Entwurf und Implementierung einer Programmiersprache im studentischen Projekt *Monty's Coconut*

Marcus Ermler Berthold Hoffmann Christian John Christopher Nottrodt Carsten Pfeffer

Fachbereich 3 – Mathematik und Informatik, Universität Bremen
Postfach 330 440, D-28344 Bremen
{maermler,hof,cjohn,knox,pfeffer}@informatik.uni-bremen.de

Zusammenfassung. Dieser Erfahrungsbericht beschreibt zum einen ein studentisches Projekt an der Universität Bremen, in dem eine Programmiersprache namens *Monty* entworfen und implementiert wurde. Des Weiteren wird auch die Sprache selbst anhand einiger Beispiele vorgestellt. *Monty* soll die Vorteile von klassischen Programmiersprachen wie *C++* und *Java* mit denen von Skriptsprachen wie *Ruby* und *Python* kombinieren.

1 Das studentische Projekt *Monty's Coconut*

Das Informatikstudium an der Universität Bremen ist schon seit seinen Anfängen in den 1980er Jahren projektorientiert. In den viersemestrigen Projekten des Diplomstudiengangs haben die Studierenden den kompletten Softwareentwicklungskreislauf von Spezifikation der Anforderungen, Entwurf der Architektur über Implementierung bis hin zu Test und Dokumentation durchlaufen. Diese Projekte sind im jetzigen Bachelor- und Masterstudium aufgeteilt worden in zweisemestrige Bachelorprojekte (mit 18 *Credit Points*) und zweisemestrige Masterprojekte (mit 24 *Credit Points*), wobei Masterprojekte oft als Fortsetzungen von Bachelorprojekten konzipiert werden.

Beim Masterprojekt *Monty's Coconut* ist besonders, dass es von Studierenden selbst initiiert wurde: Zwei Studenten gewannen Marcus Ermler und Berthold Hoffmann als Betreuer für ihre Projektidee und erarbeiteten gemeinsam einen Projektvorschlag, für den sie dann zwölf weitere Studierende begeistern konnten. Sonst entstehen die meisten Projektvorschläge aus Forschungsthemen der Arbeitsgruppen, unter denen die Studierenden dann auswählen.

Im Projekt *Monty's Coconut* sollte mit dem Entwurf der Sprache *Monty* ein Versuch unternommen werden, die Lücke zwischen klassischen Programmiersprachen und Skriptsprachen zu schließen. Einen Referenzpunkt stellte die von David Watt (Universität Glasgow) in [Wat05] vorgeschlagene Sprache dar, die ebenfalls *Monty* heißt, wobei in dieser Studie jedoch das Zusammenführen von dynamischer und statischer Typisierung im Mittelpunkt stand.

Vor dem eigentlichen Projektbeginn haben sich die Studierenden inhaltlich im Kurs *Übersetzerbau*, im Seminar *Skriptsprachen* und im Praktikum *Übersetzerwerkzeuge* auf das Projekt vorbereitet und projektbegleitend in Kursen der Studierwerkstatt Methoden zum Projektmanagement und zur Konfliktbehandlung angeeignet. Von Oktober 2013 bis September 2014 wurden in wöchentlichen vierstündigen Plenumstreffen jeweils die Ergebnisse der vergangenen Woche diskutiert und Aufgaben für die Arbeit von Untergruppen in der nächsten Woche verteilt. Die Arbeit im Projekt lässt sich grob in zwei Phasen untergliedern: (1) Diskussion und Entwicklung von Sprachkonzepten und (2) Erstellung der Sprachdefinition mit paralleler Entwicklung des Compilers. Am Ende der ersten Phase wurde die bis dahin entwickelte Konzeption der Sprache auf einem selbst organisierten Workshop mit David Watt diskutiert. In Phase (2) wurde auf dem Projekttag des Fachbereichs Mathematik/Informatik der Zwischenstand des Projekts den Studierenden, Mitarbeiterinnen und Mitarbeitern präsentiert. Zum Abschluss wurden die Ergebnisse dann in einem öffentlichen Vortrag auch Interessierten aus anderen Fachbereichen vorgestellt.

