Interpreter - Gliederung

Programmiersprache

- Syntax
  - Konkrete Syntax
  - Abstrakter Syntax Baum (Abstrakte Syntax)

Parser

- Syntaktische Struktur einer Sprache
- Parser Generator

Interpreter

Erweiterungen unserer Programmiersprache

- Bedingte Auswertung
- Lokale Bindungen
- Prozeduren
- Closures

Programmiersprache

Syntax

- Definition

  System von Regeln, das die Form einer (Programmier-)Sprache beschreibt.

- Konkrete Syntax

  Die BNF Definition (Backus-Naur Form), für Kontext-freie-Grammatiken, beschreibt eine bestimmte Repräsentation von induktiven Datentypen: sie benutzt die bestimmten Zeichenketten und Werte, erzeugt bei der Grammatik.

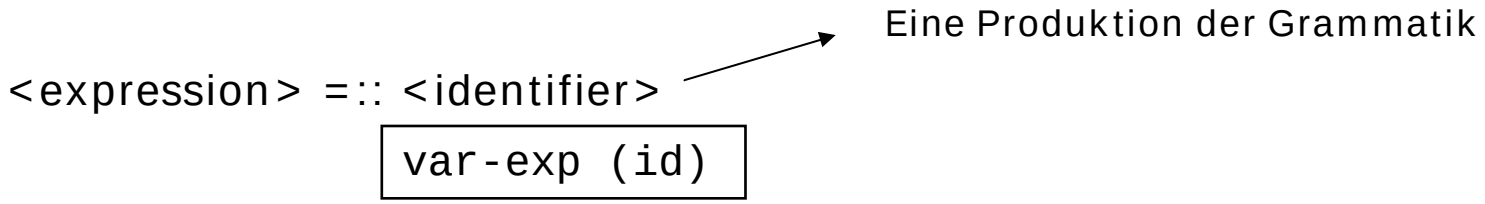
  Ist die Syntax der Programmiersprache, so wie sie der Programmierer in den Computer tippt. (Der Programm Text, hello.scm oder hello.java)

- Abstrakter Syntax Baum (Abstrakte Syntax)

  Um solche Daten bearbeiten zu können, brauche wir sie in einer internen Repräsentation zu umwandeln = abstrakte Syntax.

  Ist die konkrete Syntax als möglichst knapper Baum dargestellt. Terminale, wie Klammer brauchen nicht gespeichert werden, weil die keine Information beinhalten.

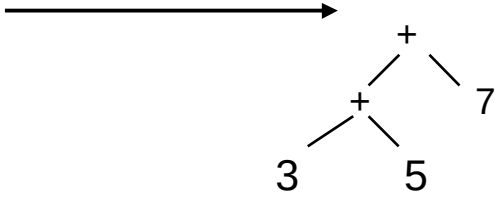
  Um abstrakte Syntax für bestimmte konkrete Syntax bauen zu können, müssen wir jede Produktion und deren nonterminals der konkreten Syntax benennen.



Beispiel: konkrete ⟶ abstrakte Syntax

Beispiel 1: Konkret

(+7(+(3 5)))

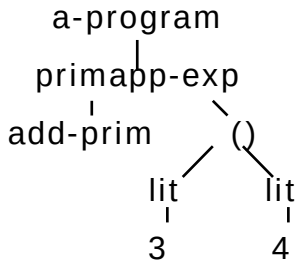


Beispiel 2:

+(3,4)


```
a-programm(
  primapp-exp(add-prim List(
    lit-exp(3)
    lit-exp(4))))
```

oder



Parser

Definition:

Der Parser organisiert die Reihenfolge von Tokens in hierarchisch - syntaktische Struktur (Expression, Statement, Blöcke). Er formt die syntaktische Struktur einer Sprache.

