` and `[person]`, respectively, neither an operator `#` _ with parameter type `C` (i. e., `[person]`) nor a suitable operator _ `[ _ ]` can be found. If these operators would be defined for lists, however, `v in ls` would become a correct expression and thus lists could be used in iterations just like sequences.

3.6 Matrices

To conclude the set of examples, an excerpt from a library of matrix operators shall be presented that can be used as follows:

```
"A" := float{2x3}(1.5, -2.7, ...);
"B" := float{3x4}(-3.2, 4.1, ...);
"C" := A B;
```

The matrices `A` and `B` are defined by enumerating their elements in row-major order (using ellipses for brevity), employing the predefined comma operator (cf. Sec. 3.1) to combine them into a sequence of type `float*`. Afterwards, matrix `C` is defined as the product of `A` and `B`, i. e., it will have type `float{2x4}`.

First of all, a generic type constructor for matrix types resembling the mathematical notation $T^{M \times N}$ has already been shown in Sec. 2.3.1:

```
["T" : type; "M" : int; "N" : int]
T "{" M "x" N "}" : type;
```

The elements of a matrix can be stored in row-major order in an attribute `elems` of type `T*` similar to Sec. 3.1:

```
[
  "T" : type; "M" : int; "N" : int;
  "A" : T{MxN}
]
A "." "elems" : T*?;
```

Then, the following operator _ `[` _ _ `]` provides direct access to an element A_{ij}:

```
[
  "T" : type; "M" : int; "N" : int;
  "A" : T{MxN}; "i" : int; "j" : int
]
A "[" i j "]" : T
{ A.elems[(i-1)*N + j] };
```

The definitions of `A` and `B` in the introductory example use the following matrix "constructor" _ `{` _ `x` _ `}` `(` _ `)`:

```
[
  "T" : type; "M" : int; "N" : int;
  "elems" : T*
]
T "{" M "x" N "}" "(" elems ")" : T{MxN}
{
  "A" : T{MxN}; A.elems << elems; A
};
```

Instead of enumerating the elements of a matrix, the following constructor allows to pass an operator `elem` _ _ that computes them, which is sometimes more convenient to use:

```
[
  "T" : type; "M" : int; "N" : int;
  "e" :=
  ["i" : int; "j" : int] "elem" i j : T
]
T "{" M "x" N "}" "(" e ")" : T{MxN}
{
  "elems" : T*?; "i" : int?; "j" : int?;
  for i from 1 to M do
    for j from 1 to N do
      elems << elems, elem i j
    end
  end;
  T{MxN}(elems)
};
```

The most interesting operator is the one for multiplying matrices `A` of type `T{LxM}` and `B` of type `T{MxN}`, yielding a result matrix of type `T{LxN}`, where `T`, `L`, `M`, and `N` are deduced from the types of `A` and `B`. To compute an element $C_{ik} = \sum_{j=1}^{M} A_{ij} \cdot B_{jk}$ of the result matrix C, it must be possible to multiply the elements A_{ij} and B_{jk} of type `T` and to add up the results, i. e., there must be suitable operators _ `*` _ and _ `+` _ for the type `T`. As explained in Sec. 3.5, these restrictions (basically requiring `T` to be a field) can be expressed by implicit parameters:

```
[
  "T" : type;
  "L" : int; "M" : int; "N" : int;
  "A" : T{LxM}; "B" : T{MxN};
  ["x" : T; "y" : T] x "*" y : T;
  ["x" : T; "y" : T] x "+" y : T
]
A B : T{LxN}
{
  T{LxN}(
    ["i" : int; "k" : int] "elem" i k : T
    {
      "sum" : T?;
      sum << (A[i 1] * B[1 k]);
      for j from 2 to M do
        sum << sum + (A[i j] * B[j k])
      end;
      sum
    }
  )
};
```

The operator's implementation simply calls the previously defined constructor with an appropriate local operator `elem` _ _ that computes C_{ik} from A_{ij} and B_{jk}.

It should be noted, that several of the above operator implementations explicitly use some of their deduced parameters whose values are determined from the types of the involved matrices.