Submission to: 8. Arbeitstagung Programmiersprachen, Dresden, Germany, 18-Mar-2015, to appear at http://ceur-ws.org

Dieses Papier ist einerseits ein Erfahrungsbericht, der den Projektverlauf und die größten Herausforderungen im Sprachentwurf zusammenfassen soll. Darüber hinaus wird auch die Sprache *Monty* selbst vorgestellt und anhand von Beispielen erläutert.

Die restlichen Abschnitte sind daher folgendermaßen gegliedert: In Abschnitt 2 wird auf den Entwurf von *Monty* eingegangen, d.h. die Zielsetzung beim Sprachentwurf und die darin umgesetzten Konzepte. Abschnitt 3 stellt den implementierten Compiler vor. In Abschnitt 4 ziehen wir ein Fazit des Projekts und wagen uns an eine vorläufige Bewertung von *Monty*.

2 Der Entwurf der Programmiersprache *Monty*

Klassische Programmiersprachen wie *C++* [Str13] und *Java* [GJS+14] haben ein statisches Typsystem und werden mit Compilern implementiert. Ihre Programme sind vielleicht etwas aufwändig zu schreiben, können dafür aber auf Typfehler überprüft und effizient ausgeführt werden. Dagegen werden Skriptsprachen wie *Ruby* [FM08] und *Python* [vRD11] meist interpretiert. Mit ihrer knappen Syntax und ihrer dynamischen Typisierung sind sie auf schnelles und bequemes Schreiben ausgerichtet, wobei Geschwindigkeitseinbußen bei der Ausführung in Kauf genommen werden. Die Sprache *Monty* wurde mit dem Ziel entwickelt, die Lücke zwischen Programmiersprachen und Skriptsprachen zu überbrücken (siehe Abbildung 1). Eine einfach zu erlernende Syntax und eine vorwiegend statische, aber flexible Typisierung soll weiterhin vergleichsweise hohen Entwicklungskomfort bieten, ohne Abstriche bei der Effizienz machen zu müssen, denn die Sprache wird kompiliert. Die Objektorientierung ist klassenbasiert, wobei – im Gegensatz zu *Java* und *C++* – in *Monty* alle Werte Objekte sind, auch Wahrheitswerte und Zahlen.

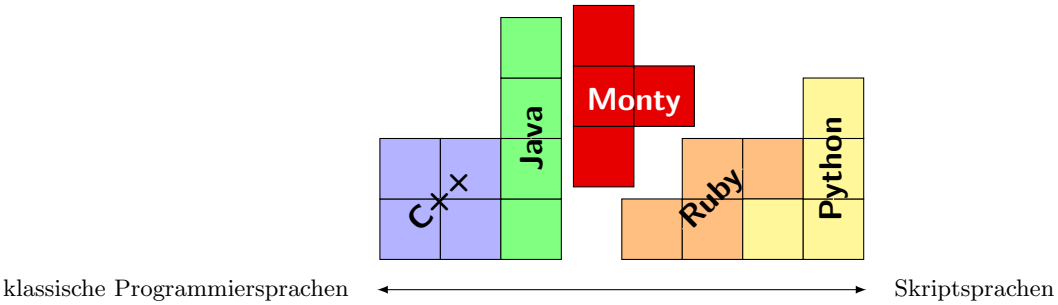

Abbildung 1: Die Sprache *Monty* soll die Lücke zwischen Programmier- und Skriptsprachen überbrücken

Schon im Semester vor dem eigentlichen Projektbeginn wurden im Seminar *Skriptsprachen* viele ältere und neuere Sprachen in Hinblick auf die von ihnen unterstützten Konzepte und die Qualität ihres Entwurfs analysiert. In der ersten Phase des Projekts wurde dann intensiv diskutiert, welche Konzepte in *Monty* unterstützt werden sollen. Bei der Vererbung standen beispielsweise die Einfachvererbung mit *Interfaces* wie in Java, mit *traits* wie in *Scala* [OSV10] und allgemeine Mehrfachvererbung wie in *C++* zur Debatte. So wurde die Grundlage für einen detaillierten Anforderungskatalog an Konzepten und Eigenschaften der Sprache gelegt. Im Rahmen eines zweitägigen Workshops wurden diese Überlegungen einem externen Berater vorgestellt und mit ihm diskutiert: David Watt von der Universität Glasgow hatte zuvor ebenfalls eine Sprache mit dem Namen *Monty* konzipiert [Wat05], die ähnliche Ziele verfolgte und eine wichtige Inspiration für die hier vorgestellte Sprache war. (Auf die Unterschiede zwischen David Watts *Monty* und unserer Sprache gehen wir am Ende dieses Abschnitts ein.)

