

Programmbestandteile

- Literale
- Variablen
- Schlüsselwörter
- Zusammengesetzte Formen

Literale

Literal: Ein Literal ist ein vordefinierter Name, der für einen festen Wert steht.

Beispiele:

- Ziffernfolgen sind Namen für Ganzzahlen
- Ziffernfolge `.` Ziffernfolge sind Namen für Fließkommazahlen
- `true` und `false` sind in OCaml und Java die Namen der booleschen Werte

Schlüsselwörter

Schlüsselwort: Ein Schlüsselwort ist ein vordefinierter Name, der dazu dient, zusammengesetzte Formen zu definieren.

- `if`
- `else`
- `while`
- `fun`
- `class`

Variable

Variable: Eine Variable ist ein Name, der an einen Wert gebunden ist.

Variablen kommen in Programmen als *gebundene Variablen* und als *Variablenreferenzen* vor

Zusammengesetzte Form

Zusammengesetzte Form: Grundlegende Bestandteile einer Programmiersprache.

Die Sprachdefinition legt die Bedeutung der zusammengesetzten Formen fest.

- `for`-Schleifen
- Funktionsdefinitionen
- Funktionsanwendung
- Variablendeklaration in C
- Sequenzen (;)

Konkrete Syntax

Konkrete Syntax: Konkrete Syntax ist die Repräsentation des Programmes als Zeichenfolge gemäß der Sprachdefinition.

Konkrete Syntax für arithmetische Ausdrücke

```
<exp> ::= <term>
        | <exp> + <term>
        | <exp> - <term>

<term> ::= <factor>
        | <term> * <factor>
        | <term> / <factor>

<factor> ::= ( <exp> )
          | <integer>
```

Konkrete Syntax für Mini

```
<exp> ::= <var>
        | <int>
        | <exp> ( <exps> )
        | <exp> + <exp>
        | fun ( <ids> ) -> <exp>
        | ( <exp> )

<exps> ::= <exp>
         | <exp> , <exps>

<ids>  ::= <var>
         | <var> , <ids>
```

Beispiele für Mini

```
fun (x) -> x
f(1,g(2,y))
(fun (f) -> f(2))
  (fun (y) -> y)
```

Weitere Features

Mini kann noch mehr:

- if
- let
- letrec
- -, *, /
- Zuweisung
- Zeiger

Abstrakte Syntax

Abstrakte Syntax: Abstrakte Syntax ist eine Repräsentation des Programmcodes, die auf unwichtige Details aus der konkreten Syntax verzichtet.

Unwichtig: Klammern, Infix-Operatoren, Whitespace

Vorteile:

- Abstrakt ;-)
- Vereinfacht Manipulation des Programms im Compiler

Annäherung an Abstrakte Syntax

Analogie: Man denkt auch nicht über Ziffernfolgen, sondern über Zahlen nach.

Ziel also: Nur Aufbau festhalten,

Verschiedene Repräsentationen sollen möglich sein

Aber jede Repräsentation muss Aufbau erhalten

Abstrakte Grammatik

Notation für Abstrakte Syntax: Abstrakte Grammatik

- Wie bei kontextfreier Grammatik: Produktionen für Nonterminale
- Einige Nonterminale werden als gegeben angenommen (z.B. Konstanten, Variablennamen)
- Die rechten Seiten der Produktionen eines jeden Nonterminals haben stets verschiedene Muster von Terminalen.

Beispiel Abstrakte Grammatik

Abstrakte Grammatik für arithmetische Ausdrücke:

```
<exp> ::= <exp> + <exp>
        | <exp> - <exp>
        | <exp> * <exp>
        | <exp> / <exp>
        | <integer>
```

Beobachtungen:

- Keine geklammerten Ausdrücke
- Keine Präzedenzen

Abstrakte Grammatik für Mini

```
<exp> ::= <int>
        | <var>
        | <id>* -> <exp>
        | apply <exp> <exp>*
```

Beobachtungen:

- Keine geklammerten Ausdrücke
- Keine Infix-Operatorkaufrufe

Andere Hälfte der Abstraktheit

Abstrakte Syntax ist auch repräsentationsunabhängig

Jede Familie von Mengen ist geeignet, wenn sie

- für jedes Nichtterminal der Abstrakten Grammatik eine *Trägermenge*
- und für jede Produktion der abstrakten Grammatik einen *Konstruktor* auf den Trägermengen

definiert.