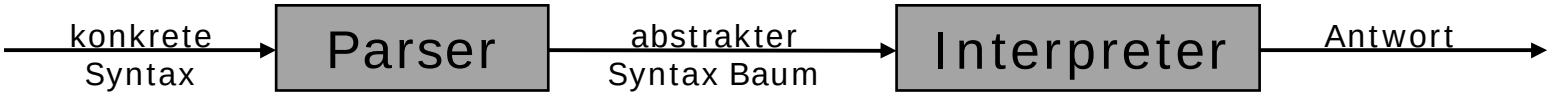
Tokens = Reihenfolge von Zeichen (Worte, Nummern, Sonderzeichen, Kommentare);

Parser-Generator = Programm, dass als Input lexikalische Spezifikation und Grammatik hat, und als Output scanner und parser für diese produziert.

Interpreter (1)

Definition:

- Ein Interpreter parst (übersetzt) Quellcode und führt ihn umgehend aus.
- Im Gegensatz zu einem Compiler erzeugt er keinen speicherbaren Maschinencode.



Interpreter können nur mit abstrakter Syntax operieren!

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 Scheme.

Interpreter (2)

Wie bauen wir unser Interpreter?:

1. Definition der konkreten Syntax:

   Unsere Sprache unterstützt Nummern, Variablen und primitive Anwendungen (+, -, *, add1, sub1);

2. Definition der abstrakten Syntax:

```
<programm> ::= <expression>
               a-program (exp)

<expression> ::= <nummer>
                 lit-exp (datum)

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

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

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

Interpreter (3)

Umgebung einer Variable:

Jede Funktion, die Variablen beinhaltet, hat ihre eigene Umgebung für sie. In der Umgebung werden die Werte und die Typen der Variablen gespeichert, damit die Funktion ausgeführt werden kann.

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

Umgebung

x	5
y	10

Im Interpreter wird die Umgebung einer Funktion überall mitgenommen.

Interpreter (4)

3. Implementieren

```
(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)))
         )))
```

Erweiterung unserer Programmiersprache
Bedingte Auswertung – if Anweisung (1)

Um das Hinzufügen von boolean-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))))
```

Beispiele für eine if Anweisung:

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

Bedingte Auswertung – if Anweisung (2)

Erweiterung der Sprache mit Fallunterscheidung – if Anweisung:

1. Definition der konkreten Syntax
2. Definition der abstrakten Syntax

```
<expression> ::= if <expression> then <expression> else <expression>
                 if-exp (test-exp true-exp false-exp)
```

3. Implementieren 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.

Erweiterung unserer Programmiersprache
Lokale Bindungen – let Anweisung (1)

Beispiel:

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

Lokale 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.

Beispiel:

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

Lokale Bindungen – let Anweisung (2)

Erweiterung mit lokalen Zuweisungen – let Anweisung:

1. Definition der konkreten Syntax
2. Definition der abstrakten Syntax

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

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

3. Implementieren in `eval-expression`:

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

Erweiterungen unserer Sprache
Prozeduren und Closures (1)

Wenn eine Prozedur aufgerufen wird, wird ihr Körper in einer Umgebung ausgeführt, die die formale 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.

Prozeduren und Closures (2)

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.

Ein 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 dem Closure zu speichern und nicht nur die Bindungen der freien Variablen.

Man sagt, dass eine Prozedur geschlossen über oder geschlossen in ihrer Deklarationsumgebung ist.

Prozeduren und Closures (3)

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

Prozeduren und Closures (4)

Wir wollen, dass Prozeduren auch erste-Klasse Werte in unserer Sprache werden. Dafür müssen wir, aber auch neue Rückgabewerte zu der Sprache definieren: ProcVal.

Definition:

ProcVal ist die Menge von Werten, die die Prozeduren repräsentieren.
Der 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.

Implemetationen von `procval` und `apply-procval`:

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

Prozeduren und Closures (5)

```
(define apply-procval
   (lambda (proc args)
      (cases procval proc
         (closure (ids body env)
            (eval-expression body (extend-env ids args env))))))
```

Ein 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))))
...)
```

Prozeduren und Closures (6)

Erweiterung mit selbst definierten Funktionen:

1. Definition der konkreten Syntax
2. Definition der abstrakten Syntax

```
<expression> ::= proc ({<identifier>}*´) <expression>
                 proc-exp (ids body)

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

3. Implementieren in `eval-expression`:

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

Prozeduren und Closures (7)

Beispiel 3:

```
(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
```

Prozeduren und Closures (8)

```
(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
```