MOSTflexiPL
Modular, Statically Typed, Flexibly Extensible Programming Language

Christian Heinlein

Hochschule Aalen − Technik und Wirtschaft
christian.heinlein@htw-aalen.de

Abstract. MOSTflexiPL (oder kurz flexiPL) ist eine momentan in Entwicklung befindliche Programmiersprache, die vom Anwender nahezu beliebig syntaktisch erweitert und angepasst werden kann. Trotz dieser enormen Flexibilität besitzt die Sprache ein statisches Typsystem mit Ähnlichkeiten zu „dependent types“. Die Semantik neu definierter Sprachkonstrukte wird durch eine Abbildung auf bereits vorhandene Konstrukte in der Sprache selbst festgelegt, d. h. es sind weder Eingriffe in Compiler oder Laufzeitsystem noch ein „prozedurales“ Makrosystem wie in Lisp erforderlich. Der Sprachkern, d. h. die Menge der Grundkonstrukte, die sich nicht (sinnvoll) auf andere Konstrukte zurückführen lassen, folgt keinem bestimmten Programmierparadigma. Die meisten Grundkonstrukte sind funktionaler Natur, durch die Bereitstellung von Variablen (d. h. änderbarer Speicherzellen) wird aber auch imperatives Programmieren (im weitesten Sinne) unterstützt. Neben diesen Grundkonstrukten, gibt es eine Sammlung vordefinierter Standardkonstrukte zur Unterstützung unterschiedlicher Programmierstile, die bereits in der Sprache selbst geschrieben sind. MOSTflexiPL-Programme werden durch einen Compiler in assemblerartigen C++-Code übersetzt, der von jedem standardkonformen C++-Compiler in ausführbaren Code übersetzt werden kann. Wenn ein Programm aus mehreren Modulen besteht, können diese unabhängig voneinander übersetzt werden.

1 Einleitung und Motivation

Programmiersprachen werden in der Regel langsam, aber kontinuierlich weiterentwickelt. Beispielsweise wurde die ursprüngliche Sprache C von Kernighan und Ritchie aus den 1970er Jahren zu ANSI-C (1989) weiterentwickelt, aus dem wiederum in mehreren Schritten C99 (1999) hervorging. Ebenso wurde die ursprüngliche Sprache Java von 1995, nach zahlreichen kleineren Änderungen, im Jahr 2004 um wesentliche Sprachmittel erweitert, z. B. um „Generics“, Annotationen und Aufzählungstypen. Auch eine spezielle `for`-Schleife zur bequemeren Iteration über Arrays und Container wurde hinzugefügt. Auf manche dieser Verbesserungen haben Benutzer der Sprache jahrelang gewartet, weil sie die Programmierung z. T. erheblich erleichtern.

Aber selbst wenn man jeweils die aktuellste Version einer Sprache verwendet, kommt man als Programmierer immer wieder in Situationen, in denen ein maßgeschneidertes Sprachkonstrukt die tägliche Arbeit erleichtern und die Lesbarkeit des Codes verbessern könnte. Beispielsweise enthalten Deklarationen von Variablen mit sofortiger Initialisierung häufig einen hohen Grad von Redundanz (der sich durch Ver-

wendung von `import`-Anweisungen ein wenig reduzieren ließe), z. B.:

```
java.util.HashMap<String, java.io.InputStream> m =
  new java.util.HashMap<String, java.io.InputStream>();
```

Durch eine neue Deklarationsform (wie sie seit längerem für C++ geplant ist) könnte dies wie folgt verkürzt werden, indem eine Variable automatisch den Typ ihres Initialisierungsausdrucks erhält:

```
auto m =
  new java.util.HashMap<String, java.io.InputStream>();
```

Aber selbst wenn dieses Sprachkonstrukt eines Tages in die jeweiligen Sprachen integriert werden sollte, wird man als Programmierer früher oder später wieder in Situationen geraten, in denen man sich irgendein zusätzliches Sprachkonstrukt wünscht. Keine noch so gut entworfene Sprache wird jemals alle Anwender rundherum zufriedenstellen können.

Da man als normaler Programmierer mit gängigen Programmiersprachen nicht in der Lage ist, neue Sprachkonstrukte selbst zu definieren, muss man sich entweder mit dem Status quo arrangieren bzw. auf die nächste Sprachversion warten oder aber mit Hilfe eines Präcompilers seine eigene erweiterte Version der Sprache implementieren (wie z. B. in [He07]), was je nach Komplexität der Sprache und der gewünschten Erweiterungen mit erheblichem Aufwand verbunden sein kann.

Ein möglicher Ausweg aus diesem Dilemma ist die Verwendung einer Programmiersprache wie MOSTflexiPL[1], die von ihren Anwendern beliebig syntaktisch erweitert und angepasst werden kann. Sie bietet als Sprachkern eine möglichst kleine Menge von *Grundkonstrukten* (wie z. B. arithmetische und logische Operatoren sowie elementare Kontrollstrukturen), mit deren Hilfe sich beliebige weitere Sprachkonstrukte definieren lassen. Zusätzlich werden häufig benötigte *Standardkonstrukte* angeboten (z. B. weitere Kontrollstrukturen), die aber bereits als Erweiterungen des Sprachkerns in der Sprache selbst geschrieben sind.

Unter Verwendung dieser Grund- und Standardkonstrukte kann man in MOSTflexiPL bereits wie in jeder anderen Sprache programmieren, hat aber jederzeit die Möglichkeit, zusätzliche Sprachkonstrukte zu definieren, die z. B. die Programmierung einer bestimmten Anwendung erleichtern. Neue Sprachkonstrukte, die über eine bestimmte Anwendung hinaus nützlich erscheinen (wie z. B. die oben erwähnte `auto`-Deklaration), kann man − analog zu Funktions- oder Klassenbibliotheken in anderen Sprachen − zu einer Bibliothek zusammenfassen, um sie in anderen Anwendungen wiederverwenden oder anderen Anwendern zur Verfügung stellen zu können.

Ein weiteres Anwendungsfeld ist die Definition problemorientierter Sprachen (domain-specific languages, DSLs) als Erweiterungen bzw. Modifikationen von MOSTflexiPL, d. h. die Verwendung von MOSTflexiPL als Wirtssprache für andere Sprachen. Da die syntaktischen Gestaltungsmöglichkeiten nahezu unbegrenzt sind, entstehen bei dieser Einbettung einer DSL in MOSTflexiPL kaum Nachteile gegenüber der

[1] Oder kurz „flexiPL“, das wie das englische „flexible“ mit hartem p statt weichem b ausgesprochen wird. Der ursprüngliche Name lautete STEEL − Statically Typed Extensible Expression Language, was ebenfalls wichtige Eigenschaften der Sprache zum Ausdruck bringt. Der neue Name sowie das im Titel verwendete Logo betonen aber noch mehr die flexible Erweiterbarkeit und Anpassbarkeit der Sprache.
Siehe auch `http://most.flexipl.info` bzw. `http://flexipl.info`.

wesentlich aufwendigeren Implementierung von Grund auf durch maßgeschneiderte Compiler, Interpreter o. ä. Prinzipiell sollte es möglich sein − obwohl das nicht das Ziel der Entwicklung ist −, nahezu beliebige andere Computersprachen in MOSTflexiPL einzubetten.