Anforderungen

(Ziel war: Bei jeder Repräsentation muss Aufbau erhalten bleiben)

- Jeder Konstruktor muss injektiv sein.
- Je zwei Constructoren in den selben Träger müssen unterschiedliche Bildbereiche haben.
- Jedes Element jeder Trägermenge muss durch eine endliche Anzahl von Konstruktoranwendungen erzeugt werden können.

Beispiele für Trägermengen

- Strings mit voller Klammerung
- Bäume mit Markierungen an den Knoten
- Algebraische Datentypen

Stringrepräsentation

Abstrakte Syntax für arithmetische Ausdrücke mit vollgeklammerten Strings als Trägermenge.

Konstruktor sind:

`c0 () = "0"`

...

`c+ (x,y) = "(" ^ x ^ ")" ^ "+" ^ "(" ^ y ^ ")"`

`c- (x,y) = "(" ^ x ^ ")" ^ "-" ^ "(" ^ y ^ ")"`

...

Abstrakte Syntax repräsentiert in OCaml

```
type exp =  
  Add of exp * exp  
  | Sub of exp * exp  
  | Mult of exp * exp  
  | Div of exp * exp  
  | Int of int
```

Abstrakte Syntax von Mini

Repräsentiert in OCaml:

```
type expr =
  Const of int
| Var of string
| Fun of string list * expr
| App of expr * (expr list)
```

Erzeugen von Abstrakter Syntax

In einem Compiler:

- Zuerst: Lexer wandelt Zeichen in Wörter (*Tokens*) um
- Dann leitet der Parser Syntaxbaum gemäß konkreter Syntax her
- und liefert Abstrakte Syntax

Lexen für Mini

Token-Spezifikation für `camllex`:

```
rule initial = parse
  whites      { initial lexbuf }
| "fun"      { FUN      }
| ","        { COMMA    }
| "->"      { ARROW    }
| "("        { LPAR     }
| ")"        { RPAR     }
| ['a'-'z'](['a'-'z''A'-'Z''0'-'9''_']* )
  { ID(Lexing.lexeme lexbuf) }
| ['0'-'9']+ { INT(Lexing.lexeme lexbuf) }
```

Erzeugen der Abstrakten Syntax

Beim Parsen mit Programm `ocamlyacc`

```
exp: ID { Var $1 }
exp: INT { Const (int_of_string $1) }
exp: LPAR exp RPAR { $2 }

exp: FUN LPAR ids RPAR ARROW exp { Fun($3,$6) }
ids: ID { [ $1 ] }
ids: ID COMMA ids { $1 :: $3 }

exp: exp PLUS exp { App(Var "+", [ $1; $3 ]) }
exp: exp LPAR exps RPAR { App($1,$3) }

exps: exp COMMA exps { $1 :: $3 }
exps: exp { [ $1 ] }
```

Bindung

Betrachte folgenden Ausdruck:

```
fun (x) -> (fun (x) -> x + 2)
```

Welcher Binder gehört zur Referenz der Variable **x**?

Lexikalische Bindung

Lexikalische Bindung sagt: Der Binder für denselben Namen, der im Programmtext als nächster kommt, wenn man von innen nach außen sucht

"von innen nach außen suchen": Gemäß des Programmaufbaus durch zusammengesetzte Formen

In abstrakter Syntax offensichtlich

Lexikalische Bindung

Auch "statische Bindung" genannt

Großer Vorteil: Inspektion des Programmtextes reicht aus, um Binder zu bestimmen

Von den meisten Programmiersprachen verwendet

Scope

Scope: Sichtbarkeitsbereich eines Binders, also der Programmteil in dem die Bindung aktiv ist

Andere Sichtweise auf lexikalische Bindung:

- Von außen nach innen
- Variablenreferenzen im Scope eines Binders sind vom Binder gebunden.

Shadowing

Shadowing: Ein Binder bindet einen Namen, der momentan sichtbar ist.