For an application of the multiplication operator such as `A B` to be type-correct, the following conditions must be fulfilled:

- It must be possible to consistently deduce the values of the parameters `T`, `L`, `M`, and `N` from the types of the explicit operands `A` and `B`, i. e., both types must be matrix types with the same element type `T`; furthermore, the column count of `A` must be equal to the row count of `B`, because both are denoted by the same parameter `M`. In the specific example, the deduced values will be `float`, `2`, `3`, and `4`.
- At the point of application, there must be visible operators `_ * _` and `_ + _` corresponding to the operator's implicit parameters, whose parameter and result types must all be equal to the type that has been deduced for `T`, i. e., `float` in the example. In fact, there are predefined operators for multiplying and adding `float` values that can be used for that purpose.

4. Technology

4.1 Compiler

flexiPL programs are parsed, type-checked, and translated to simple-minded C++ code by a compiler that is also written in C++. Using a high-level target language is both easier and more portable than generating assembler code and allows low-level optimizations to be performed by the subsequently running C++ compiler. The main reason for choosing C++ instead of C is a more powerful standard library providing, e. g., dynamically growing vectors (used to implement sequences) and associative maps (used to implement the lookup tables required by static operators). Both the compiler and the target code use the Boehm-Demers-Weiser conservative garbage collector [4] to simplify dynamic storage management.

4.1.1 Parser and Type Checker

Figure 1 sketches the structure of the recursive-descent parser constituting the backbone of the compiler front-end.

Most parse functions receive as parameters the current position in the input stream and the set of all operators that are visible at this place. In the beginning, this is the set of all predefined operators, which can successively be extended with user-defined operators by calls to `create_oper` (cf. Sec. 2.4). On the other hand, the set of visible operators can be locally restricted by operator-specific import and export declarations (cf. Sec. 2.2).

Because in general there might be multiple possible interpretations of an input sequence, some of which might get ruled out later for various reasons (e. g., typing errors or violation of an exclude declaration, cf. Sec. 2.1.1), most functions return a set of expressions (i. e, operator applications) constituting possible decompositions of the subsequent input.

Parsing begins with the parameterless function `parse_top` which in turn calls `parse_all` with input position 1 and the set of predefined operators. `parse_all`'s job is to determine and return *all* possible interpretations of the subsequent input. For that purpose, it calls `parse_one` once for each visible operator, which determines and returns all possible applications of this *single* operator. To achieve that, the operator's syntax is traversed from left to right and for each of its parts one of the functions `parse_name` (if the part is a name of the operator) or `parse_opnd` (if it is a placeholder for an operand) is called. `parse_name` simply checks whether the operator name appears at the respective input position after discarding white space and comments if necessary. (That is, `parse_name` is actually a degenerate scanner.) If it does not, processing of the operator in `parse_one` can be aborted immediately.

`parse_opnd`'s job is to extend the partial expression constructed by `parse_one` so far with another operand. For that purpose, `parse_all` is called recursively to determine the set of basically possible operands at the respective input position. For each expression returned by this function, the type checking function `check` is called to test whether the expression's type matches the type of the corresponding explicit parameter. If this is the case, the expression is added as an operand to a copy of the partial expression. Finally, `parse_opnd` returns the set of all expressions extended that way.

If `parse_one` has successfully parsed a complete operator application, it finally checks whether the operator's implicit parameters can be initialized with suitable visible operators, again using the type checking function `check`. If the type of an explicit or implicit parameter contains deduced parameters, their values are deduced as a side effect during the type checks performed by this function. Finally, if the expression is a declaration, `create_oper` is called to create the operator declared by it.

Taken together, these functions implement a recursive backtracking algorithm that determines all possible decompositions of the input into type-correct expressions. If there is exactly one such decomposition in the end, the input is accepted as a correct program and the back-end is called to generate target code for it. Otherwise, the program is rejected because it is either erroneous or ambiguous. The latter might happen in particular if the precedence between operators has not been specified sufficiently.