Syntaktisch erinnert *Monty* stark an *Python*, was vor allem dadurch deutlich wird, dass Blöcke nicht durch geschweifte Klammern oder Schlüsselwörter begrenzt, sondern eingerückt werden. Beispiel 1 zeigt ein einfaches *Monty*-Programm. Anweisungen können wie in einer Skriptsprache als Sequenz aneinandergereiht werden, ohne dass zusätzliche Konstrukte wie *main*-Funktionen oder Klassen eingeführt werden müssen.

```
Array<Int> numbers := [45, -13, 18, -5, -9, 0, 16]

Bool positive(Int x):
    return x > 0

print(numbers.filter(positive)) // [45, 18, 16]
```

Beispiel 1: Ein einfaches *Monty*-Programm (Filtern von Sequenzen)

Das Beispiel definiert ein Array von *Int*-Objekten und eine Funktion, die auf die Elemente des Arrays angewendet werden kann. Die Klasse *Array* ist generisch: sie kann für beliebige andere Elementtypen instantiiert werden. Darüber hinaus ist sie eine Implementierung der abstrakten Klasse *Sequence*, von der auch Typen wie *List* und *String* erben. Die Zuweisung wird in *Monty* im Gegensatz zu den anderen oben genannten Sprachen durch ein "`:=`" ausgedrückt, da diese Schreibweise mathematisch plausibler ist. Somit kann das einfache Gleichheitszeichen "`=`" für das Gleichheitsprädikat benutzt werden.

In der letzten Zeile des Beispiels wird die Methode *filter* aufgerufen, die ebenfalls auf *Sequence* definiert ist. Sie erhält als Parameter die Funktion *positive*, die als Prädikat für das Filtern verwendet wird. Dies ist möglich, da Funktionen und Prozeduren ebenfalls Objekte sind und somit als Parameter oder Rückgabewerte für Funktionen verwendet werden können.

Beispiel 2 zeigt die Definition einer Klasse `Vector2` für Vektoren im Vektorraum $\mathbb{R}^2$. Die Klasse besitzt zwei Attribute vom Typ *Float*. Die Sichtbarkeit von Attributen und Methoden wird mit Symbolen ausgedrückt, wie sie auch in *UML*-Klassendiagrammen [RJB04] verwendet werden. Dabei steht `+` für *public*, `#` für *protected*, `-` für *private* und `~` für *package*. Die Methode `initializer` wird bei der Instantiierung aufgerufen.

Die Klasse `Vector2` implementiert außerdem drei spezielle Methoden, die Operatoren überladen: Die Addition mit einem anderen Vektor, die Multiplikation mit einem Skalar und das Skalarprodukt. Bei einem Aufruf der Form `a + b` wird die Methode `operator +` auf dem Objekt `a` aufgerufen und `b` als Parameter übergeben. Dies erlaubt eine intuitive Notation von Operationen auf Vektoren, was in den letzten Zeilen des Code-Beispiels deutlich wird.

```
class Vector2:
    # Float x
    # Float y

    + initializer(Float x, Float y):
        self.x := x
        self.y := y

    + Vector2 operator +(Vector2 other): // vector addition
        return Vector2(self.x + other.x, self.y + other.y)

    + Vector2 operator *(Float other): // multiplication with a scalar
        return Vector2(self.x * other, self.y * other)

    + Float operator *(Vector2 other): // scalar product
        return self.x * other.x + self.y * other.y

Vector2 p := Vector2(2.5, 3.75)
Vector2 q := Vector2(1.75, 6.89)
Vector2 r := (p*2.0)+q // Vector2(6.75, 14.39)
Float f := p*q // 30.2125
```

Beispiel 2: Definition und Benutzung von Klassen in *Monty* (2D-Vektoren)