Der alte Binder wird also verdeckt

Gehört zu lexikalischer Bindung

Implementierung statischer Bindung

Grund: Implementierung hilft beim Verständnis

Mittels eines Interpreters:

- Umgebung bildet Namen auf Werte ab
- Neuer Binder: Abbildung erweitern
- Scope verlassen: Alte Umgebung wiederherstellen

Bindung in Mini

Ebenfalls lexikalisch

Binder sind `fun` und `let`

`fun` bindet Parameter im Rumpf an die Werte der Argumente

`let` bindet Variable im Rumpf an Wert der rechten Seite

Auswertung

Auswertung: Ausdruck in Wert umformen

Was ist zu tun?

- Konstanten: Nix, sind schon Werte
- Funktionsaufrufe: Argumente auswerten, Parameter an Ergebnisse binden, Rumpf auswerten.
- Funktionen: Eigentlich nix, aber...

Operationen auf Umgebungen

Umgebung ist Liste von Assoziationslisten, die Bezeichner auf Werte abbilden

Für jeden Binder neue Assoziationsliste ("Frame")

```
let initial_env = []

let extend_env env names vals =
  List.combine names vals :: env

let rec lookup_env env name =
  match env with
  [] -> failwith "Unbound identifier"
  | frame::frames ->
    try
      List.assoc name frame
    with Not_found -> lookup_env frames name
```

Werte für Mini

Werte in Mini sind Zahlen und Closures

```
type value =
  Int of int
  | Clos of string list * expr * env

and env = ((string * value) list) list
```

Closures

```
let x = 23 in
  fun y -> x + y
```

Für lexikalische Bindung muss die Auswertung einer Funktion die momentane Umgebung speichern

⇒ Closure

Interpreter für Mini

```
let rec eval e env =
  match e with
  Const i -> Int i
  | Var name -> lookup_env env name
  | Fun (paras, body) -> Clos (paras, body, env)
  | App (f, args) ->
    let fVal = eval f env in
    let argsVals =
      List.map (fun e -> eval e env) args in
    match fVal with
    Clos (paras, body, clos_env) ->
      eval body (extend_env clos_env
        paras
        argsVals)
    | _ -> failwith "Operator is not a function"
```

De-Bruijn-Indizes

Suche in der Umgebung erfordert Namensvergleich: teuer

Beobachtung: Wegen statischer Bindung lässt sich Position des Bezeichners in der Umgebung am Programmtext ablesen

Idee: Position vor der Interpretation bestimmen und anstelle der Namen in Variablen speichern

⇒ De-Bruijn-Indizes

Didaktischer Hintergrund

Einstimmung auf Java-Compiler

- Umwandlung von Namen zu Indizes ist einfache Form der Compilierung
- Übersetzung von abstrakter Syntax in Zwischenrepräsentation
- Einführung einer Compile-Zeit-Umgebung

Außerdem: zweite Sicht auf statische Bindung

Position

Position ist Integer-Paar: Abstand zum Binder und Index im Binder

Beispiel:

```
let x = 23 in
  fun y z ->
    let w = z + 666 in
      x + y + z
```

Position von `z` in `z + 666` ist $(0,1)$

Position von `x` in `x + y + z` ist $(2,0)$

Position von `y` in `x + y + z` ist $(1,0)$

Position von `z` in `x + y + z` ist $(1,1)$

Zwischenrepräsentation mit de-Bruijn-Indizes

Funktionsrepräsentation braucht Parameter-Namen nicht mehr.

```
type pos = int * int
```

```
type dbexp =
  Const of int
  | Var of pos
  | Fun of dbexp
  | App of dbexp * (dbexp list)
```

Compile-Zeit-Umgebung

Umgebung, die während der Übersetzung Informationen über Variablen speichert

Ist gemäß lexikalischer Bindung zu verwalten

Hier:

```
type cenv = (string list) list
```

```
lookup_cenv: cenv -> string -> pos liefert Position
```

Implementierung der Compile-Zeit-Umgebung

```
let initial_cenv = []

let extend_cenv cenv vars = vars::cenv

let lookup_cenv cenv name =
  let rec loop cenv depth =
    match cenv with
    | [] -> failwith "Unbound variable"
    | names::cenv_rest ->
      try
        (depth,index name names)
      with Not_found -> loop cenv_rest (depth+1) in
  loop cenv 0
```

