Ausarbeitung des „Interpreter“ Referats

Gliederung

1. Programmiersprache
   1.2. Syntax
      1.2.1. Konkrete Syntax
      1.2.2. Abstrakter Syntax Baum (Abstrakte Syntax)

2. Parser
   2.1. Syntaktische Struktur einer Sprache
   2.2. Parser Generator

3. Interpreter

4. Erweiterungen unserer Programmiersprache
   4.1. Bedingte Auswertung
   4.2. Lokale Bindungen
   4.3. Prozeduren und Closures

1. Programmiersprache

Ein Interpreter beschäftigt sich damit ein Programm auszuführen. Aber zuerst müssen wir das Programm schreiben und dafür benötigen wir eine Programmiersprache. Es gibt viele Aspekte unter denen wir eine Programmiersprache analysieren können, aber wir werden uns nur mit der Syntax einer Sprache beschäftigen.

1.2. Syntax:

System von Regeln, das die Form einer (Programmier-) Sprache beschreibt. Die Syntax ist nicht die Bedeutung der Sprache das ist die Semantik.

Man unterscheidet zwischen konkrete und abstrakte Syntax einer Programmiersprache.

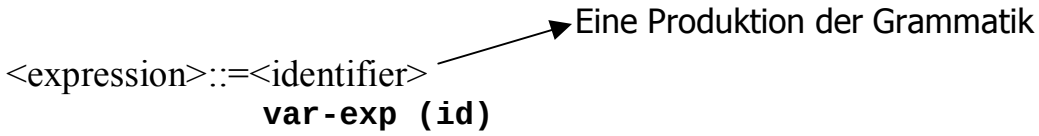
1.2.1. Konkrete Syntax:

Ist die Syntax der Sprache, so wie sie der Programmierer in den Computer tippt. (Der Programmtext, hello.scm oder hello.java). Die BNF Definition, für *Kontext-freie-Grammatiken*, beschreibt wie bestimmte Datentypen repräsentiert werden (z.B. Addition mit „+“, Multiplikation mit „*“ u.s.w.) und dafür benutzt sie die bestimmten Zeichenketten und Werte, erzeugt bei der Grammatik. = externe Repräsentation. Die konkrete Syntax wurde erfunden nur damit es für die Menschen leichter ist, ihre

Programme schreiben zu können und die abstrakte Syntax(= interne Repräsentation) zu verstehen.

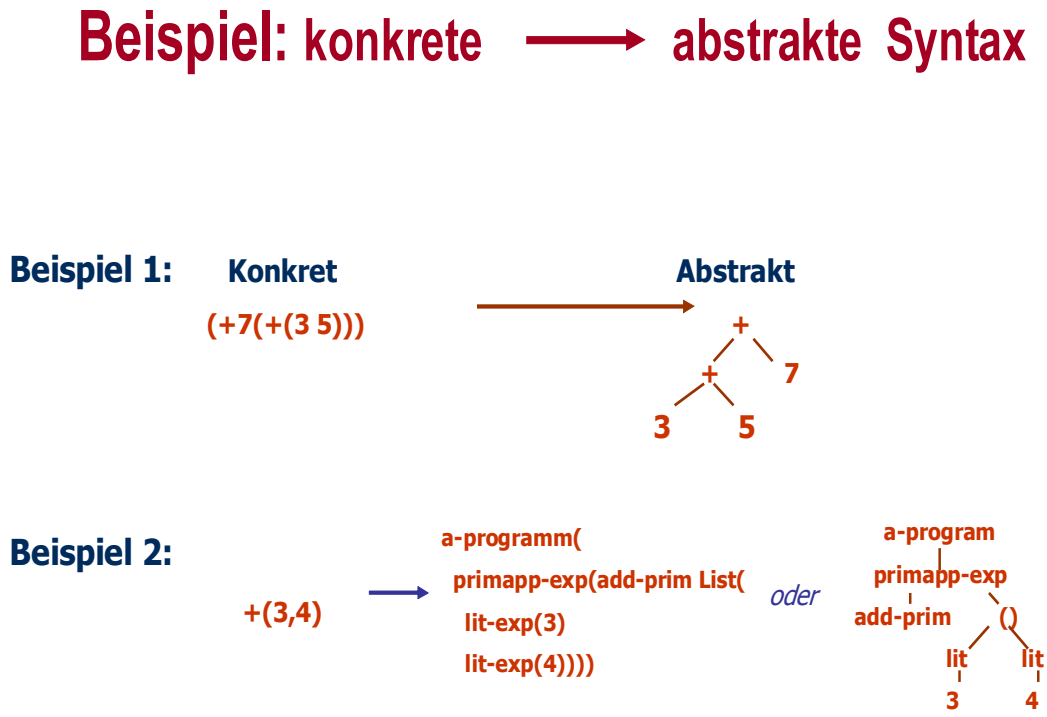
1.2.2. Abstrakte Syntax (Abstrakter Syntax Baum):

Ist die konkrete Syntax als möglichst knapper Baum dargestellt. Terminals, wie Klammer, brauchen nicht gespeichert werden, weil die keine Information beinhalten. Um abstrakte Syntax für bestimmte konkrete Syntax bauen zu können, müssen wir jede Produktion und deren nonterminals der konkreten Syntax benennen.