Monty besitzt ein statisches Typsystem, um eine schnelle Ausführung der Programme zu ermöglichen. Um dem Komfort von Skriptsprachen näher zu kommen, erlaubt das Typsystem die dynamische Erweiterung von Objekten. Dies wird in Beispiel 3 benutzt, um für ein Objekt zusätzliche Attribute zu definieren. Angenommen, es gäbe eine Funktion `normalize`, die einen Vektor als Parameter erwartet und den normalisierten Vektor zurückliefert (d.h. einen Vektor, der in die gleiche Richtung zeigt, aber die Länge 1 hat). Die Klasse `Vector2` besitzt kein Attribut, das angibt, ob ein Vektor normalisiert ist oder nicht. Mit dem Operator "`->`" kann für das `Vector2`-Objekt `q` ein dynamisches Attribut `isNormalized` vom Typ `Object` definiert werden. In der Bedingung "`q->isNormalized as Bool`" wird darauf dynamisch zugegriffen. Dabei muss zur Laufzeit überprüft werden, ob das Attribut vorhanden ist und den richtigen Typ hat. Das Schlüsselwort "`as`" stellt einen Typcast dar. Ein Ausdruck der Form *exp* `as` *type* wandelt den Wert des Ausdrucks *exp* in den Typ *type* um, sofern dies nach den Regeln des Typsystems möglich ist. Zugunsten des Entwicklungskomforts kann also explizit auf die Typsicherheit und Performanz statischer Typisierung verzichtet werden, um eine Art *Duck Typing* zu erreichen [CRJ12].

```
Vector2 p := Vector2(2.5, 3.75)
Vector2 q := normalize(Vector2(1.75, 6.89))
q->isNormalized := true       // introducing a dynamic attribute

if q->isNormalized as Bool:   // accessing a dynamic attribute
    print("The vector is normalized!")
```

Beispiel 3: Einführen und benutzen dynamischer Attribute in *Monty*

Attribute, die nicht dynamisch zu einer Klasse hinzugefügt werden, müssen zum Zeitpunkt der Instantiierung initialisiert sein, da es in *Monty* keine Null-Referenzen gibt. Dies kann entweder bei der Deklaration der Attribute oder in der Initialisierungsmethode geschehen. Beispiel 4 illustriert, wie ohne Null-Referenzen ein Binärbaum implementiert werden kann. Für den trivialen Fall (in diesem Fall die Blätter des Baumes) muss explizit ein Typ eingeführt werden. Dies stellt zwar auf der einen Seite einen geringen Overhead dar, auf der anderen Seite bietet es aber den Vorteil, dass stets Gewissheit über den Typen eines Knoten besteht.

```
abstract class BinaryTree<T>:
    + abstract height()


class Branch<T> inherits BinaryTree:
    BinaryTree<T> left
    T value
    BinaryTree<T> right

    + initializer(BinaryTree<T> left, T value, BinaryTree<T> right):
        self.left = left
        self.value = value
        self.right = right

    + height():
        return 1+max(self.left.height(), self.right.height)


class Leaf<T> inherits BinaryTree:
    + height():
        return 0
```

Beispiel 4: Ein Binärbaum ohne Null-Referenzen

Die vollständige Definition der Sprache [Mon14b] enthält noch einige weitere Konzepte, die in diesem Papier aus Platzmangel nicht beschrieben werden können. Dazu zählen eine nähere Beschreibung abstrakter Klassen, allgemeiner Mehrfachvererbung, von Modulen und Paketen. Darüber hinaus sind weitere Konzepte in der gegenwärtigen Fassung der Sprache noch nicht enthalten: lokale Typinferenz, Lambda-Funktionen, eingeschränkt generische Parameter und die explizite Verwendung von *Mutability* und *Immutability*. Solche zukünftigen Erweiterungen von *Monty* sind im Bericht [Mon14a] beschrieben.

Auch die von David Watt in [Wat05] vorgeschlagene Sprache *Monty* (die nie implementiert wurde) hat sich syntaktisch an Python orientiert und dynamische Typisierung erlaubt. Der Schwerpunkt dieser Studie liegt in Überlegungen zur Unterscheidung von Klassen mit und ohne veränderlichen Attributen (*mutable* und *immutable*) und zur Varianz bei den Typparametern generischer Klassen. Syntaktisch und auch in Hinblick auf die Objektorientierung (Werte als Objekte, allgemeine Mehrfachvererbung) orientiert sich unser *Monty* aber eher an *Python* als an *Java*.