Schließlich kann MOSTflexiPL als Experimentierfeld für neue Sprachkonstrukte (z. B. im Bereich parallele oder natürlichsprachliche Programmierung) dienen. Da die Implementierung neuer Sprachkonstrukte vergleichsweise einfach ist, kann man sich als Entwickler vor allem auf ihren Entwurf konzentrieren. Außerdem lassen sich Spracherweiterungen leicht und schnell in Form von Bibliotheken an interessierte Nutzer bzw. „Tester“ weitergeben, deren Erfahrungen und Rückmeldungen anschließend zur Weiterentwicklung und Verbesserung verwendet werden können.

Neben diesen Vorteilen einer syntaktisch erweiterbaren Sprache, besteht natürlich auch die Gefahr des „Missbrauchs“, indem jeder Programmierer gedankenlos unsinnige Erweiterungen definiert, die dazu führen, dass sein Quellcode für andere Personen kaum noch verständlich ist. Diesem „Wildwuchs“ kann jedoch durch Richtlinien oder Absprachen begegnet werden, die ähnlich wie heutige Kodierrichtlinien für ein Projekt oder eine Firma vereinbart werden können.

Der vorliegende Artikel ist wie folgt gegliedert. Abschnitt 2 erläutert anhand einfacher Beispiele in aller Kürze die wichtigsten vordefinierten Sprachkonstrukte von MOSTflexiPL. Abschnitt 3 zeigt dann exemplarisch einige Beispiele nützlicher Spracherweiterungen. In Abschnitt 4 werden die wesentlichen Ideen zur Implementierung von MOSTflexiPL skizziert. Verwandte Arbeiten werden in Abschnitt 5 diskutiert. Abschnitt 6 gibt schließlich eine Zusammenfassung sowie einen Ausblick auf zukünftige Arbeiten. Um dem Leser einen Gesamteindruck von MOSTflexiPL vermitteln zu können, musste an vielen Stellen aus Platzgründen auf detailliertere Ausführungen verzichtet werden.

2 Vordefinierte Sprachkonstrukte

2.1 Operatoren, Ausdrücke und Programme

Ein *Operator* besteht aus einem oder mehreren *Namen* (Operatorsymbolen) und kann auf eine bestimmte Anzahl von *Operanden* angewandt werden. Abhängig von der Anzahl der Operanden unterscheidet man null-, ein-, zweistellige Operatoren etc., wie zum Beispiel (die Positionen, an denen die Operanden einzusetzen sind, sind durch Unterstriche gekennzeichnet):

- Der Subtraktionsoperator `_ - _` ist zweistellig und besitzt einen Namen.
- Der Vorzeichenoperator `- _` ist einstellig und besitzt ebenfalls einen Namen.
- Der Operator `_ ? _ : _` in C, C++ und Java ist dreistellig und besitzt zwei Namen.
- Der Klammerungsoperator `( _ )` ist einstellig und besitzt zwei Namen.
- Der Schleifenoperator `while _ do _ end` ist zweistellig und besitzt drei Namen.
- Der anonyme Operator `_ _`, mit dem zwei Operanden ohne Operatorsymbol dazwischen aneinandergereiht werden können, ist zweistellig und besitzt keinen Namen.

- Die Bezeichner `start`, `end`, `p-1` und `f'` sind alle nullstellig und besitzen jeweils einen Namen.
- Der mehrteilige Bezeichner `2nd last char` ist ebenfalls nullstellig und besitzt drei Namen.
- Literale wie z. B. `123`, `3.14` oder `'x'` sind formal ebenfalls nullstellige Operatoren.

Wie die Beispiele zeigen, können Operatornamen prinzipiell beliebige Zeichenfolgen sein (z. B. `p-1` und `f'`), und unterschiedliche Operatoren können durchaus gemeinsame Namen besitzen (z. B. `end` als Teil von `while _ do _end` und gleichzeitig als eigener Operator). Damit ist es z. B. auch möglich, den gleichen Operatornamen präfix, infix und postfix mit unterschiedlichen Bedeutungen zu verwenden. Allerdings können ungeschickt gewählte Namen zu Mehrdeutigkeiten bei der Verwendung der Operatoren und damit zu Übersetzungsfehlern führen (z. B. wenn es neben `p-1` einen weiteren nullstelligen Operator `p` gibt, weil dann `p-1` auch die Differenz von `p` und `1` bezeichnen kann, zumindest wenn `p` einen numerischen Typ besitzt).

Ein *Ausdruck* ist die Anwendung eines *n*-stelligen Operators auf *n* Operanden, bei denen es sich wiederum um Ausdrücke handelt. Als Grenzfall ergibt sich für $n = 0$ die Anwendung eines nullstelligen Operators auf null Operanden, d. h. einfach die Verwendung eines nullstelligen Operators, die auch als *atomarer Ausdruck* bezeichnet wird. Für $n > 0$ ergibt sich ein *zusammengesetzter Ausdruck*.

Ein MOSTflexiPL-*Programm* ist nichts anderes als ein Ausdruck.

2.2 Deklarationen

Eine *Deklaration* ist die Anwendung eines vordefinierten *Deklarationsoperators* auf entsprechende Operanden, zum Beispiel:

- Der Deklarationsoperator `_ : _` dient zur Deklaration von Konstanten, die automatisch einen eindeutigen Wert erhalten, wie z. B. `"Person" : type` (`Person` wird als Konstante des vordefinierten Typs `type` deklariert und stellt damit einen neuen Typ dar) oder `"p-1" : Person` (`p-1` wird als Konstante des soeben definierten Typs `Person` deklariert und stellt damit quasi ein Objekt dieses Typs dar).
  Formal sind Konstanten nichts anderes als nullstellige Operatoren, die bei jeder Anwendung den ihnen zugeordneten Wert als Ergebnis liefern.
  Anmerkung: Bei der Deklaration eines Operators muss sein Name (bzw. allgemein seine Signatur, siehe unten) in Anführungszeichen stehen (z. B. `"Person"` in der ersten Deklaration oben), bei seiner Verwendung jedoch nicht (z. B. `Person` in der zweiten Deklaration oben).
- Der Deklarationsoperator `_ : = _` dient zur Deklaration von Konstanten, die mit einem vorgegebenen Wert initialisiert werden, wie z. B. `"N" : = 10` (`N` wird als Konstante mit Wert `10` deklariert und besitzt damit automatisch den vordefinierten Typ `int`), `"M" : = 2 * N` oder `"p-2" : = p-1`.
- Der Deklarationsoperator `[ _ ] _ : _ { _ }` dient zur Deklaration von Operatoren, die eine *Parameterliste* (innerhalb der eckigen Klammern), eine *Signatur* (vor dem Doppelpunkt), einen *Resultattyp* (nach dem Doppelpunkt) und eine *Implementierung* (innerhalb der geschweiften Klammern) besitzen. Solche Operatoren entspre-

chen Prozeduren oder Funktionen in anderen Programmiersprachen, z. B.:

```
["x" : int; "y" : int]                  Parameterliste
"max x y" : int                         Signatur und Resultattyp
{ if x > y then x else y end }          Implementierung
```