Wir haben diese Produktion der konkreten Syntax mit `var-exp` benannt und ihre nonterminlas mit `id`.

Die folgenden Beispiele zeigen wie die konkrete Syntax als abstrakt aussieht. Man sieht ganz genau wie die unnötigen Terminals von dem Programm herausgenommen worden sind.
Im Beispiel 2 ist es nicht so ganz klar. Stellt euch vor den ganzen Text weg, das was übrig bleibt sind die Elemente, genommen von der konkreten Syntax, nämlich: add-prim (steht für das „+“ Zeichen), 3 und 4. Der Text ist von dem Interpreter (sein eigenen Programmtext).



2. Parser

Wenn wir ein Programm schreiben, schreiben wir es in konkrete Syntax. Aber wie wir schon geklärt haben ist die konkrete Syntax nur für uns (die Menschen) gedacht. Damit der Interpreter uns verstehen kann, was wir programmiert haben muss unser Programm in abstrakte Syntax umgewandelt werden, die der Interpreter schon versteht. Diese Umwandlung macht der Parser:

Dem Parser werden Tokens(=Reihenfolge von Zeichen(Worte, Zahlen, Sonderzeichen…)) als Input gegeben und er organisiert sie in hierarchisch-syntaktische Struktur(Expressions, Statements, Blöcke…). Er formt die syntaktische Struktur einer Sprache.

2.2. Es besteht die Möglichkeit den Parser selbst zu programmieren, aber heut zu Tage macht man das nicht. Dafür gibt es ein anderes Programm der Parser-Generator: hat als Input lexikalische Spezifikation und Grammatik, und produziert als Output scanner und parser für diese.

3. Interpreter

Nachdem der Parser unser Programm in abstrakte Syntax umgewandelt hat übergibt er sie dem Interpreter. Der Interpreter liest sie und führt sie zeilenweise aus. Im Gegensatz zu einem Compiler wird kein Maschinencode erzeugt.

Ein Interpreter kann nur mit abstrakter Syntax operieren!!!

Beim programmieren eines Interpreters ist sehr wichtig zu unterscheiden zwischen definierte und definierende Sprache!

Definierte Sprache (*oder* source Sprache):

Die Sprache, die wir mit dem Interpreter definieren.

Definierende Sprache (*oder* host Sprache):

Die Sprache, in der wir den Interpreter schreiben. In unserem Fall wird es Scheme sein.

Was auch sehr wichtig ist die definierende Sprache sehr gut zu kennen und mit ihr gut umgehen zu können. Warum, werden wir später erfahren.

Wie bauen wir unser Interpreter?

Immer wenn wir was Neues in unserer Programmiersprache definieren wollen, müssen wir uns an einem bestimmten Plan (Definitionsplan) halten:

1. Definition der konkreten Syntax: wie wollen wir, dass die Sachen aussehen;
2. Definition der abstrakten Syntax;
3. Implementieren in dem Programm.
4.

Fangen wir mit den Schritten 1 und 2:

So sehen unsere konkrete und abstrakte Syntax aus:

<programm> ::= <expression>

```
a-program (exp)
```

<expression> ::= <nummer>

```
lit-exp (datum)
```

::= <identifier>
`var-exp (id)`

::= <primitive> ({<expression>}*′)
`primapp-exp (prim rands)`

<primitive> ::= + | - | * | add1 | sub1|

Wir haben definiert, dass ein Programm nicht anderes außer ein Expression ist und ein Expression kann die folgenden Sachen sein: Zahl (`lit-exp`), Variable (`var-exp`) oder eine primitive Anwendung (`primapp-exp`). Für primitive Anwendungen definieren wir die Addition, Subtraktion, Multiplikation, addiere 1 und subtrahiere 1.
In unserer Sprache haben wir auch die Variablen definiert und damit wir richtig mit denen umgehen können, müssen wir noch ein Begriff klären: die Umgebung einer Funktion.
Jede Funktion, die variablen beinhaltet, hat ihre eigene Umgebung für sie. In der Umgebung werden die Werte und Typen der Variablen gespeichert, damit die Funktion ausführbar wird.

```
(define f (lambda (x)
   (let (y 10)
      (in (+ (x y))))))
(f(5))
```

