

Parse- und Compilezeitprogrammierung mit Factor

(BlockTag3, SEKS, WS2009, 2009-12-10)

Einleitung

In Factor ist es möglich, Programme explizit zur Parse- oder Compilezeit auszuführen. Parsezeitprogrammierung ermöglicht die Einführung von nicht-konkatenativen Syntaxen durch Kontrolle des Parsevorgangs. Compilezeit-Macros werden zur Codeoptimierung verwendet.

Parsevorgang

Factor-Programme sind Sequenzen von mit Leerzeichen getrennten Tokens. Das Programm „20 3 +“ besteht aus den drei Tokens „20“, „3“ und „+“. Das Programm „[20 3 +]“ besteht aus den Tokens „[“, „20“, „3“, „+“ und „]“. Der Factor-Parser liest diese Tokens von links nach rechts ein und erstellt einen Parsebaum. Dieser Parsebaum ist eine Quotation, die compiliert und zur Laufzeit ausgeführt wird. Grundlegend unterscheidet der Parser zwischen Strings, Zahlen und Wörtern. Wörter werden noch einmal in „normale“ Wörter und Parse-Wörter unterteilt.

Strings spielen eine Sonderrolle, da zwischen den Anführungszeichen beliebig Leerzeichen stehen können. Der Parser arbeitet hier auf Zeichenebene: Wird ein Anführungszeichen entdeckt, liest der Parser so lange vom Zeichenstrom bis wieder ein Anführungszeichen vorkommt, welchem kein Escape-Zeichen voransteht. Dann wird der gelesene String in den Parsebaum eingehängt und der Parsevorgang geht mit dem nächsten Token nach dem String weiter.

Steht ein Token für eine Zahl, so wird ein entsprechendes Zahl-Objekt in den Parsebaum eingehängt.

Ist das Token ein Wort, so wird versucht dieses im Vokabularsuchpfad nachzuschlagen. Handelt es sich um ein „normales“ Wort, wird dieses in den Parsebaum eingehängt. Handelt es sich um ein Parse-Wort, wird dieses *sofort ausgeführt*.

Parse-Wörter

Eine der Besonderheiten von konkatenativen Sprachen ist, dass Programme als Sequenzen aufeinander folgender Kommandos (bzw. Funktionen) verstanden werden können, die in genau der Reihenfolge ausgeführt werden, in der sie im Programmtext stehen. Beispielsweise führt das Programm „2 3 +“ zunächst „2“, dann „3“, dann „+“ aus. Programme in Infix-Schreibweise (wie z.B. „2 + 3“) oder Prefix-Schreibweise (wie z.B. „+ 2 3“) haben nicht den gewünschten Effekt, da das „+“ bereits ausgeführt wird, bevor die beide Zahlen auf dem Stack liegen. Anders ausgedrückt: Aufgrund des sequentiellen Ausführungsmodells und der engen Beziehung zwischen Syntax und Semantik ist es in konkatenativen Programmen nicht ohne weiteres möglich, auf Dinge Bezug zu nehmen, die im Programmtext weiter rechts stehen.

Parsewörter ermöglichen es, diese Einschränkung zu durchbrechen. Ein Parsewort ist ein Wort, welches zur Parsezeit ausgeführt wird. Stößt der Parser auf ein Token welches für ein Parse-Wort steht, so wird die Implementierung des Parseworts ausgeführt – das Parsewort *erhält die Kontrolle* über den weiteren Parsevorgang!

Das Beispiel

```
SYNTAX: ADD scan-word parsed scan-word parsed \ + parsed ;
```

führt ein Parse-Wort mit dem Namen ADD ein. Parsewörter werden per Konvention in Großbuchstaben geschrieben.

Dieses Parse-Wort ermöglicht Programme wie das folgende:

```
ADD 2 3
```

mit Ergebnis 5.