To improve the performance of the backtracking algorithm, erroneous decompositions are detected and ruled out as soon as possible by integrating the type checker within the parser. Furthermore, repeated calls to `parse_all` with the same arguments (input position and visible operators), which are frequently generated by the backtracking strategy, are optimized by memoizing their results. By that

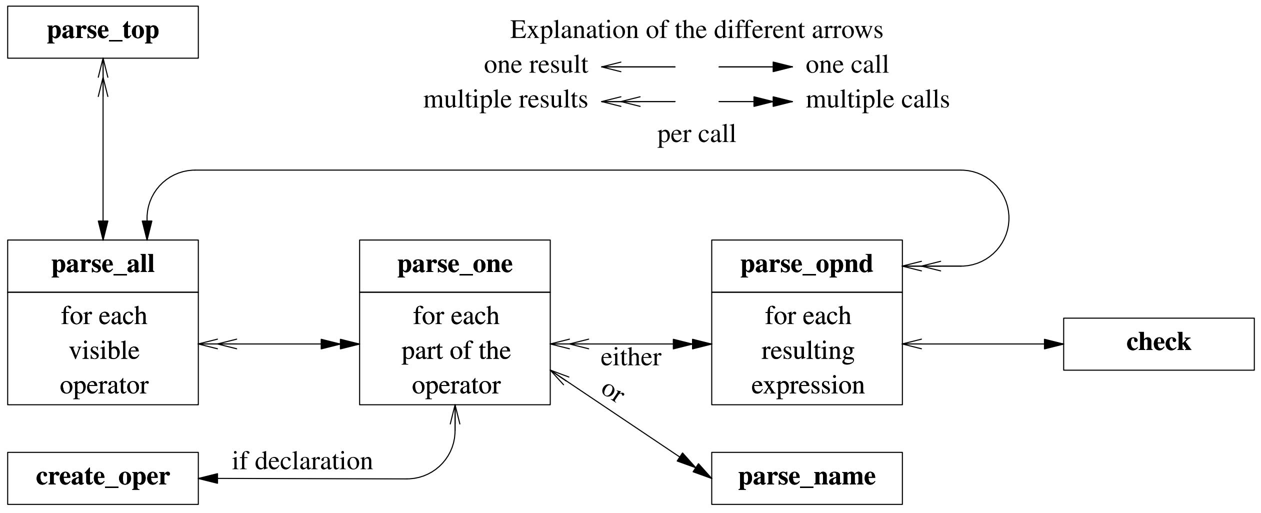

Figure 1. Functional structure of the parser

means, the compiler's run time usually grows only linearly with the size of the input.

4.1.2 Error Handling

Not surprisingly, the issues of error diagnosis and recovery are inherently difficult for a syntactically extensible programming language. Because there is no predefined grammar at all, established techniques such as resuming at particular "recovery symbols" (e. g., keywords such as `else` or `end`) or augmenting the grammar with special "error productions" do not work. Furthermore, if the compiler does not find a correct interpretation of the input, it is difficult to produce an appropriate error message, because usually several alternatives have been tried where each of them might have failed for a different reason. (This in turn complicates error recovery, because it also has to pursue multiple alternatives.) On the other hand, detecting that a particular decomposition of (a part of) the input does not work, does not necessarily imply an error that should be diagnosed, because a different decomposition might well work.

Therefore, the currently implemented error diagnosis is rather simple-minded: If the input cannot be decomposed into a complete correct expression, the highest input position up to which a successful parse of correct subexpressions has been possible is printed and the parse is aborted. Usually, this position is a good approximation of the actual position of the first error, and with a little training a programmer can quickly identify its reason.

For future versions of the compiler, however, more sophisticated error diagnosis and recovery is highly desirable in order to support more productive programming. A reasonable strategy might be to record an appropriate error message for each failed attempt detected in the function `parse_one`. If a correct decomposition is found in the end, these messages can simply be discarded; otherwise, it might be reasonable to print them all, each with an identification of the operator whose application failed and therefore caused the message. For error recovery, an adaptation of the idea of recovery symbols might work: For each currently active invocation of `parse_one`, the operator symbol expected after the operand currently being parsed by `parse_opnd` might be used as a recovery symbol at which the respective invocation of `parse_one` can try to continue if `parse_opnd` does not return any correct operand.