Die Parameterliste besteht selbst wieder aus einer Folge von Deklarationen, die durch den zweistelligen sequentiellen Kompositionsoperator `_ ; _` getrennt sind.
Die Signatur (die, wie bereits erwähnt, in Anführungszeichen stehen muss) besteht aus beliebig vielen Wörtern, bei denen es sich entweder um Namen von Parametern (im Beispiel `x` und `y`) oder um Namen des zu deklarierenden Operators handelt (im Beispiel `max`, weil dies kein Name eines Parameters ist). Dadurch wird indirekt die *Syntax* des Operators festgelegt, d. h. wie er anzuwenden ist: Im Beispiel müssen dem Operatornamen `max` zwei Operanden (d. h. Ausdrücke) mit Typ `int` folgen, die den Parametern `x` und `y` entsprechen, z. B. `max 1 2` oder `max 1+2 max 3 4`. Hätte man als Signatur z. B. `"x max y"` angegeben, so müsste der Name `max` zwischen den beiden Operanden stehen.
Die Implementierung ist ein beliebiger Ausdruck, dessen Typ mit dem Resultattyp des Operators (im Beispiel `int`) übereinstimmen muss. Bei einer Anwendung des Operators wird dieser Ausdruck ausgewertet und sein Wert als Ergebnis der Operatoranwendung geliefert, nachdem zuvor die Operanden ausgewertet und die Parameter des Operators mit ihren Werten initialisiert wurden.
Die in der Implementierung verwendeten Operatoren `if _ then _ else _ end` und `_ > _` sind vordefinierte Standardoperatoren.

Da Deklarationen spezielle Ausdrücke sind und Ausdrücke beliebig verschachtelt werden können, können auch Deklarationen beliebig verschachtelt werden, zum Beispiel:

```
["n" : int]
"even n" : bool                          globaler Operator
{
  ["n" : int]                            lokaler Operator,
  "odd n" : bool                         der den globalen aufruft
  { if n == 0 then false else even n-1 end };

  if n == 0 then true else odd n-1 end   Aufruf des lokalen
}                              Operators in der Implementierung des globalen
```

Das Beispiel zeigt nebenbei, dass Operatorimplementierungen beliebig (auch wechselseitig) rekursiv sein können. Außerdem sind Operatoren „Bürger erster Klasse“, d. h. sie können zum Beispiel als Parameter- und Resultatwerte anderer Operatoren auftreten, in Variablen gespeichert werden u. ä. Wenn ein lokaler Operator dadurch die Ausführung eines oder mehrerer umschließender Operatoren „überlebt“, werden deren Ausführungskontexte (die z. B. ihre Parameter und lokalen Variablen enthalten) so lange aufbewahrt, bis sie garantiert für keine lokale Operatorausführung mehr benötigt werden, d. h. lokale Operatoren stellen vollwertige Funktionsabschlüsse (closures) dar. Da jede Operatordeklaration letztlich die Syntax der Sprache erweitert, stellen lokale Deklarationen lokal begrenzte Syntaxerweiterungen dar (vgl. auch §2.5).

2.3 Explizite, implizite und deduzierte Parameter

Die Parameter `x` und `y` des oben definierten Operators `max _ _` heißen *explizite Parameter*, weil die zugehörigen Operanden bei einer Anwendung des Operators explizit angegeben werden.

Um einen *generischen* Maximumoperator definieren zu können, der auf zwei Operanden eines beliebigen Typs `T` angewandt werden kann, braucht man zusätzlich einen *deduzierten* Parameter:

```
["T" : type; "x" : T; "y" : T]
"max x y" : T
{ if x > y then x else y end }
```

Der Parameter `T` heißt deduziert, weil sein Wert bei einer Anwendung des Operators nicht explizit angegeben, sondern aus dem Typ anderer Operanden abgeleitet (deduziert) wird. Dementsprechend erkennt man einen deduzierten Parameter daran, dass er im Typ anderer Parameter (im Beispiel `x` und `y`) auftritt. Außerdem tritt sein Name typischerweise *nicht* in der Signatur des Operators auf. (Wenn er dort auftreten würde, wäre der Parameter sowohl explizit als auch deduziert, was ein relativ seltener Sonderfall ist.)

Typische Anwendungen des soeben definierten Maximumoperators sind `max 1 2` und `max 1.5 3.7`. Im ersten Fall besitzen die Operanden `1` und `2`, die den expliziten Parametern `x` und `y` entsprechen, beide den Typ `int`, sodass `T` gleich `int` deduziert wird. Im zweiten Fall ergibt sich entsprechend `T` gleich `float`, da die Operanden `1.5` und `3.7` beide diesen Typ besitzen. Ein Ausdruck wie z. B. `max 1 3.7` wird vom Compiler als fehlerhaft zurückgewiesen, weil es keine eindeutige Belegung für den deduzierten Parameter `T` gibt: Aus dem Operanden `1` mit Typ `int` ergäbe sich `T` gleich `int`, während sich aus dem Operanden `3.7` mit Typ `float` die Belegung `T` gleich `float` ergäbe. (In einer zukünftigen Version der Sprache soll `int` implizit in `float` umwandelbar sein. Dann wäre `T` gleich `float` in diesem Beispiel eine korrekte Belegung.)

Tatsächlich ist die obige Definition des Maximumoperators auch noch fehlerhaft: Da der Typ `T` der Parameter `x` und `y` beliebig sein kann, findet der Compiler für den Teilausdruck `x > y` keinen passenden Operator `_ > _` und weist daher die gesamte Operatordeklaration als fehlerhaft zurück. Er kennt nur die vordefinierten Vergleichsoperatoren für `int` und `float` sowie eventuelle benutzerdefinierte Operatoren für weitere Typen, aber keinen Größer-Operator für Werte eines beliebigen Typs.

