Guile

Die Erweiterungssprache des GNU-Projekts

Matthias Köppe

Otto-von-Guericke-Universität Magdeburg
Magdeburger Linux User Group e. V.

19. Mai 2001

Überblick

- Guile als Erweiterungssprache
- Guile als Scheme-Implementierung
  - Syntax
  - Spracheigenschaften
  - Interaktivität
- SWIG als *wrapper generator* für Guile

Erweiterungssprachen

- Situation: Programmsystem bzw. Programmbibliotheken liegen vor
- Anforderung: Integration von Sprachen zur **Konfiguration**, **Steuerung**, **Erweiterung**, **Interaktion** und zum **Testen**
- Typische Lösung:
  - Beginne mit „kleinen“ Sprachen,
  - erweitere allmählich zu einer „großen“ Ad-hoc-Sprache
- Bessere Lösung:
  - Bette von Anfang an eine richtige Programmiersprache ein!

Scheme – Implementierungen und Einsatzgebiete

Ein Sprachstandard (R^5RS), viele unabhängige Implementierungen:

- Interpreter – MIT Scheme, SCM, Guile, Scheme48, . . .
- Compiler – Scheme2C, MzScheme, Static Language Implementation, . . .
- Kawa, Bigloo übersetzen Scheme in Java-Bytecode

Einsatzgebiete

- als Skriptsprache, **Erweiterungssprache** oder vollwertiges Programmiersystem
- als Websprache: Beautiful Request Language (BRL)

Scheme – Verwandte Sprachen

Scheme gehört zur Lisp-Familie.

- Common Lisp – der große Bruder von Scheme
  - umfangreiche Standardbibliothek
  - verschiedene kommerzielle Anbieter, auch freie Implementierungen verfügbar
  - eingesetzt für komplexe, robuste industrielle Anwendungen
- Emacs Lisp – Implementierungs- und Erweiterungssprache von GNU Emacs und XEmacs

Einfache Syntax

Programme (**Ausdrücke**) sind:

- **Atome**: `91`, `"Text"`, `write`
  - Zahlen, Strings, . . . stehen für sich selbst
  - **Symbole** stehen für **Werte**
- **Listen**: `(max 17 3 55 10)`
  - **Prozedur**aufrufe: `(write "Text")`
  - **Spezialformen**:
```
(if (even? x) (write "gerade")
              (write "ungerade"))
```
  Das Symbol (der Wert) am Anfang der Liste entscheidet (**Präfixnotation**).

Einfache Syntax – Leistungsfähige Makros

- Parsen von Scheme-Programmen ist trivial (*reader*)
- Programme können in natürlicher Weise als **Daten** verarbeitet werden (Listenverarbeitung)

$\longrightarrow$ Leistungsfähige Makros (Syntax-Transformationen)

$\longrightarrow$ Problemspezifische Sprachen können integriert werden

Syntax – Vergleich mit C/C++/Java

- Parsen von C/C++/Java ist schwer
- Geparstes Programm steht nur in speziellen Datenstrukturen zur Verfügung

→ Für C/C++ ist nur ein simples Textersetzungssystem (Präprozessor) verfügbar

→ Java hat kein Makrosystem

→ *Cut-and-Paste*-Techniken beim Programmieren sind populär

Spracheigenschaften

- Lexikalischer Abschluß, Prozeduren als Werte. . .
- Dynamische Typen: Typen sind mit Werten, nicht mit Variablen verbunden
- Automatische Speicherverwaltung: *Garbage collection*
- Mehrfache Rückgabewerte
- Ausnahmenbehandlung und *first-class continuations*

Spracheigenschaften – Lexikalischer Abschluß

Sortieren in C mit Standardfunktion:

```
void qsort(void *, size_t, size_t,
           int (*cmp)(*, *));
```

Was, wenn die Vergleichsfunktion `cmp` von Parametern abhängen soll? Etwa Sortieren nach der i-ten Komponente eines Vektors:

```
void sort_vectors(int **vectors,
                  size_t count,
                  size_t i);
```

Der Parameter i muß in die Sortierfunktion hineingereicht werden!

Spracheigenschaften – Lexikalischer Abschluß II

Als globale Variable hineinreichen:

```
int the_i;  /* yuck! */

int vector_cmp(int **a, int **b) {
  return (*a)[the_i] - (*b)[the_i];
}

void sort_vectors(int **vectors, size_t count, int i)
{
  the_i = i;  /* yuck! */
  qsort(vectors, count, sizeof(int*), vector_cmp);
}
```

Oder neue Sortierfunktion schreiben!

Spracheigenschaften – Lexikalischer Abschluß III

In Scheme:

```
(define (sort-vectors vectors i)
  (sort vectors
        (lambda (a b)
          (< (vector-ref a i)
             (vector-ref b i)))))
```

Eine **anonyme Prozedur** mit Argumenten `(a b)`, die über i **abschließt**, wird zum Vergleichen benutzt.

Spracheigenschaften – Lexikalischer Abschluß IV

Funktionen höherer Ordnung (nehmen Prozeduren als Argument):

- Auswahl aus Listen

```
(pick even? '(12 13 17 18 21))
 => (12 18)
```

- Iteration, z. B. über Listen:

```
(map (lambda (x y) (* x y))
     '(1  2  3  4  5)
     '(2  3  4  5  6))
 =>   (2  6 12 20 30)
```

Interaktivität

- *Read-eval-print loop*
- Reflexion (Introspektion)
  - Abfragen von Werten, Dokumentation, Signatur
- Code und Daten zur Laufzeit änderbar

$\longrightarrow$ wesentlich schnellere Entwicklung möglich als mit traditionellem Debugging-Zyklus

Wrapper[1]

Wrapper-Code ist nötig,

- um Datentypen zwischen der Implementierungssprache und der Erweiterungssprache zu konvertieren,
- um die Funktionen, Variablen, Strukturen, . . . des Programms der Erweiterungssprache zugänglich zu machen.

Wrapper-Code zu schreiben ist trivial, aber lästig

- insbesondere bei sich häufig ändernden Schnittstellen

[1] *to wrap* – einwickeln

SWIG als Wrapper-Generator

SWIG (*Simplified Wrapper and Interface Generator*)

- liest C/C++-Header mit speziellen Typ-Annotationen
- erzeugt Wrapper-Code für verschiedene Skript- und Erweiterungssprachen
  - Perl, Python, Tcl, Java, MzScheme, Guile

Warum nicht vollautomatisch möglich?

```
void multiply(int number, int *x);
void sort(int number, int *x);
```

Wie x behandeln?

Typ-Annotationen in SWIG

Ein- und Ausgabe via Zeigerargument mit Standard-Annotation `BOTH`:

```
void multiply(int number, int *BOTH);
```

→ `multiply` ist auf Scheme-Seite eine Prozedur mit Signatur

```
(multiply number number) => number
```

Neue Typ-Annotationen können vom Anwender geschrieben werden, etwa für `sort`:

```
void sort(int VECTOR_LEN, int *VECTOR_IN_OUT);
```

→ `sort` ist auf Scheme-Seite eine Prozedur mit Signatur

```
(sort vector) => vector
```

Ende