Parsewörter werden zur Parsezeit ausgeführt, wenn sie vom Parser als solche erkannt werden. Der Stack-Effekt eines Parsewortes ist (Akkumulator -- Akkumulator'). Der Akkumulator ist so etwas wie der Zustand des Parsers. Er enthält den bereits erstellten Parsebaum und den noch zu parsenden Rest des Programms. Wörter wie „scan“, „scan-word“, „parse-quotation“, „parse-until“ und „parsed“ erlauben es, den Parser zu steuern und den Akkumulator – und damit Parsebaum und noch zu parsendes Programm – kontrolliert zu modifizieren.

Im obigen Beispiel versucht „scan-word“ das nächste Token einzulesen. Handelt es sich tatsächlich um ein Wort, so wird dieses auf dem Stack abgelegt; wenn nicht, gibt es einen Fehler. Das Wort „parsed“ bewirkt, dass das Wort vom Stack in den Parsebaum im Akkumulator gehängt wird. Beim Aufruf von „ADD 2 3“ bewirkt „scan-word“, dass „2“ und der Akkumulator auf dem Stack liegen. Das Wort „parsed“ fügt die „2“ dem Parsebaum im Akkumulator hinzu. Dasselbe passiert beim erneuten Aufruf von „scan-word“ und „parsed“ mit der „3“. Durch „\ +“ wird das Wort „+“ zum Akkumulator auf den Stack gelegt, „parsed“ fügt das „+“ dem Parsebaum hinzu. Was als Parsebaum bei „ADD 2 3“ entsteht entspricht dem Parsebaum, der beim Programm „2 3 +“ aufgebaut wird. Das Parsewort führt also folgende Umschreibung durch:

```
ADD $x $y ==> $x $y +
```

Dies wird ersichtlich, wenn das Parse-Wort innerhalb einer Quotation verwendet wird:

```
[ ADD 2 3 ]
```

ergibt

```
[ 2 3 + ]
```

Parsewörter bieten die Möglichkeit genau zu kontrollieren, was der Parser liest, das Gelesene auf dem Stack zu modifizieren und schließlich zu entscheiden, was vom Gelesenen in den Parsebaum eingehängt wird.

Durch einen einfachen Trick kann der Akkumulator sichtbar gemacht werden. Das Parsewort

```
SYNTAX: SHOWME dup parsed ;
```

dupliziert den Akkumulator und fügt diesem dem Parsebaum hinzu. Das Ergebnis des Programms

```
1 2 SHOWME 3 4
```

ist

```
1 2 V{ clear 1 2 ~vector~ 3 4 } 3 4
```

Der Vektor im Vektor markiert die aktuelle Parseposition. „1“ und „2“ wurden dem Parsebaum hinzugefügt, „3“ und „4“ sind noch zu parsen.

Die Modifikation

```
SYNTAX: SHOWME scan-word drop dup parsed ;
```

bewirkt, dass das nächste Token gelesen und verworfen wird, bevor der „Schnappschuss“ des Akkumulators in den Parsebaum gehängt wird. Für das Programm

```
1 2 SHOWME 3 4
```

ist das Ergebnis nun:

```
1 2 V{ clear 1 2 ~vector~ 4 } 4
```

Die „3“ wurde vom Parser verworfen.

Eine weitere Modifikation

```
SYNTAX: SHOWME 3.5 parsed dup parsed ;
```

führt dazu, dass die Zahl „3.5“ vor dem Schnappschuss des Akkumulators in den Parsebaum gehängt wird. Für das Programm

```
1 2 SHOWME 3 4
```

ist das Ergebnis nun:

```
1 2 3.5 V{ clear 1 2 ~vector~ 4 } 4
```

Die Zahl „3.5“ wurde dem Parsebaum programmatisch hinzugefügt.

Das Wort „scan-word“ erzwingt, dass es sich beim folgenden Token um ein Wort handelt. Das wesentlich weniger restriktive „scan“ kann hingegen beliebige Token einlesen. Diese werden dann *als Strings* auf den Stack gelegt.

```
SYNTAX: WHATEVER scan parsed ;
```

```
WHATEVER 3
```

entspricht dem Programm „3“.

```
WHATEVER +
```

entspricht dem Programm „+“.