Um dieses Problem zu beheben, könnte man den jeweiligen Vergleichsoperator explizit als zusätzlichen Operanden übergeben, was die Verwendung des Operators jedoch unnötig verkomplizieren würde. Alternativ kann man den Größer-Operator als *impliziten Parameter* übergeben:

```
[ "T" : type; "x" : T; "y" : T;
  ["u" : T; "v" : T] "u > v" : bool]
"max x y" : T
{ if x > y then x else y end }
```

Neben den bereits bekannten Parametern `T`, `x` und `y`, besitzt der jetzt definierte Maximumoperator einen weiteren Parameter `_ > _`, der selbst wiederum explizite Parame-

ter `u` und `v` mit Typ `T` besitzt. Demnach kann dieser Parameter jetzt als Operator im Teilausdruck `x > y` verwendet werden. Dieser Parameter ist weder explizit, weil sein Name nicht in der Signatur des Operators auftritt, noch deduziert, weil er nicht im Typ anderer Parameter auftritt. Diese dritte Art von Parametern heißt *implizit*.

Analog zu einem deduzierten Parameter, wird auch für einen impliziten Parameter bei einer Operatoranwendung kein expliziter Operand übergeben. Stattdessen wird implizit ein an der Anwendungsstelle sichtbarer Operator übergeben, der die gleiche Syntax (d. h. die gleiche Folge von Operatornamen und Operandenpositionen) wie der implizite Parameter und den gleichen (oder einen kompatiblen) Operatortyp besitzt. Daraus folgt, dass die Namen impliziter Parameter − im Gegensatz zu den Namen expliziter und deduzierter Parameter − für die Anwendung eines Operators relevant sind.

Für den Beispielausdruck `max 1 2` bedeutet das konkret: Die expliziten Parameter `x` und `y` werden mit den Werten der Operanden `1` bzw. `2` initialisiert. Da diese beide den Typ `int` besitzen, wird `T` gleich `int` deduziert. Für den impliziten Parameter `_ > _` wird ein Operator mit der gleichen Syntax `_ > _` übergeben, der zwei Operanden des Typs `T` (also `int`) akzeptiert und Resultattyp `bool` besitzt. Hierfür kommt nur der vordefinierte Größer-Operator für `int`-Werte in Frage.

Lautet der Beispielausdruck `max 1.5 3.7`, so ergibt sich `T` gleich `float`, d. h. für den impliziten Parameter `_ > _` wird jetzt ein Operator `_ > _` übergeben, der zwei Operanden des Typs `float` akzeptiert und wiederum Resultattyp `bool` besitzt.

Ein Ausdruck wie z. B. `max p-1 p-2` (mit `p-1` und `p-2` vom Typ `Person`) wird vom Compiler als fehlerhaft zurückgewiesen, weil es (standardmäßig) keinen Operator `_ > _` gibt, der zwei Operanden des Typs `Person` akzeptiert, und der Compiler somit keine Belegung für den impliziten Parameter `_ > _` finden kann.

Das letzte Beispiel zeigt, dass implizite Parameter nicht nur der Bequemlichkeit dienen − weil man ihre Werte nicht explizit übergeben muss −, sondern auch dazu verwendet werden können, Einschränkungen oder Nebenbedingungen für deduzierte Parameter auszudrücken: Der implizite Parameter `_ > _` des Maximumoperators zeigt an, dass dieser Operator nur auf Operanden angewandt werden kann, für deren Typ `T` es einen entsprechenden Vergleichsoperator gibt. Das ist in etwa vergleichbar mit „bounded polymorphism“, wie man ihn z. B. in Java findet, oder mit den Typklassen von Haskell.

2.4 Typen

Ein *Typ* ist ebenfalls nichts anderes als ein Ausdruck (in aller Regel ein *statischer* Ausdruck, siehe unten). Elementare Typen wie z. B. `int` oder `float`, aber auch einfache benutzerdefinierte Typen wie z. B. `Person`, entsprechen atomaren Ausdrücken. Strukturierte Typen, wie z. B. `List Person` (Liste von Personen), `int [10]` (Array von 10 `int`-Werten) oder `int # bool` (Paar von `int` und `bool`), erhält man durch die Anwendung „typwertiger“ Operatoren (d. h. Operatoren mit Resultattyp `type`, anderweitig auch als Typkonstruktoren bezeichnet) auf andere Typen und/oder Werte, zum Beispiel:[2]

[2] Da Typen somit von Werten abhängen können, handelt es sich um eine Form von „dependent types“ [Wi12].

```
["T" : type] "List T" : type;
["T" : type; "N" : int] "T [ N ]" : type;
["X" : type; "Y" : type] "X # Y" : type
```

Die hier definierten Operatoren sind *statisch*, weil sie keine benutzerdefinierte Implementierung (geschweifte Klammern) besitzen, die ihr Verhalten zur Laufzeit bestimmt, sondern ein vordefiniertes Verhalten mit folgenden Eigenschaften:

- Die mehrmalige Anwendung eines statischen Operators auf die gleichen Operandenwerte liefert immer den gleichen Resultatwert. Daraus folgt z. B., dass jede Verwendung von `List Person` denselben Typ liefert.
- Die Anwendung eines statischen Operators auf unterschiedliche Operandenwerte liefert unterschiedliche Resultatwerte. Daraus folgt z. B., dass `List Person` und `List int` garantiert unterschiedliche Typen sind.
- Anwendungen unterschiedlicher statischer Operatoren liefern unterschiedliche Resultatwerte. Daraus folgt z. B., dass `List Person` und `int # bool` garantiert unterschiedliche Typen sind (da `List _` und `_ # _` unterschiedliche statische Operatoren sind), ebenso aber auch `List Person` und `Person` (da `List _` und `Person` unterschiedliche statische Operatoren sind; tatsächlich sind die in §2.2 eingeführten Konstanten nichts anderes als parameterlose statische Operatoren).

Aus diesen Eigenschaften folgt, dass zwei statische Ausdrücke (d. h. Ausdrücke, in denen nur statische Operatoren auftreten) genau dann den gleichen Wert besitzen, wenn sie *strukturgleich* sind (d. h. wenn es sich um Anwendungen desselben Operators auf paarweise strukturgleiche Operanden handelt).