3 Die Implementierung eines Compilers für *Monty*

Der Compiler für *Monty* wurde in *Java* implementiert. Damit wurde schon während der Arbeit an der Sprachdefinition begonnen. Im Folgenden soll ein Einblick in die Architektur und Funktionsweise des Compilers gegeben werden (vgl. Abbildung 2).

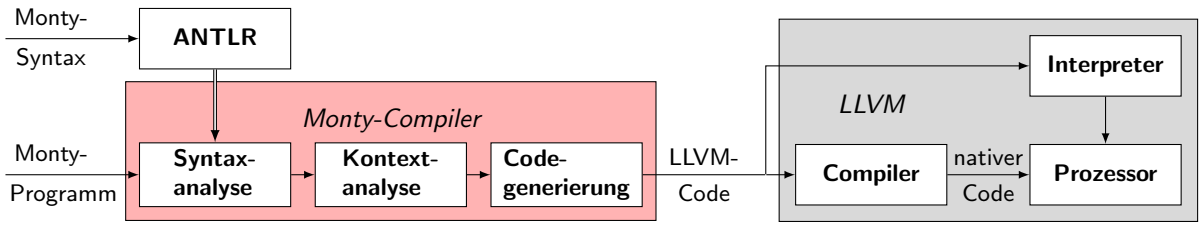

Abbildung 2: Struktur des *Monty*-Compilers

Wird der Compiler ausgeführt, so erwartet er eine Datei mit *Monty*-Code. Eine mit dem *ANTLR*-Framework [Par14a] generierte und in den Compiler eingebundene Syntaxanalyse generiert aus der eingelesenen Datei einen Ableitungsbaum. In den folgenden Schritten werden einige Entwurfsmuster (*Design Patterns*) eingesetzt[1]. Hervorzuheben ist hier das *External Tree Visitor-Pattern*[2]. Auf Grundlage des Ableitungsbaumes wird zuerst ein abstrakter Syntaxbaum (AST) erstellt, auf dessen Basis alle weiteren Operationen ausgeführt werden. Mehrere *Visitors* laufen dabei über den AST, wobei verschiedene Aktionen auf den besuchten Knoten ausgelöst werden. Die *Visitors* übernehmen dabei Aufgaben wie:

- *Identifikation*: Welche Deklaration ist in dem aktuellen Kontext diesem Bezeichner zugeordnet?
- *Typüberprüfung*: Passen die Typen zueinander, etwa in Zuweisungen oder bei Funktionsaufrufen, oder ist mit Fehlern zu rechnen?
- *Kontrollflussvalidierungen*: Kommen `break`- und `skip`-Befehle nur in Schleifenrümpfen vor? Wird der `return`-Befehl nur innerhalb von Funktionsrümpfen verwendet und dort auf allen Verzweigungen erreicht?

Der letzte *Visitor*, der über den AST läuft, generiert *LLVM*-Code. *LLVM* [LA04] ist eine Architektur und Sammlung von Werkzeugen für die Codeoptimierung und Codeerzeugung in Compilern. Der generierte Code liegt dabei in einer speziellen Zwischenrepräsentationssprache vor, die Bestandteil dieser *LLVM*-Werkzeugsammlung ist. Das Ergebnis des *Monty*-Compilers ist also ein in *LLVM*-Code übersetztes *Monty*-Programm. Dieses kann anschließend dank der *LLVM*-Architektur sowohl direkt interpretiert als auch erst kompiliert und dann ausgeführt werden. Die Verwendung von *LLVM* ermöglicht es ebenso, *Monty*-Programme zu optimieren und für eine Vielzahl von Plattformen zu übersetzen.

4 Fazit

Im Projekt *Monty's Coconut* wurde die Programmiersprache *Monty* entworfen, definiert und ein Compiler für einen Großteil der Sprache implementiert. Wir wollen hier zunächst zusammenfassen, welche Erfahrungen während des Projekts gewonnen wurden, sowie anschließend die Sprache und den Compiler bewerten.