Übersetzung

Programm rekursiv ablaufen

- Bei Binder Compile-Zeit-Umgebung erweitern
- Bei Variablenreferenz Position nachfragen

```
let rec compile e cenv =
  match e with
  | Ast.Const i -> Const i
  | Ast.Var name -> Var (lookup_cenv cenv name)
  | Ast.Fun (paras, body) ->
    Fun (compile body (extend_cenv cenv paras))
  | Ast.App (f, args) ->
    App (compile f cenv,
         List.map (fun e -> compile e cenv) args)
```

Werte und (Laufzeit-)Umgebungen

Umgebung: Listen von Arrays

```
type value =
  Int of int
  | Clos of dbexp * env

and env = (value array) list

let initial_env = []

let lookup_env env (depth,pos) =
  (List.nth env depth).(pos)

let extend_env env vals =
  (Array.of_list vals)::env
```

Interpreter mit de-Bruijn-Indizes

Wegen neuem CLOS sind Fun und App leicht verändert

```
let rec eval e env =
  match e with
  | Const i -> Int i
  | Var name -> lookup_env env name
  | Fun body -> CLOS (body, env)
  | App (f, args) ->
    let fVal = eval f env in
    let argsVals =
      List.map (fun e -> eval e env) args in
    (match fVal with
     | CLOS (body, clos_env) ->
       eval body (extend_env clos_env argsVals)
     | _ -> failwith "Operator is not a function")
```

Dynamische Bindung

Alternative Form der Bindung

Zur Referenz einer Variable gehört der Binder, der bei Programmausführung als letzter aktiv war

Binder ist also abhängig von der Ausführung des Programms

Dynamische Bindung

Populär in LISP

Nicht modular, schwer effizient zu implementieren

Anwendung: Implizite Parameter

Beispiel

kein OCaml:

```
letdyn x = 23 in
  letdyn f y = x + y in
    letdyn x = 24 in
      f 1
```

```
letdyn add23 =
  letdyn x = 23 in
    fundyn y -> y + x
```

```
letdyn x = 24 in
  add23 1
```

⇒ 25

Zuweisung

Zuweisung: Eine Zuweisung ersetzt den Wert, an den eine Variable gebunden ist durch einen neuen Wert

Weit verbreitet in imperativen Programmiersprachen

Zuweisung in Mini

Konkrete Syntax:

```
<exp> ::= ... | <var> := <exp>
```

Lexer:

```
| "==" { ASSIGN }
```

Abstrakte Syntax:

```
type exp =  
  ...  
  | Assign of string * expr
```

Parser:

```
exp: ID ASSIGN exp { Assign($1,$3) }
```

Zuweisung im Interpreter

Wert in der Umgebung verändern

Erfordert veränderbare Umgebung

⇒ Nur möglich, wenn Frames durch Arrays repräsentiert sind

Zuweisung und Umgebung

Neue Operation auf Umgebungen: Wert einer Variable ändern

```
let update_env env (depth,pos) new_val =  
  (List.nth env depth).(pos)<-new_val
```

Zuweisung und der Compiler

Compiler muss Variablenname durch Position ersetzen

Anpassung der Zwischenrepräsentation:

```
type dbexp =  
  ...  
  | Assign of pos * dbexp
```

Anpassung des Compilers:

```
let rec compile e cenv =  
  ...  
  | Ast.Assign (n, e) ->  
    Assign (lookup_cenv cenv n, compile e cenv)
```

Zuweisung im Interpreter

Ausdruck auswerten und Umgebung verändern:

```
let rec eval e env =  
  ...  
  | Assign (pos, e) ->  
    let new_val = eval e env in  
    update_env env pos new_val;  
    Int 42
```

Rückgabewert ist beliebig

(Wechselseitig) rekursive Definitionen

Mini kennt `letrec` analog zu `let rec` in Ocaml

- Variablen sind im Rumpf und in den rechten Seiten gebunden
- Ähnlich zu Funktionsdeklarationen in C

letrec in Mini

Konkrete Syntax:

```
<exp> ::= ... | letrec <bind-group> in <exp>  
<bind-group> ::= <bind> | <bind> and <bind-group>
```

Lexer:

```
| "letrec" { LETREC }  
| "and"    { AND }  
| "in"     { IN }
```

letrec in Mini

Abstrakte Syntax:

```
type expr =  
  ...  
  | Letrec of (string * expr) list * expr
```