Wenn Typen statische Ausdrücke sind, bedeutet das, dass der Compiler ihre Gleichheit einfach durch einen Vergleich ihrer Struktur überprüfen kann. Wenn ein Typausdruck nicht statisch ist, d. h. Operatoren mit prinzipiell nicht vorhersagbarem dynamischem Verhalten enthält, gilt diese Äquivalenz von Wert- und Strukturgleichheit nicht mehr. Da der Compiler aber nur die Strukturgleichheit von Typausdrücken zur Übersetzungszeit und nicht ihre eventuelle Wertgleichheit zur Laufzeit überprüfen kann, betrachtet er Typen trotzdem nur dann als gleich, wenn sie strukturgleich sind. Daraus folgt zum Beispiel, dass `int [1+2]`, `int [2+1]` und `int [3]` aus Sicht des Compilers drei unterschiedliche Typen sind, weil die Ausdrücke nicht strukturgleich sind, obwohl ihre (i. d. R. uninteressanten) Werte zur Laufzeit natürlich gleich sein werden.[3]

Neben elementaren Typen wie `int` und `float`, gibt es zwei vordefinierte Typoperatoren `_ *` und `_ ?` zur Definition von *Sequenz-* bzw. *Variablentypen*. Für einen beliebigen Typ `T` bezeichnet der Typ `T*` Sequenzen von `T`-Werten beliebiger Länge, ähnlich wie `vector<T>` o. ä. in anderen Sprachen, während `T?` Variablen von `T`-Werten bezeichnet. Für eine Sequenz `s` vom Typ `T*` liefert `#s` ihre Länge, d. h. die Anzahl ihrer Elemente, während `s[i]` für `i` von 1 bis `#s` (jeweils einschließlich) ihr `i`-tes Element

[3] In einer zukünftigen Version der Sprache lassen sich vielleicht Regeln formulieren (etwa das Kommutativgesetz der Addition), die dem Compiler erlauben, durch entsprechende Strukturtransformationen unterschiedliche Ausdrücke als „logisch gleich“ zu erkennen und zu akzeptieren.

(vom Typ `T`) liefert. Für eine Variable `v` vom Typ `T?` liefert `?v` ihren aktuellen Wert (vom Typ `T`)[4], während `v << x` oder `x >> v` den Wert von `x` als neuen Wert zuweist.

Statische Operatoren, deren Resultattyp ein Variablentyp ist, können z. B. wie folgt zur (inkrementellen) Definition von „Attributen“ (vgl. [He07]) verwendet werden:

```
["p" : Person] "p name" : string?
```

Für jede Person `p` liefert der Ausdruck `p name` eine eindeutige `string`-Variable, deren Wert mittels `? p name` abgefragt und mittels `p name << ...` verändert werden kann. In gleicher Weise könnte man z. B. auch Attribute `_ head` und `_ tail` für verkettete Listen definieren und damit dem bis jetzt noch „nackten“ Typ `List T` eine konkrete Bedeutung geben.

2.5 Import-, Export- und Ausschlussdeklarationen

Mit *Import-* und *Exportdeklarationen* lässt sich detailliert spezifizieren, welche Operatoren in welchen Teilausdrücken eines Programms sichtbar und damit verwendbar sein sollen. Beispielsweise gibt es für den vordefinierten sequentiellen Kompositionsoperator `_ ; _` eine Exportdeklaration, die besagt, dass die Deklarationen beider Operanden exportiert werden, sodass sie über die jeweilige Anwendung des Operators hinaus sichtbar sind. Außerdem gibt es eine Importdeklaration, die besagt, dass die Deklarationen des ersten Operanden im zweiten Operanden sichtbar sind. Für einen benutzerdefinierten Operator `let _ in _ end` könnte man ebenfalls vereinbaren, dass die Deklarationen des ersten Operanden im zweiten Operanden sichtbar sind, aber nur die Deklarationen des zweiten Operanden exportiert werden, sodass die Deklarationen des ersten Operanden „privat“ bleiben und die durch sie definierten Syntaxerweiterungen nur lokal gültig sind.

Mit *Ausschlussdeklarationen* können Vorrang und Assoziativität von Operatoren sehr flexibel geregelt werden (insbesondere wesentlich flexibler als mit einer einfachen Tabelle ganzzahliger Operatorprioritäten). Beispielsweise wird die bekannte Punkt-vor-Strich-Regel dadurch ausgedrückt, dass additive Operatoren (`_ + _` und `_ - _`) als Hauptoperator (d. h. als „oberster“ Operator im zugehörigen Operatorbaum) der Operanden eines multiplikativen Operators (`_ * _` und `_ / _`) verboten sind. Damit ist von den zwei prinzipiell denkbaren Zerlegungen einer Zeichenfolge wie `2 + 3 * 4` nur diejenige korrekt, bei der sich die Multiplikation „unterhalb“ der Addition befindet. Der explizit geklammerte Ausdruck `(2 + 3) * 4` ist nichtsdestotrotz korrekt, da der Klammeroperator `( _ )` in MOSTflexiPL ein selbständiger Operator ist und der Plusoperator damit nicht mehr der Hauptoperator des linken Operanden der Multiplikation ist.

Die Links- bzw. Rechtsassoziativität eines binären Operators (oder einer Gruppe solcher Operatoren) kann ebenfalls durch eine Ausschlussdeklaration spezifiziert werden, die den Operator selbst als Hauptoperator seines rechten bzw. linken Operanden verbietet.

---

[4] Mit Hilfe impliziter Typumwandlungen kann man in einer zukünftigen Version der Sprache vereinbaren, dass eine Variable mit Typ `T?` bei Bedarf automatisch durch eine Anwendung des Operators `? _` in ihren Wert des Typs `T` umgewandelt wird. Das entspricht dann der aus anderen Sprachen bekannten automatischen Umwandlung von Variablen (L-Werte in C/C++) in ihre Werte (R-Werte).

3 Beispiele für Spracherweiterungen

3.1 Iteration über Sequenzen

Mit den in §2.4 erwähnten Operatoren für Sequenzen sowie dem ebenfalls vordefinierten Wiederholungsoperator `while _ do _ end` kann man wie folgt über eine Sequenz `s` mit Typ `T*` iterieren, um beispielsweise ihre Elemente auszugeben:

```
"i" : int?;                    int-Variable i
i << 1;                        Zuweisung i gleich 1
while ?i <= #s do              Solange Wert von i ≤ Länge von s:
  ... s[?i] ...                  Verwendung des Elements s[?i]
  i << ?i + 1                    Zuweisung i gleich Wert von i plus 1
end
```

Kompakter und bequemer wäre jedoch folgende Formulierung:

```
"v" : T?;                      T-Variable v
for v in s do                  Für jedes Element v von s:
  ... ?v ...                     Verwendung von ?v
end
```

Hierfür wird ein neuer Operator `for _ in _ do _ end` benötigt, der folgende explizite Parameter besitzt:

- eine Variable `v` des Typs `T?`, wobei `T` ein beliebiger Typ sein kann;
- eine Sequenz `s` des Typs `T*`;
- einen Schleifenrumpf `body` mit einem beliebigen Typ `B`.

Folglich benötigt der Operator zusätzlich deduzierte Parameter `T` und `B`, jeweils mit Typ `type`. Seine Deklaration kann daher wie folgt lauten:

```
["T" : type; "B" : type; "v" : T?; "s" : T*; "body" : B {}]
"for v in s do body end" : int
{ "i" : int?; i << 1;
  while ?i <= #s do
    v << s[?i]; body; i << ?i + 1
  end
}
```

Der Operator besitzt Resultattyp `int`, weil jeder Operator einen Resultattyp besitzen muss und Schleifen per Konvention die Anzahl der ausgeführten Iterationen als Resultatwert liefern. Demnach kann der Resultatwert der `while`-Schleife direkt als Resultatwert des `for`-Operators verwendet werden.