If a program is ambiguous, it is even more vital to get hints from the compiler about the cause, because otherwise it is extremely hard to find it. Usually, a programmer is mentally tied to his intended interpretation of the input and thus hardly recognizes other possibilities. Furthermore, a "global" ambiguity of the whole program is always caused by a "local" ambiguity of a particular subexpression that cannot be resolved later, and finding this particular subexpression in a large program is almost impossible without specific help.

To assist the programmer in such cases, the compiler produces an HTML representation of the input whose outline is identical to the source code. When the mouse pointer is positioned on an operator name, the entire operator application is highlighted by displaying the operands with alternating colors, and a tool tip shows the operator's signature and the line number where it is declared. Furthermore, the corresponding declaration is also highlighted with a different color. This augmented source code representation is useful even for a program without errors or ambiguities to explore its logical structure and to see which operator is applied at which input position. For an ambiguous program, all operator names belonging to ambiguous operator applications are immediately highlighted in red. When positioning the mouse pointer on such a name, the tool tip shows the signatures and line numbers of all the operators that could be applied here. Alternatively, if an ambiguity is not caused by overloaded operator names, but by insufficient operator precedences, the tool tip shows the different groupings that are possible at this place.

4.2 Current Limitations

After completion of the first, albeit partially incomplete, compiler it has been possible to gain practical experience with using the language. Not really surprisingly, this has revealed some conceptual and technical flaws which shall be remedied in the future.

4.2.1 User-Defined Declaration Operators

The most severe limitation at the moment is the fact that user-defined declaration operators (i. e., user-defined operators whose application causes declarations of other operators, cf. Sec. 2.4) are very restricted at the moment, because their parameters can neither appear in the signature nor in the parameter list of the declarations they perform. For example, it would be desirable to define a new operator, say `_ :<< _`, that combines a variable declaration with an assignment of its initial value, i. e., `"x" :<< 1` should be equivalent to `"x" : int?; x << 1` (the type `int` can be deduced from the initial value `1`). Currently, this is impossible for two reasons: First, the names appearing in an operator's signature must be string literals and thus cannot depend on parameters of the enclosing (virtual) declaration operator. (This means that the example given in Sec. 2.4 does not work either at the moment.) This restriction could be relaxed rather easily in the compiler, but the second reason is more fundamental: If an operator's name depends on a parameter, it is difficult to apply this operator afterwards, because its actual name is not known, for example:

```
["T" : type; "name" : string; "init" : T]
name ":<<" init = ...
```

This operator's realization (indicated by the ellipses) could start with the declaration `name : T?` that declares a variable of type `T?` whose name is given by the value of the parameter `name`. (Therefore, `name` is not enclosed in quotation marks.) But afterwards it is impossible to assign to this variable within the realization, because its actual name is not known there. (`name << init` would of course not be a correct expression, since `name` denotes a string instead of a variable.)

In this particular example, it is possible to circumvent this general problem by first defining and assigning an auxiliary variable with a fixed name and afterwards defining the actually required variable as an "alias" for it (the operator `use _ in _ end` has been defined in Sec. 2.2; `! "x" : T?` is a *redeclaration* that simply returns the previously declared operator `x`):

```
["T" : type; "name" : string; "init" : T]
name ":<<" init =
use
  "x" : T?; x << init
in
  name : T? = ! "x" : T?
end
```

But things get even more difficult, if the parameter list of an operator declaration shall also depend on parameters of the enclosing declaration operator. A typical example for such a declaration operator is a wrapper for one of the predefined declaration operators, which provides exactly the same functionality with different syntax. It might be desirable, for example, to declare parameterized operators with a new declaration operator `with _ define _ : _ begin _ end` that is equivalent to `[ _ ] _ : _ { _ }`, e. g.:

```
with
  "n" : int
define
  n "!" : int
begin
  if n > 1 then (n-1)! * n else 1 end
end
```