Der Entwurf einer Programmiersprache ist zeitaufwändig. Es dauerte drei Monate, um einen Katalog von Konzepten für *Monty* zu erstellen. In langen wöchentlichen Plenumssitzungen, deren Themen in jeweils wechselnden Kleingruppen vorbereitet wurden, haben die Teilnehmerinnen und Teilnehmer sehr engagiert den Entwurf der Sprache diskutiert. Wegen unterschiedlicher Vorkenntnisse und verschiedener Lieblingssprachen waren die Diskussionen oft kontrovers und langwierig. Gelegentlich entstand so der Eindruck, man käme kaum voran oder drehe sich sogar im Kreis. Gleichzeitig wuchs die Besorgnis, Zeit zu vergeuden, und der Wunsch, endlich konkreter an sichtbaren "Produkten" zu arbeiten. Diesen Prozess hätte man potenziell rationalisieren können, wäre schon gleich zu Projektbeginn ein Interpreter für eine elementare objektorientierte Sprache implementiert worden. Anhand dieser Implementierung hätte man vermutlich zielgerichteter diskutieren können, welche Auswirkungen bestimmte Erweiterungswünsche auf Definition und Implementierung der Sprache haben. Zur Strukturierung der Diskussion wäre es auch nützlich gewesen, Teilnehmerinnen und Teilnehmer zu "*Fachleuten*" für Sprachen wie *Python* und *Ruby* oder für Konzepte wie Mehrfachvererbung und Typinferenz zu ernennen, damit sie ihr Detailwissen jederzeit in die Diskussion einbringen können.

Die Definition einer Programmiersprache ist schwierig. Manche Probleme mit Konzepten – und speziell mit deren Kombination – haben sich erst gezeigt, als man die Sprache in sich schlüssig, präzise und gleichzeitig

[1] Terence Parr, der Entwickler von *ANTLR*, schlägt diese *Patterns* in Kombination mit *ANTLR* vor [Par14b].
[2] Bei dem *External Tree Visitor* handelt es sich um eine Variante des *Visitors* von Gamma et al. [GHJV95].

verständlich beschreiben musste. Hilfreich wäre wohl gewesen, zunächst nur eine elementare Kernsprache zu definieren, um sie dann schrittweise zu erweitern. Ein weiteres, organisatorisches Problem lag darin, dass die Arbeit an der Definition nur wenig Vorsprung vor der Implementierung hatte, so dass die Reihenfolge, in der Konzepte definiert werden mussten, aufwändig mit der Compilergruppe synchronisiert werden musste. Auch hier hätte womöglich der Entwurf einer erweiterbaren Kernsprache die Abhängigkeiten zwischen den Gruppen reduzieren können, um so den Fokus von der Organisation auf die Konzepte zu verlagern.

Die Entwicklung eines Compilers ist sehr anspruchsvoll. Das kann in einem Jahr kaum bewältigt werden. Glücklicherweise standen Werkzeuge zur Verfügung: *ANTLR* für die Syntaxanalyse, und *LLVM* für die Codererzeugung. Durch Aufteilung in mehrere *Visitors*, die voneinander unabhängige Teilaufgaben übernehmen, konnte der Compiler gut im Team entwickelt und die Arbeit parallelisiert werden, so dass während der Implementierung nur selten Engpässe entstanden. Testgetriebene Entwicklung trug zur Qualitätssicherung bei, und ein diszipliniertes Vorgehensmodell – umgesetzt mit Hilfe von *Git* – sorgte dafür, dass es immer eine stabile Fassung des Compilers gab.

Monty *ist ein erster Schritt in die richtige Richtung.* Im Rahmen des Projekts konnten nicht alle Ideen umgesetzt werden. So wurden einige Aspekte wie die Typinferenz oder Lambda-Funktionen in der Spezifikation und folglich im Compiler gar nicht berücksichtigt und haben nur über das *Enhancement Proposal* [Mon14a] ihren Weg in die Sprache gefunden. Daher kann wohl nicht davon gesprochen werden, dass die Lücke zwischen den gegenwärtigen Sprachkonzepten tatsächlich geschlossen wurde, da noch viel Verbesserungspotential besteht. Grundsätzlich sind wir unserem Ziel aber schon nahe gekommen, da *Monty* sich zumindest innerhalb dieser Lücke positioniert. In *Monty* konnten eine leichtgewichtige Syntax und ein einheitliches Objektsystem mit einem statischen Typsystem in einer kompilierten Sprache vereint werden. Darüber hinaus enthält *Monty* den *SDA*, der – wenngleich mit Einschränkungen – dynamische Typisierung in direkter Kombination mit statischer Typisierung ermöglicht.