Die leeren geschweiften Klammern am Ende der Deklaration des Parameters `body` zeigen an, dass der entsprechende Operand nicht, wie sonst üblich, als Wert („call by value“), sondern als *unausgewerteter Ausdruck* übergeben wird, ähnlich zu „lazy evaluation“ in funktionalen Sprachen und „call by name“ in Algol. Seine Auswertung, deren Effekt ja typischerweise vom aktuellen Wert der Variablen `v` abhängt, erfolgt dann jedesmal, wenn der Parameter `body` in der Implementierung des `for`-Operators verwendet wird, d. h. konkret einmal pro Iteration.

3.2 Bildung von Teilsequenzen

Mit den in §2.4 erwähnten Operatoren und einem weiteren vordefinierten Operator `_ + _` zur Verkettung zweier Sequenzen oder einer Sequenz und eines einzelnen Elements könnte man z. B. wie folgt einen Operator zur Bildung von Teilsequenzen definieren:

```
["T" : type; "s" : T*; "i" : int; "j" : int]
"subseq s i j" : T*
{ "t" : T*?; "k" : int?; k << max i 1;
  while ?k <= min j #s do
    t << ?t + s[?k]; k << ?k + 1
  end;
  ?t
}
```

Allerdings ist die Bedeutung einer Anwendung wie z. B. `subseq s 3 7` ohne Konsultation der Dokumentation nicht ohne weiteres klar. Wird eine Teilsequenz vom dritten bis zum siebten Element von `s` geliefert − und wenn ja, sind die Randelemente jeweils im Ergebnis enthalten oder nicht −, oder erhält man eine Teilsequenz mit sieben Elementen, die beim dritten Element von `s` beginnt? In unterschiedlichen Sprachen bzw. den zugehörigen Bibliotheken findet man in der Tat zahlreiche Variationen zu diesem Thema.

Wenn man sich gedanklich von der gewohnten Funktions- bzw. Methodensyntax anderer Sprachen löst und berücksichtigt, dass Operatoren in MOSTflexiPL eine beliebige Syntax besitzen können, kann man mit etwas Kreativität z. B. folgende Anwendungsmöglichkeiten konzipieren:

```
s[3 .. 7]      s[3 .. 7)      s(3 .. 7]      s(3 .. 7)
```

Hier dürfte aufgrund bekannter mathematischer Konventionen auch ohne viel Erklärung klar sein, in welchem Fall welche Randelemente zur gelieferten Teilsequenz gehören und welche nicht. Auch weitere Variationen wie z. B. `s[3 ..]` oder `s(.. 7]` dürften nahezu selbsterklärend sein. So liefert die Kombination `s[3 ..][.. 7]` z. B. eine Teilsequenz mit sieben Elementen, die beim dritten Element von `s` beginnt.

Der Operator `_ [ _ .. _ ]` besitzt dieselbe Implementierung wie der obige Operator `subseq _ _ _`, er bietet lediglich eine andere syntaktische „Verpackung“ an:

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

Auch die übrigen Operatoren können auf diesen zurückgeführt werden, z. B.:

```
["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 }
```

4 Implementierung

Abbildung 1 zeigt die funktionale Grobstruktur des Parsers, der zusammen mit der statischen Typprüfung das Herzstück des MOSTflexiPL-Compiler-Frontends darstellt.

Die meisten `parse`-Funktionen erhalten als Parameter die aktuelle Position im Eingabestrom sowie die Menge der an dieser Stelle sichtbaren Operatordeklarationen. Zu Beginn ist dies die Menge der vordefinierten Operatoren, zu der durch Aufrufe von `create_oper` sukzessive benutzerdefinierte Operatoren hinzukommen können. Durch operatorspezifische Import- und Exportdeklarationen (vgl. §2.5) kann die Menge der sichtbaren Deklarationen aber auch gezielt lokal eingeschränkt werden.

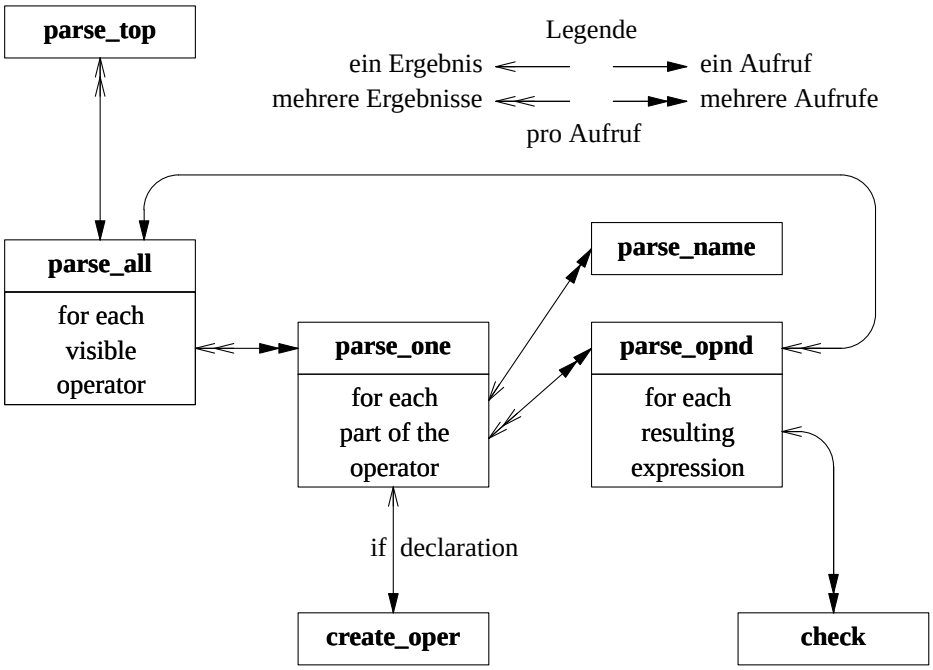

Abbildung 1: Funktionale Grobstruktur des Parsers

Da eine Eingabefolge prinzipiell immer mehrere Interpretationsmöglichkeiten besitzen kann, von denen manche später aufgrund von Ausschlussdeklarationen (vgl. §2.5) oder Typprüfungen eventuell wieder ausgesondert werden können, liefern die meisten Funktionen als Resultat eine Menge von Ausdrücken (d. h. Operatoranwendungen), die mögliche Zerlegungen der nachfolgenden Eingabe darstellen.