Basically, the definition of the new declaration operator must read as follows:

```
[
  "Pars" : type; "pars" : Pars;
  "Sig" : type; "sig" : Sig;
  "res" : type;
  "impl" : res
]
"with" pars "define" sig ":" res
"begin" impl "end" =
[pars] sig : res { impl }
```

Since the actual types of the parameter list `pars` and the signature `sig` may vary for different applications of the operator, they are declared as deduced parameters of type `type`. The compiler will not accept the realization of this operator, however, since it does not know anything specific about these two essential parts of the operator declared in the realization by the predefined declaration operator `[ _ ] _ : _ { _ }`.

There are specific ideas to solve this apparently intractable problem, which have to be worked out in detail. As the previous example demonstrates, the problem has significant practical relevance, because in a fully extensible and customizable language it should of course be possible to extend or customize even the syntax of the most fundamental predefined declaration operators in order to be able to change the syntax that is used to define new syntax.

As another practical example, the operators defined in Secs. 3.3 to 3.5 would be even more convenient to use if the loop variables could be declared by the operator itself, e. g.:

```
for "x" in 1, 2, 3 do
  print x   ← use x as if it has been declared as
end             "x" : int = (current iter. elem.)
```

However, this has non-trivial implications for the concept of visibility rules, which must also be worked out in detail.

4.2.2 Implicit Type Conversions

Implicit type conversions (one of the special cases of multi-part operator combinations in Sec. 2.1.2) are also rather limited at the moment. Even though their most useful application, i. e., the implicit conversion of a variable to its value (cf. Sec. 2.1.3), works fine, there are other practical applications that do not work satisfactorily at the moment.

If, for example, an `int` value is implicitly convertible to the corresponding `float` value, a simple addition of `int` values might also be interpreted as the addition of the corresponding `float` values. But, of course, it seems natural to automatically resolve this ambiguity in favor of the integer addition in that case. Therefore, it is necessary to define general rules to automatically resolve such "artificial" ambiguities.

As another example, if a value of some type `X` is implicitly convertible to a sequence with type `X*` containing just this value, this conversion might be applied repeatedly, yielding a sequence of sequences with type `X**`, a threefold sequence with type `X***`, and so on. The question when to stop these repeated applications because they do no longer make sense is difficult to answer in general when there are other implicit conversions that could be applied, too.

A general algorithm to decide whether a given source type is implicitly convertible to a target type by a finite sequence of individual conversions has been developed and successfully tested with various real and artificial examples, but before integrating it into the compiler its correctness and termination should be formally proven.

5. Related Work

5.1 Basic Language Features

Apart from the syntactical extensibility discussed separately in Sec. 5.2, flexiPL provides several other interesting language features, which shall be reviewed in the sequel.

Deduced parameters are well-known under different names from many other languages, e. g., template parameters in C++ and type variables in Java and Haskell. In contrast to these languages, however, types are first-class objects in flexiPL and deduced parameters are declared and used just like other parameters.

Implicit parameters have also been proposed earlier, both by the author himself [10] and others [13]. In a somewhat different form, they can also be found in Scala [17]. As already pointed out in Secs. 3.5 and 3.6, they are not merely a matter of convenience – because their values need not be given explicitly when applying an operator –, but they are typically employed to specify restrictions or constraints for deduced parameters, similar to bounded polymorphism as, e. g., in Java and type classes as, e. g., in Haskell.

The possibilities for arbitrarily nesting operator definitions, where local operators can safely use more global ones, passing operators as parameters and results of other operators etc. are well-known as closures and higher-order functions.

Static operators as described in Sec. 2.3.1, in particular parameterized ones, as a means to establish immutable relationships between objects and to implicitly create the required objects on demand seem to be a new concept for a programming language.

The approach to provide mutable storage or state as an orthogonal concept of its own by making variable types (`T?`) explicit, is also rather unusual. By omitting these types and their accompanying operators, the language would immediately turn into a purely functional one.