Die Worte „scan“ und „scan-word“ weisen den Parser an, ein Token zu parsen und das Ergebnis auf dem Stack abzulegen. Mit dem Word „parse-until“ kann der Parser angewiesen werden, alle Token bis zu einem bestimmten Terminierungstoken einzulesen. Die Ergebnisse werden in einem Vektor auf dem Stack abgelegt.

Das Wort

```
SYNTAX: VECTORIZE \ ; parse-until parsed ;
```

liest alle Tokens bis zum Vorkommen von „;“ und legt diese als Vektor ab. Dieser Vektor wird dann durch das Wort „parsed“ dem Parsebaum hinzugefügt.

```
VECTORIZE 1 2 3 4 ;
```

hat das Ergebnis

```
V{ 1 2 3 4 }
```

Der Strichpunkt ist in Factor ein generischer Begrenzer, der auch bei vielen Parsewörtern der Factor-Bibliotheken zum Einsatz kommt, beispielsweise bei „SYNTAX:“ selbst oder bei Wort-Definitionen. Eigene Begrenzer können durch das Wort „delimiter“ definiert werden. Es macht das zuletzt definierte Wort zum „delimiter“ und sorgt dafür, dass dieses nur in Verbindung mit einem Parsewort verwendet werden kann.

Beispielsweise definiert

```
: VECTOR-END ( -- ) ; delimiter
```

„VECTOR-END“ als Begrenzer. Dies erlaubt z.B. die Definition

```
SYNTAX: VECTOR-BEGIN \ VECTOR-END parse-until parsed ;
```

und Programm wie

```
VECTOR-BEGIN 1 2 3 4 VECTOR-END
```

Im Beispiel

```
: ++ ( -- ) ; delimiter
SYNTAX: INC \ ++ parse-until [ 1 + ] map [ parsed ] each ;
```

wird ein Begrenzer „++“ definiert. Das Parsewort „INC“ liest alle Wörter bis zum Vorkommen des Begrenzers vom Stack und legt diese in einem Vektor ab. Auf alle Elemente des Vektors wird dann das Programm „[1 +]“ ausgeführt. Danach wird auf jedes Element des Vektors „parse“ ausgeführt, was dazu führt, dass die Elemente des Vektors einzeln in den Parsebaum eingehängt werden.

Das Programm „INC 1 2 3 ++“ wird also zur Parsezeit in „2 3 4“ umgeschrieben. Die Inkrementberechnung findet zur Parsezeit(!) statt.

Mit dem Wort „parse-quotation“ können Quotations geparkt werden. Dabei ersetzt das Parsewort die öffnende Klammer der Quotation. Beispielsweise schreibt das Wort

```
SYNTAX: [! parse-quotation parsed ;
```

das Programm

```
[! 2 3 + ]
```

in das Programm

```
[ 2 3 + ]
```

um.

Das Parse-Wort

```
SYNTAX: APPLY: scan "[" assert= parse-quotation scan  

  "TOEACH:" assert= scan "[" assert= parse-quotation swap map parsed ; inline
```

schreibt das Programm

```
APPLY: [ dup * ] TOEACH: [ 2 3 4 ]
```

in

```
[ 2 3 4 ] [ dup * ] map
```

um. Durch die Asserts wird korrekte Syntax sichergestellt. So würde beispielsweise

```
APPLY: [ dup * ] TO: [ 2 3 4 ]
```

zu einem Fehler führen.

Da in der Definition von „APPLY:“ durch „map“ eine eingeleseene Quotation angewendet wird, muss das Wort „inline“ deklariert werden.

Macros

Macros sind Programme, die potentiell zur Compilezeit ausgeführt werden. Ein Macro entspricht einer normalen Wortdefinition. Der Unterschied ist nur, dass ein Macro eine Quotation erzeugt. Diese Quotation wird zur Laufzeit ausgeführt, es wird also implizit ein „call“ angewendet. Die Berechnung der Quotation selbst wird – wenn möglich – zur Compilezeit durchgeführt. Ist dies nicht möglich, wird die Quotation zur Laufzeit berechnet und ihr Ergebnis gespeichert, damit ihr Ergebnis nicht immer wieder neu berechnet werden muss. Macros werden durch das Parsewort „MACRO:“ definiert.