Die parameterlose Funktion `parse_top`, die vom Hauptprogramm des Compilers aufgerufen wird, ruft die Funktion `parse_all` mit Eingabeposition 1 und der Menge der vordefinierten Operatordeklarationen auf. `parse_all` hat die Aufgabe, *alle* möglichen Zerlegungen der nachfolgenden Eingabe zu ermitteln und zurückzuliefern. Sie ruft hierfür für jeden sichtbaren Operator die Funktion `parse_one` auf, die alle möglichen Anwendungen dieses *einen* Operators ermittelt und zurückliefert. `parse_one`

arbeitet hierfür die Syntax des Operators Schritt für Schritt ab, indem sie für jeden Teil der Syntax eine der Hilfsfunktionen `parse_name` (wenn es sich um einen Operatornamen handelt) oder `parse_opnd` (wenn es sich um einen Platzhalter für einen Operanden handelt), jeweils mit der entsprechenden Eingabeposition, aufruft. `parse_name` überprüft lediglich, ob der jeweilige Operatorname an der jeweiligen Position in der Eingabe steht, wenn zuvor ggf. Zwischenraum und Kommentar überlesen wird. (Das heißt, `parse_name` ist eigentlich ein degenerierter Scanner.) Wenn dies nicht der Fall ist, kann die Verarbeitung des Operators in `parse_one` sofort abgebrochen werden.

`parse_opnd` hat die Aufgabe, zu dem bis jetzt von `parse_one` konstruierten partiellen Ausdruck einen weiteren Operanden hinzuzufügen. Um die Menge der prinzipiell möglichen Operanden an der entsprechenden Eingabeposition zu ermitteln, wird zunächst rekursiv `parse_all` aufgerufen. Für jeden von dieser Funktion zurückgelieferten Ausdruck wird dann mit Hilfe der Funktion `check` überprüft, ob sein Typ auf den Typ des entsprechenden expliziten Parameters passt und ob keine Ausschlussdeklaration verletzt wird. Wenn dies der Fall ist, wird der Ausdruck als Operand zu einer Kopie des partiellen Ausdrucks hinzugefügt. Als Resultat liefert `parse_opnd` dann die Menge all dieser erweiterten Ausdrücke.

Auf diese Weise werden durch rekursives Backtracking alle möglichen Zerlegungen der Eingabe in typkorrekte Ausdrücke ermittelt. Wenn es genau eine solche Zerlegung gibt, wird die Eingabe als korrektes Programm akzeptiert und vom Compiler-Backend für diesen Ausdruck Zielcode generiert. Wenn es keine typkorrekte Zerlegung gibt, ist das Programm fehlerhaft. Gibt es mehr als eine korrekte Zerlegung, ist das Programm mehrdeutig (und damit ebenfalls fehlerhaft), was insbesondere dann auftreten kann, wenn der Vorrang zwischen Operatoren nicht ausreichend festgelegt wurde.

Um die Performance des Backtracking-Algorithmus zu verbessern, werden fehlerhafte Zerlegungen durch die in den Parse-Vorgang integrierte Typprüfung so früh wie möglich erkannt und ausgesondert. Außerdem werden wiederholte Aufrufe von `parse_all` mit denselben Aufrufparametern (Eingabeposition und sichtbare Deklarationen), die aufgrund der Backtracking-Strategie sehr häufig auftreten, durch dynamisches Programmieren optimiert, indem die einmal ermittelten Zerlegungen in einer Tabelle gespeichert werden. Damit ist die Laufzeit des Compilers normalerweise etwa proportional zur Länge der Eingabe.

5 Verwandte Arbeiten

Eine der ältesten Programmiersprachen, die von Anfang an syntaktisch erweiterbar konzipiert wurde, dürfte Lisp sein [St90]. Ähnlich wie in MOSTflexiPL, besteht in Lisp weder ein Unterschied zwischen Operatoren (wie z. B. `+`) und Funktionen (wie z. B. `max`) noch ein Unterschied zwischen vordefinierten und benutzerdefinierten Operatoren bzw. Funktionen. Durch die Definition neuer Funktionen − oder Makros, die syntaktisch genauso verwendet werden wie Funktionen − erweitert man in gewisser Weise ständig die Syntax der Sprache.

Dies hatte zur Folge, dass zahlreiche problemorientierte Sprachen (domain-specific languages), beispielsweise im Bereich Wissensrepräsentation [Ma91], als Spracherweiterungen von Lisp konzipiert und implementiert wurden. Aber auch das Common

Lisp Object System (CLOS), das heute ein fester Bestandteil von Common Lisp ist, ist nichts anderes als eine Erweiterung der ursprünglichen funktionalen Sprache Lisp.

Ebenfalls analog zu MOSTflexiPL ist die Tatsache, dass Spracherweiterungen in der Sprache selbst formuliert werden und dass hierfür ein äußerst kleiner Sprachkern ausreichend ist. Wesentliche Unterschiede bestehen jedoch darin, dass Lisp kein statisches Typsystem besitzt und dass Ausdrücke immer geklammert werden müssen, wodurch der syntaktische Gestaltungsspielraum deutlich eingeschränkt ist. Außerdem kommt MOSTflexiPL bis jetzt ohne ein „prozedurales“ Makrosystem aus, d. h. es wird keinerlei Benutzercode zur Übersetzungszeit ausgeführt.[5]

Zusammengefasst bietet MOSTflexiPL im wesentlichen die gleiche Flexibilität wie Lisp, aber mit vollkommen frei gestaltbarer Syntax sowie statischer Typprüfung.

Eine modernere Sprache, die viele Ideen von Lisp übernommen hat, ist Dylan [Cr97]. Auch sie erlaubt dem Programmierer, die Syntax der Sprache in der Sprache selbst (eigentlich in einem Makrosystem, das Teil der Sprache ist) zu erweitern. Obwohl die Möglichkeiten weiter reichen als mit den einfachen Klammerausdrücken von Lisp, sind dem Anwender dennoch klare syntaktische Grenzen gesetzt, die er nicht überschreiten kann. Demgegenüber erlaubt das Operatorkonzept von MOSTflexiPL nahezu beliebige syntaktische Gestaltungsmöglichkeiten. Außerdem bietet auch Dylan kein statisches Typsystem.

Zahlreiche unterschiedliche Sprachen, wie z. B. Haskell [Ma10] und Prolog [CM94], bieten die Möglichkeit, neue Operatorsymbole zu definieren und damit die Syntax von Ausdrücken zu erweitern. Da in funktionalen Sprachen, ähnlich wie in MOSTflexiPL, kein Unterschied zwischen Ausdrücken und Anweisungen besteht, ist damit prinzipiell auch die Syntax von Anweisungen (z. B. Kontrollstrukturen) erweiterbar. Nicht erweiterbar ist jedoch die Syntax von Typen und Deklarationen.

Durch das einfache Prinzip „*Alles* ist ein Ausdruck“, unter anderem auch Typen und Deklarationen, lassen sich diese Teile der Sprache in MOSTflexiPL durch die Definition neuer Operatoren genauso erweitern wie beispielsweise Ausdrücke und Anweisungen.