User-definable implicit type conversions (even generic ones to some extent) are provided in some way or another by several programming languages including C++, Scala, and others, but they are not applied transitively there. Furthermore, predefined and user-defined conversions are strictly separated and subject to different rules. (For example, conversions implied by subclass relationships are in fact applied transitively, of course.) According to flexiPL's general principle "one law for all," user-defined implicit conversions are treated just like predefined ones, leading to much simpler and more uniform rules for their application.

The possibility to define and use types that depend on non-type values is known as dependent types. However, the precise expressiveness of flexiPL's type system and its relationship to other systems has not been investigated yet. It is basically possible, however, to use type definitions and virtual operators to encode computations that will be carried out at compile time, similar to, e. g., template metaprogramming in C++.

5.2 Extensible Programming Languages

During the history of programming language development, the idea of an extensible programming language has appeared every now and then.

One of oldest and most well-known examples is Lisp [18] with its different dialects and flavors. Similar to flexiPL, Lisp does neither distinguish between operators and functions nor between predefined and user-defined operators/functions. By defining new functions – or macros, whose syntactic appearance is identical to that of functions – a programmer is actually extending the language all the time. Another parallel to flexiPL is the fact that language extensions are defined in the language itself, and that a very small language core is sufficient for that purpose. However, there are also essential differences: First of all, Lisp does not possess a static type system. Furthermore, Lisp expressions must always be parenthesized, which significantly restricts the possibilities for defining new syntax. Finally, flexiPL does not have a "procedural" macro engine, i. e., no user code will be executed at compile time in order to perform syntactic transformations. (Virtual operators are just a declarative and type-safe "rewrite" macro system.) In summary, flexiPL has considerable advantages over Lisp (complete syntactic freedom and static type safety), while the deliberately omitted procedural macro facility has not been perceived as a major limitation yet.

Dylan [7] is a more modern language that has been strongly influenced by Lisp's ideas. It also supports syntactic extensibility in the language itself (actually in a rewrite macro system which is an integral part of the language). Even though the programmer has more freedom than with Lisp's simple s-expressions, there are also strict syntactic limitations which cannot be exceeded. In contrast, the operator concept of flexiPL offers virtually unlimited syntactic freedom. Apart from that, Dylan does not have a static type system either.

D-Expressions [1] extend Dylan's rewrite macro system with a procedural macro engine that allows to use the full expressive power of the programming language in syntax transformations. The Java Syntactic Extender (JSE) [2] applies the same idea to the Java programming language. But despite the unlimited expressiveness of syntax *transformations*, both approaches severely restrict the possibilities for their *application* in the same way as Dylan does. For example, JSE distinguishes call and statement macros, both of which are "limited to a few contexts and shapes corresponding to existing Java syntactic contexts and shapes," which allows the parser to perform an initial "skeletal parse" of the input without knowing the actual set of macro definitions. For other syntactic constructs, especially method, field, and variable declarations, it is judged that "it's practically impossible to allow programmers to introduce constructs of similar status in a modular way, and no attempt is made to address this." flexiPL, however, does in fact address exactly this.

Maya [3] is another interesting approach to syntax extensions in Java, where grammar productions are treated as compile-time generic functions and semantic actions as corresponding multimethods. In contrast to flexiPL, however, extensibility is provided as a separate add-on (called Mayans) whose appearance (quasiquoted templates) is rather different from that of the base language. Furthermore, Mayans have to be precompiled separately from the applications using them. Even though the syntactic flexibility seems to be much higher than with JSE, there are still limitations compared with flexiPL, e. g., brackets must always be balanced to allow "lazy parsing," similar to JSE's "skeletal parsing."

Many different languages, e. g., Haskell [14], Prolog [6], and Scala [17], allow the user to extend at least the syntax of expressions by defining new operator symbols. Since functional languages, just as flexiPL, do not distinguish between expressions and statements, the syntax of statements (e. g., control structures) becomes also extensible in principle. However, the syntax of types and declarations still remains fixed. In flexiPL, however, the basic principle "*everything* is an expression" implies that *all* parts of the language can be extended simply by defining new operators.