Umgebung →

x	5
y	10

Jetzt können wir schon mit der Implementierung anfangen. Und so sieht der Interpreter für unsere einfache Programmiersprache:

```
(define eval-program
   (lambda (pgm)
      (cases program pgm
         (a-program (body)
            (eval-expression body (init-
env))))))

(define eval-expression
   (lambda (exp env)
      (cases expression exp
         (lit-exp (datum) datum)
         (var-exp (id) (apply-env env id))
         (primapp-exp (prim rands)
            (let ((args (eval-rands rands
env)))
               (apply-primitive prim args)))
         )))
```

Die Funktion `init-env`, deren Implementierung hier nicht gezeigt ist, dient dazu eine leere Umgebung zu erzeugen für das Programm mit Körper `body`. Danach wird `eval-expression` so oft aufgerufen bis wir alle einzelnen Elemente von `body` haben und eine eventuelle Anwendung auf denen ausgeführt haben.

4. Bis jetzt ist unsere Sprache ganz einfach und das wollen wir mit ein paar Erweiterungen verändern.

4.1. Bedingte Auswertung – if Anweisung

Wie wir alle wissen ist die if Anweisung eine Fallunterscheidung und damit wir die Auswertung durchführen zu können brauchen wir die boolesche Werte, um zu entscheiden welchen Weg wir nehmen sollen. Um das Hinzufügen von Bohlesche-Werte zu unserer Sprache zu vermeiden, werden wir sie als Integer-Werte kodiert. Dafür implementieren wir die folgende Hilfsfunktion:

```
(define true-value?
   (lambda (x)
      (not (zero? x))))
```

Mit dieser Funktion sagen wir, dass die Null als falsch interpretiert werden soll und alle andere Werte als richtig. Hier sind 2 Beispiele wie unser if aussehen wird und ihre Ausgabewerte:

```
--> if 1 then 2 else 3
2
--> if -(3, + (1, 2)) then 2 else 3
3
```

Nachdem wir das Problem mit den booleschen Werten schon beseitigt haben, bleibt uns nur übrig die 3 Schritte des bestimmten Definitionsplans zu folgen um die if Anweisung zu unserer Sprache hinzufügen.

Schritt 1. und 2.: Definition der konkreten und abstrakten Syntax:

<expression> ::= if <expression> then <expression> else <expression>

```
if-exp (test-exp true-exp false-exp)
```

Die Funktion if, in unserer Sprache, hat 3 formale Parameter. Nach der Bewertung des `test-exps` wird entschieden ob der `true-exp` oder der `false-exp` ausgewertet werden soll.

Schritt 3.: Implementierung in `eval-expression`:

```
(if-exp (test-exp true-exp false-exp)
   (if (true-value? (eval-expression test-exp env))
      (eval-expression true-exp env)
      (eval-expression false-exp env)))
```

Dieser Code beinhaltet die if Form von der definierenden Sprache um die if Form der definierten Sprache zu implementieren. Deshalb ist es so wichtig, dass man mit der definierenden Sprache gut umgehen kann.

4.2. Lokale Bindungen – let Anweisung

Beispiel:

```
let x = 5
    y = 6
in +(x, y)
```

Das ist ein Beispiel wie wir wollen, dass unsere let Anweisung aussieht.

Bei lokalen Bindungen innere Deklarationen beschatten, oder schaffen Löcher in dem Körper von, äußere Deklarationen. D.h. ein variablen Objekt wird immer mit der meist näheren lexikalischen Bindung gebunden. Dieser Gesetzt gilt bei allen Programmiersprachen mit Blockstruktur.

Hier ist ein Beispiel bei dem die lokalen Bindungen gut zu sehen sind:

```
let x = 1
 in let x = +(x, 2)
    in add1 (x)
> 4
```

Wie gewohnt folgen wir die Schritte in dem Definitionsplan:

Schritt 1. und 2.:

<expression> ::= let {<identifier> = <expression>}* in <expression>

```
let-exp (ids rands body)
```

Unser let ist so gebaut, dass er die Werte des Operanden den Identifikatoren, im Ausdruck `body`, übergibt.

Schritt 3.:

```
(let-exp (ids rands body)
   (let ((args (eval-rands rands env)))
         (eval-expression body (extend-env
          ids args env)))
```

Dieser Code beinhaltet die let Form der definierenden Sprache um die let Form der definierten Sprache zu implementieren.

4.3. Prozeduren und Closures

Wenn eine Prozedur aufgerufen wird, wird ihr Körper in einer Umgebung ausgeführt, die die formalen Parameter der Funktion mit den Argumenten der Applikation bindet.