Neben reinen Programmiersprachen, die innerhalb bestimmter Grenzen „in sich selbst“ erweiterbar sind, gibt es diverse „Frameworks“, bei denen syntaktische Erweiterungen − wiederum innerhalb bestimmter Grenzen − in einer separaten Metasprache definiert werden können. Beispiele hierfür sind XL bzw. XLR [Di12], Seed7 [Me12] und JSE [BP01].

Die Nachteile dieser Ansätze gegenüber MOSTflexiPL sind einerseits, dass Spracherweiterungen nicht direkt und einfach in der Sprache selbst formuliert werden können, und andererseits, dass wiederum bestimmte „syntaktische Grenzen“ nicht überschritten werden können.

Ein Ansatz, dessen Grundidee der von MOSTflexiPL am ähnlichsten ist, ist „π − a Pattern Language“ [KM09]. Das dort als Pattern bezeichnete Grundkonstrukt der Sprache − nach Aussage der Autoren das einzige, das es gibt − entspricht einem Operator in MOSTflexiPL: Es besitzt eine Syntax, die aus Namen bzw. Symbolen sowie Platzhaltern für Operanden besteht, und eine Bedeutung, die der Implementierung ei-

[5] Es gibt lediglich einen deklarativen (und typsicheren) Ersetzungsmechanismus (sog. „virtuelle Operatoren“), der u. a. zur Abkürzung von Typausdrücken verwendet werden kann, z. B. `IntSeq = int*`. Die Ersetzung von `IntSeq` durch `int*` ist wichtig, damit der Compiler beide als gleichen Typ erkennt (vgl. §2.4).

nes Operators in MOSTflexiPL entspricht. Damit bieten beide Ansätze dieselbe syntaktische Flexibilität, die letztlich daher rührt, dass beide Sprachen keinerlei vordefinierte Grammatik besitzen.

Ein entscheidender Pluspunkt von MOSTflexiPL gegenüber π ist wiederum das statische Typsystem − π ist vollständig dynamisch typisiert. Darüber hinaus bietet MOSTflexiPL eine Reihe weiterer Möglichkeiten, wie z. B. implizite und deduzierte Parameter (wobei letztere in einer dynamisch typisierten Sprache nicht benötigt werden) sowie Import-, Export- und Ausschlussdeklarationen, mit denen sich z. B. lokal begrenzte Syntaxerweiterungen realisieren lassen.

6 Zusammenfassung und Ausblick

MOSTflexiPL ist eine momentan in Entwicklung befindliche Programmiersprache, die vom Anwender nahezu beliebig syntaktisch erweitert und angepasst werden kann und damit u. a. als erweiterbare „general purpose programming language“ sowie zur Einbettung von „domain-specific languages“ verwendet werden kann. Sie besitzt ein statisches Typsystem und wird durch einen Compiler nach C++ übersetzt.

Seit der Verfügbarkeit eines ersten, stellenweise noch unvollständigen Compilers[6] konnten erste Erfahrungen mit der Verwendung der Sprache gesammelt werden. Dabei traten, wie nicht anders zu erwarten, noch einige Schwachstellen zutage, die im Zuge der Weiterentwicklung beseitigt werden sollen.

Das größte konzeptuelle Defizit ist momentan die Tatsache, dass benutzerdefinierte Deklarationsoperatoren (d. h. vom Anwender geschriebene Operatoren, die zur Deklaration anderer Operatoren verwendet werden sollen) nur sehr eingeschränkt funktionieren. Um diesen Mangel zu beseitigen, muss das Konzept von Deklarationen noch einmal grundlegend überarbeitet werden, insbesondere die Interpretation der Signatur eines Operators, die momentan eine Stringkonstante sein muss und daher z. B. nicht von den Parametern eines benutzerdefinierten Deklarationsoperators abhängen kann.

Weitere konzeptuelle Verbesserungen betreffen die Unterstützung für benutzerdefinierte Literale, Zwischenraum und Kommentare; momentan sind diese Sprachmerkmale fest vordefiniert. Zur Definition komplexerer Kontrollstrukturen wird noch ein allgemeiner Sprungmechanismus benötigt, mit dem sich dann spezielle Anweisungen wie `break`, `return` oder `throw` realisieren lassen.

Weitere Punkte auf dem konzeptuellen Wunschzettel sind implizite Typumwandlungen − die natürlich auch wieder beliebig benutzerdefinierbar sein sollen − sowie ein geeignetes Modulkonzept. Für beide Punkte sind die wesentlichen Ideen vorhanden und ausgearbeitet − daher findet sich das Stichwort „modular“ bereits im Namen der Sprache −, sie müssen aber noch im Compiler implementiert werden.

Schließlich gibt es momentan nur eine rudimentäre Fehlerdiagnose und -behandlung. Diese zu verbessern, dürfte aufgrund der Flexibilität der Sprache noch ein umfangreiches und schwieriges Forschungsunterfangen sein.

Neben diesen konzeptuellen Verbesserungen, muss auch der MOSTflexiPL-Compiler an diversen Stellen weiterentwickelt und verbessert werden. Beispielsweise gerät die

[6] Dieser wurde zum größten Teil von den Studierenden Marco Perazzo, Miriam Klement und Stefan Billet im Rahmen ihrer exzellenten Abschlussarbeiten implementiert.

Suche nach impliziten Parameterbelegungen bei „bösartigen“ Beispielprogrammen momentan noch in eine Endlosschleife, während die Suche nach deduzierten Parameterbelegungen bei komplexen Beispielen noch unvollständig ist. Außerdem kann sowohl die Performance des Compilers als auch die des generierten Codes noch deutlich verbessert werden. Schließlich sollen in Zukunft weitere Zielsprachen unterstützt werden, beispielsweise Java-Quell- oder Bytecode, wodurch MOSTflexiPL-Programme auf jeder Java Virtual Machine ausgeführt werden können.

Literaturverzeichnis

[BP01] 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), 31−42.

[CM94] W. F. Clocksin, C. S. Mellish: *Programming in Prolog* (Fourth Edition). Springer-Verlag, Berlin, 1994.

[Cr97] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[Di12] C. de Dinechin: *XL Extensible Language*. http://xlr.sourceforge.net (18.01.2012)

[He07] 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.

[KM09] R. Knöll, M. Mezini: “π − a Pattern Language.” In: *Proc. 24th Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)* (Orlando, FL, October 2009), 503−521.

[Ma10] S. Marlow (ed.): *Haskell 2010 Language Report*. HaskellWiki, 2010. http://haskell.org/definition/haskell2010.pdf (18.01.2012)

[Ma91] R. MacGregor: “The Evolving Technology of Classification-Based Knowledge Representation Systems.” In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 385−400.

[Me12] T. Mertes: *Seed7 − The Extensible Programming Language*. http://seed7.sourceforge.net (18.01.2012)

[St90] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

[Wi12] Wikipedia contributors: *Dependent Type*. http://en.wikipedia.org/w/index.php?title=Dependent_type&oldid=470369653 (18.01.2012)