Seed7 [15] is a rather old approach of a statically typed, extensible programming language which is still actively maintained, however [16]. The language is divided into two distinct levels causing an unnatural and unnecessary breach between "primitive actions" and special syntax declarations on the one hand and normal operators on the other hand. In contrast, irreducible core constructs are provided as normal operators in flexiPL, whose usage does not differ in any way from other operators. Furthermore, the definition of a new operator – which simultaneously defines a new syntactic construct – is an operator application itself. By that means it is basically possible to extend or customize even the syntax that is used to define new syntax.

Ganz et al. [8] provide seminal work for augmenting functional languages with type-safe macros, which might even define new binding constructs, by formally viewing macros as multi-stage computations. To illustrate "how the main semantic subtleties of a typed macro system can be addressed," (Core) MacroML is presented as an extension of ML, that does not strive for great syntactic flexibility, however. In fact, the addition of distfix macros as a means for syntax customization is only mentioned briefly as a possible extension.

An approach whose basic ideas and objectives are almost identical to that of flexiPL is "π – a Pattern Language" [12]. The concept called pattern there – which is "the only language construct in π" – directly corresponds to an operator in flexiPL: It possesses a syntax, composed of names (or symbols) and placeholders for operands, and an associated meaning corresponding to the implementation of a flexiPL operator. Thus, both approaches provide the same virtually unlimited syntactic flexibility that ultimately stems from the lack of any predefined grammar.

A significant difference and advantage of flexiPL over π is once again the static type system, since π is completely dynamically typed. In fact, the endeavour to reconcile extreme flexibility on the one hand with a maximum of static checkability on the other hand has been and still is the most ambitious challenge in the development of flexiPL.

Apart from that, flexiPL provides several other useful facilities not found in π, e. g., implicit and deduced parameters (where the latter are dispensable in a dynamically typed language) or import, export, and exclude declarations which allow, amongst others, user-defined scoping rules and locally confined syntax extensions.

Finally, flexiPL might also be considered an adaptive grammar formalism [5, 19]. Because "everything is an expression," there is a single non-terminal symbol `X` denoting expressions. Every operator declaration induces a new production for `X` whose right hand side can be derived from the operator's signature by treating the operator's names as terminal symbols and replacing explicit parameters with the non-terminal `X`. The type information associated with the parameters and the result type of the operator can be added as grammar attributes. Import and export declarations control the set of currently active productions, while exclude declarations can be used to rule out some otherwise possible derivations.

6. Conclusion

MOSTflexiPL is a programming language currently under development whose syntax can be extended and customized by its users in a virtually unlimited way, where a rather small number of core constructs is sufficient to support a broad range of different programming styles. Therefore, it can be used amongst others as an extensible general purpose programming language and as a host language for domain-specific languages. It possesses a static type system and is translated by a compiler to C++.

The most severe limitations that shall be remedied in the future have already been mentioned in Sec. 4.2. Other conceptual improvements of the language include the support for user-defined literals, white space, and comments, because these language features are currently hard-wired in the language definition and the compiler. To support the definition of advanced control structures, a general jump mechanism is required which can be used to implement specific statements such as `break`, `return`, and `throw`. A concept for modules and separate compilation has been designed (and therefore the term "modular" is already part of the language's name), but not implemented in the compiler yet. The issues of error diagnosis and recovery have already been discussed in Sec. 4.1.2.

Apart from these conceptual improvements, several aspects of the flexiPL compiler have to be enhanced, too. Amongst others, the search for deduced parameter bindings is incomplete if these parameters have parameters themselves (which is not needed very often, however) and the search for implicit parameter bindings might run into an infinite recursion for some "malicious" (and rather artificial) examples. Furthermore, both the performance of the compiler itself and that of the generated code can be improved significantly, for example by inlining applications of frequently used predefined operators (e. g., for arithmetic). While some of these optimizations can be automatically performed by the subsequently executed C++ compiler, others must be done by the flexiPL compiler back-end, because, e. g., functions that are indirectly called via function pointers cannot be inlined by the C++ compiler. Finally, it is planned to develop additional compiler back-ends to support other target languages and architectures, e. g., Java source or byte code and the LLVM compiler infrastructure.