Der Compiler ist nur ein Prototyp. Er übersetzt eine Teilsprache von *Monty*, produziert aber noch keinen guten Code. Optimierungen zu implementieren erscheint uns erst sinnvoll, wenn die Implementierung ansonsten vollständig und stabil ist. Eine Standardbibliothek wurde auch noch nicht bereitgestellt, denn dies erfordert viel Routinearbeit, die später nachgeholt werden kann.

Auch wenn die Arbeit an der Sprache und am Compiler bis zum Projektende nicht ganz abgeschlossen werden konnte, haben die Studierenden sehr viel gelernt über Programmiersprachen und ihre Übersetzer, auch über die Schwierigkeiten der selbstständigen Arbeit in einem größeren Team.

Im Übrigen geht die Arbeit auch nach offiziellem Projektende weiter: Die Sprachdefinition ist fertig, und am Compiler wird weiter gearbeitet. Umfangreichere Erweiterungen der Sprache und Verbesserung des Compilers sind Gegenstand von mehreren Abschlussarbeiten zu Themen wie beschränkten generischen Typparametern, automatischer Speicherbereinigung (*garbage collection*) und lokaler Typinferenz. Der aktuelle Stand des Projekts *Monty's Coconut* kann auf der Webseite `http://www.informatik.uni-bremen.de/monty/` eingesehen werden.

Danksagung. Die Autoren möchten sich bei David Watt dafür bedanken, dass sie den Namen *Monty* für ihre Sprache verwenden dürfen. Des Weiteren gilt unser Dank allen Teilnehmerinnen und Teilnehmern des Projekts *Monty's Coconut* für ihre aktive Mitwirkung an der Entwicklung der Sprachspezifikation und des Compilers.

Literatur

[CRJ12] Ravi Chugh, Patrick M. Rondon und Ranjit Jhala. Nested Refinements: A Logic for Duck Typing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, Seiten 231–244, New York, NY, USA, 2012. ACM.

[FM08] David Flanagan und Yukihiro Matsumoto. *The Ruby Programming Language.* O'Reilly, 2008.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJS+14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha und Alex Buckley. *The Java Language Specification.* Addison-Wesley, 2014. 8. Auflage.

[LA04] Chris Lattner und Vikram Adve. CGO '04: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Washington, DC, USA, Mar 2004. IEEE Computer Society. 520043.

[Mon14a] Monty's Coconut. Enhancements and Extensions Proposed for the Programming Language Monty. Projektbericht, Fachbereich 3 – Mathematik und Informatik, Universität Bremen, September 2014. `http://www.informatik.uni-bremen.de/monty/spec/Montys_Coconut_-_Enhancement_and_Extension_Proposal.pdf`.

[Mon14b] Monty's Coconut. The Monty Language Specification. Projektbericht, Fachbereich 3 – Mathematik und Informatik, Universität Bremen, September 2014. `http://www.informatik.uni-bremen.de/monty/spec/Montys_Coconut_-_The_Monty_Programming_Language_%28Pre-Release%29.pdf`.

[OSV10] Martin Odersky, Lex Spoo und Bill Venners. *Programming in Scala*. Artima developer, 2010. 2. Auflage.

[Par14a] Terrence Parr. *The Definitive ANTLR4 Reference*. The Pragmatic Programmers, LLC, September 2014. Version 2.0.

[Par14b] Terrence Parr. *Language Implementation Patterns*. The Pragmatic Programmers, LLC, September 2014. Version 5.0.

[RJB04] James Rumbaugh, Ivar Jacobson und Grady Booch. *The Unified Modeling Language reference manual*. The Addison-Wesley Object Technology Series. Addison-Wesley, Boston, 2004. 2. Auflage.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, Inc., 2013. 4. Auflage.

[vRD11] Guido van Rossum und Fred L. Drake. *Python Language Reference Manual*. Network Theory Ltd, Boston, March 2011. Release 3.2.

[Wat05] David A. Watt. The Design of Monty: a Programming/Scripting Language. *Electronic Notes in Theoretical Computer Science*, 141(4):5–28, 2005.