Beispiel:

```
let x = 5
in let f = proc (y, z) +(y, -(z, x))
       x = 28
in (f 2 x)
```

Die Funktion f wird in der Umgebung ausgeführt, die y mit 2, z mit 28 und x mit 5 bindet. z wird mit 28 gebunden, weil x = 28 die meist nähere lexikalische Bindung von x, zu `in (f 2 x)`, ist.

Wir wissen was Prozeduren sind, aber was sind Closures?

Definition:

Um eine Prozedur ihre Bindungen, die ihre freie Variablen am Prozedur-Erzeugung hatten, behalten zu können, braucht sie ein *geschlossenes* Packet, unabhängig von der Umgebung, in der sie ausgeführt wird! So ein Packet nennt man **Closure**.

Eine Closure muss den Körper, die Namen der formalen Parameter und die Bindungen der freien Variablen der

Prozedur beinhalten. Es ist günstiger und praktischer die ganze Deklarationsumgebung in der Closure zu speichern und nicht nur die Bindungen der freien Variablen. Wenn wir nur die Bindungen in der Closure speichern würden und die Prozedur in einer fremden Umgebung ausführen, könnte sie mit fremdem Körper ausgeführt werden.
Man sagt, dass eine Prozedur *geschlossen über* oder *geschlossen in* ihrer Deklarationsumgebung ist.
Hier sind 2 Beispiele wie eine Closure gebaut wird und wie sie aussieht:

Beispiel 1:

```
(define f
   (lambda (x)
      (let (y 10)
         (lambda (z) (+ x y z)))))
(define g
   (f 5))
```

Freie Variablen

Gebundene Variable

Beispiel 2:

```
(define g (33))
```

Closure →

x	5
y	10
z	33

, z, (+ x y z)

env	**ids**	**body**
Bindungen der freien Variablen	Namen der formalen Parameter	Körper der Prozedur

Bis jetzt kann unsere Sprache nur Integer-Werte zurückgeben. Nachdem wir auch die Prozeduren definiert haben wollen wir, dass sie auch erste-Klasse Werte werden. Dafür müssen wir, aber auch neue Rückgabewerte zu der sprache definieren: **ProcVal**.

Definition:

ProcVal ist die Menge von Werten, die die Prozeduren repräsentieren. Das Interface des Datentyps ProcVal besteht aus:

1. Closure – der beschreibt wie die Prozedur Werte gebaut werden sollen.
2. apply-procval – Funktion, die beschreibt wie die Prozedur Werte verwendet werden sollen.

Implementationene von `procval` und `apply-procval`:

```
(dafine-datatype procval procval?
   (closure
      (ids (list-of symbol?))
      (body expression?)
      (env environment?)))
```

ProcVal ist eine Menge von Werten und nicht nur ein Wert, weil der Wert einer Prozedur aus 3 verschiedene Elemente besteht: ids, body, env.

```
(define apply-procval
   (lambda (proc args)
      (cases procval proc
```

```
        (closure (ids body env)
           (eval-expression body (extend-
            env ids args env))))))
```

Eine Closure wird in procval erzeugt und in apply-procval ausgeführt. Eine Prozedur wird in eval-expression ausgewertet:

```
(...
   (app-exp (rator rands)
      (let ((proc (eval-expression rator
env))
            (args (eval-rands rands
env)))
         (if (procval? Proc)
            (apply-procval proc args)
            (eopl:error eval-expression
              „Attempt to apply non-
procedure ~s“ proc))))
...)
```

Nachdem wir schon alle nötige Teile für die Prozedurendefinition definiert und implementiert haben, müssen wir die allgemeine konkrete und abstrakte Syntax definieren:

<expression> ::= proc ({<identifier>}*´) <expression>

```
proc-exp (ids body)
```

::= (<expression> {<expression>}*)
app-exp (rator rands)

Schritt 3.: Implementieren in eval-expression:

```
(proc-exp (ids body)
   (closure ids body env))
```

Jetzt haben wir die volle Implementierung von den Prozeduren und deren Werte. Das folgende Beispiel zeigt die Anwendung aller Codes.

```
(eval-expression <<let x = 5
                   in let x = 38
                      f = proc (y, z) *(y,
                          +(x, z))
                      g = proc (u) +(u, x)
                   in (f (g 3) 17)>>
                  env0)

(eval-expression <<let x = 38
                      f = proc (y, z) *(y,
                          +(x, z))
                      g = proc (u) +(u, x)
                   in (f (g 3) 17)>>
                  env1)
    wo env1 = [x = 5]env0

(eval-expression <<(f (g 3) 17)>> env2)
```

```
wo env2 =
     [x = 38,
      f = (closure (y z) <<*(y, +(x,
           z))>> env1);
      g = (closure (u) <<+(u, x)>> env1)
      ] env1
```