Simultaneously, the practical experiences with flexiPL shall be broadened, e. g., by developing more extensive operator libraries and by employing the language in real-world programming projects. As a long term goal, the compiler shall ultimately be implemented in the language itself.

Acknowledgements

Marco Perazzo, Stefan Billet, and Miriam Klement have done an excellent job in implementing major parts of the MOSTflexiPL compiler. Sean McDirmid has provided lots of valuable comments for improving the presentation of the paper. Last but not least: Soli Deo Gloria!

References

[1] J. Bachrach, K. Playford: *D-Expressions: Lisp Power, Dylan Style*. Technical Report, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1999.

[2] J. Bachrach, K. Playford: "The Java Syntactic Extender (JSE)." In: *Proc. 2001 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '01)* (Tampa Bay, FL, October 2001). *ACM SIGPLAN Notices* 36 (11) November 2001, 31–42.

[3] J. Baker, W. C. Hsieh: "Maya: Multiple-Dispatch Syntax Extension in Java." In: *Proc. 2002 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Berlin, Germany, June 2002). *ACM SIGPLAN Notices* 37 (5) May 2002, 270–281.

[4] H. Boehm, M. Weiser: "Garbage Collection in an Uncooperative Environment." *Software—Practice and Experience* 18 (9) September 1988, 807–820.

[5] H. Christiansen: "A survey of adaptable grammars." *ACM SIGPLAN Notices* 25 (11) November 1990, 35–44.

[6] W. F. Clocksin, C. S. Mellish: *Programming in Prolog* (Fourth Edition). Springer-Verlag, Berlin, 1994.

[7] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[8] S. E. Ganz, A. Sabry, W. Taha: "Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML." In: *Proc. Int. Conf. on Functional Programming* (Firenze, Italy, September 2001). *ACM SIGPLAN Notices* 36 (9) September 2001, 74–85.

[9] C. Heinlein: "C+++: User-Defined Operator Symbols in C++." In: P. Dadam, M. Reichert (eds.): *INFORMATIK 2004 – Informatik verbindet. Band 2* (Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e. V.; September 2004; Ulm). Lecture Notes in Informatics P-51, Gesellschaft für Informatik e. V., Bonn, 2004, 459–468.

[10] C. Heinlein: "Implicit and Dynamic Parameters in C++." In: D. Lightfoot, C. Szyperski (eds.): *Modular Programming Languages* (Joint Modular Languages Conference, JMLC 2006; Oxford, England, September 2006; Proceedings). Lecture Notes in Computer Science 4228, Springer-Verlag, Berlin, 2006, 37–56.

[11] C. Heinlein: "Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance." *Journal of Object*

Technology 6 (3) March/April 2007, 101−151,
http://www.jot.fm/issues/issue_2007_03/article3.

[12] R. Knöll, M. Mezini: "$\pi-$ a Pattern Language." In: *Proc. 24th Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)* (Orlando, FL, October 2009). *ACM SIGPLAN Notices* 44 (10) October 2009, 503−521.

[13] J. R. Lewis, M. B. Shields, E. Meijer, J. Launchbury: "Implicit Parameters: Dynamic Scoping with Static Types." In: *Proc. 27th ACM Symp. on Principles of Programming Languages* (Boston, MA, January 2000), 108−118.

[14] S. Marlow (ed.): *Haskell 2010 Language Report*. HaskellWiki, 2010.
http://haskell.org/definition/haskell2010.pdf

[15] T. Mertes: *Definition einer erweiterbaren höheren Programmiersprache*. Dissertation, Technische Universität Wien, 1986. (in German)

[16] T. Mertes: *Seed7. The Extensible Programming Language*. http://seed7.sourceforge.net (2012-07-28).

[17] M. Odersky: *The Scala Language Specification* (Version 2.9). Programming Methods Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), May 2011.

[18] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

[19] Wikipedia Contributors: *Adaptive Grammar*.
http://en.wikipedia.org/w/index.php?title=Adaptive_grammar&oldid=496772431 (2012-07-28)