Abgesehen von der möglichen Ausführung zur Compilezeit, sind folgende Aufrufe identisch:

```
MACRO: mymacro ... ;  

: mymacro ... call ;
```

Die Berechnung zur Compilezeit ist möglich, wenn folgende Bedingungen erfüllt sind:

- Alle Macro-Inputs sind Literale
- Das Wort, welches das Macro aufruft, hat einen statischen Stack-Effect
- Die berechnete Quotation hat einen statischen Stack-Effect

Fried-Quotations

Macros sind Worte, welche eine Quotation berechnen. Diese Berechnung kann zur Compilezeit stattfinden. Fried-Quotations sind ein einfacher Template-Mechanismus, der die Berechnung von Quotations erleichtert. Fried-Quotations sind Quotations mit Lücken, welche mit Werten vom Stack gefüllt werden. Eine Fried Quotation beginnt mit „[“. Beispiel:

```
' [ 1 2 3 ]
```

Diese Fried-Quotation hat keine Lücke und ist daher äquivalent zur normalen Quotation

```
[ 1 2 3 ]
```

Folgende Fried-Quotation hat an der mit dem Unterstrich markierten Stelle eine einfache Lücke:

```
' [ 1 _ 3 ]
```

Das Programm

```
clear ' [ 1 _ 3 ]
```

führt zu einem Stackunterlauf, da ein Wert zum Füllen der Lücke erwartet wird.

```
2 ' [ 1 _ 3 ]
```

hingegen ergibt die Quotation

```
[ 1 2 3 ]
```

Mehrere Lücken können durch mehrere Underscores definiert werden. Die Lücken werden von rechts nach links (Quotation) und oben nach unten (Stack) gefüllt. Intuitiv entspricht dem Programm

```
2 4 ' [ 1 _ 3 _ 5 ]
```

also das Programm

```
[ 1 2 3 4 5 ]
```

Allerdings ist dies nicht völlig korrekt. Factor produziert aus einer Fried-Quotation und den Werten zur Füllung der Lücken eine Quotation, die zwar – wenn sie als Programm interpretiert wird (mit „call“ aufgerufen wird) – die intuitiv richtige Bedeutung hat, nicht jedoch als Datenstruktur. Obiges Beispiel ergibt nämlich nicht

```
[ 1 2 3 4 5 ]
```

sondern

```
[ 2 4 [ [ 1 ] dip 3 ] dip 5 ]
```

Beide Quotations haben dasselbe Ergebnis wenn nach ihnen ein „call“ steht, nicht aber, wenn z.B. ein „first“ folgt.

Das Programm

```
[ 2 3 ] '[ 1 _ 4 ]
```

liefert eine Quotation, die programatisch der Quotation

```
[ 1 [ 2 3 ] 4 ]
```

entspricht.

Um die Lücke mit den Werten einer Quotation zu füllen, kann eine spezielle Lücke mit dem Zeichen „@“ definiert werden:

```
[ 2 3 ] '[ 1 @ 4 ]
```

ergibt eine Quotation, die programmatisch äquivalent ist zu:

```
[ 1 2 3 4 ]
```

Beispiel zu Fried-Quotations und Macros

In der Dokumentation von Factor wird das Beispiel eines Macros gegeben, welches eine Quotation aufruft und alle Werte sichert, die diese Quotation konsumiert.

```
USING: fry generalizations ;
MACRO: preserving ( quot -- )
  [ infer in>> length ] keep '[ _ ndup @ ] ;
```

Mit diesem Macro lässt sich eine Variante von „if“ implementieren, die ein Prädikat statt einem boolean erwartet:

```
: ifte ( pred true false -- ) [ preserving ] 2dip if ; inline
```