Parser:

```
exp: LETREC bind_group IN exp { Letrec($2,$4) }  
bind_group: bind { [ $1 ] }  
bind_group: bind AND bind_group { $1 :: $3 }  
bind: ID EQU exp { ($1,$3) }
```

letrec im Compiler

Zwischenrepräsentation:

```
type dbexp =  
  ...  
  | Letrec of dbexp list * dbexp
```

Compiler:

```
let rec compile e cenv =  
  ...  
  | Ast.Letrec (bs, body) ->  
    let (names,rhss) = List.split bs in  
    Übungsaufgabe!
```

letrec im Interpreter

Alle gebundenen Variablen des `letrec` sind in allen rechten Seiten und im Rumpf sichtbar

⇒ Alle in einen Frame in neuer Umgebung

Werte im Frame sind die Werte der rechten Seiten

Zur Auswertung der rechten Seiten ist aber die neue Umgebung nötig

⇒ Zirkelschluss

Beispiel

Auflösung des Zirkels ist im Allgemeinen nicht möglich:

```
letrec x = y and y = x in  
  x
```

Werte der rek. gebundenen Variablen werden jedoch nicht sofort benötigt, wenn rechte Seiten Funktionen sind:

```
letrec f = fun (x) -> ... g(x) ... and  
  g = fun (y) -> ... f(x) in  
  f(...)
```

... denn die Funktionsrumpfe werden noch nicht ausgewertet

Idee

```
letrec x1 = e1
      and x2 = e2
in body
```

wird ausgewertet als

```
let x1 = 0
and x2 = 0
in x1 := e1;
   x2 := e2;
   body
```

letrec im Interpreter

Also:

- Frame erzeugen
- Variablen an Dummywerte binden
- Rechte Seiten mit dieser Umgebung auswerten (hoffen, dass Variablen nicht ausgewertet werden)
- Setzen per Zuweisung die Variablen auf die Werte der rechten Seiten

letrec implementieren

```
let rec eval e env =
  ...
  | Letrec (rhss, body) ->
    let new_env =
      extend_env
        env (List.map (fun _ -> Int 42) rhss) in
    List.fold_left
      (fun pos rhs ->
        update_env new_env
          (0, pos)
          (eval rhs new_env);
        pos + 1)
      0
      rhss;
    eval body new_env
```

Primitiva

Primitivum: Funktion, die nicht mit den Mitteln der Sprache definiert werden kann.

Beispiele:

- Ein/Ausgabe-Funktionen
- Arithmetische Funktionen
- Funktionen auf Datentypen

Werden von der Implementierung bereitgestellt

Primitiva für Mini

Parser erzeugt für Anwendung eines Primitivums in der Abstrakten Syntax `App`-Term:

```
exp: exp PLUS exp { App(Var "+", [ $1; $3 ]) }
```

Müssen Variablen entsprechend binden

Wert muss OCaml-Funktion sein

Neuer Fall in der Menge der Werte:

```
type value =  
  Int of int  
| Clos of string list * expr * env  
| Prim of (value list -> value)
```

Bindung von Primitiva

Globale (initiale) Umgebung bindet Primitiva an `Prim`-Werte

Für Umgebungen mit Namen:

```
let initial_env =  
  [[("+", Prim (fun [(Int i1); (Int i2)] ->  
                  Int (i1 + i2)))]]
```

Für de-Bruijn-Indizes:

- Für Interpreter `Prim`-Werte in der Umgebung
- Für den Compiler muss initiale Compilezeitumgebung Namen der Primitiva enthalten

Auswertung von Primitiva

Erweiterung der Funktionsanwendung

```
let rec eval e env =  
  ...  
| App (f, args) ->  
  let fVal = eval f env in  
  let argsVals =  
    List.map (fun e -> eval e env) args in  
  (match fVal with  
  Clos (paras, body, clos_env) ->  
    eval body  
    (extend_env clos_env paras argsVals)  
| Prim f -> f argsVals  
| _ -> failwith "Operator is not a function")
```

Zeiger und Speicher

- Dynamische Allokation von Speicher ist wichtig
 - C bietet über Zeiger direkten Zugriff auf den Speicher
 - C ist die Zielsprache des Java-Compilers
- ⇒ Müssen verstehen, was Zeiger und Speicher sind

Zeiger-API

Drei Funktionen

- Speicher allozieren
- Wert aus dem Speicher laden
- Wert im Speicher speichern

Neue Wertemenge: Adressen = Indizes in den Speicher

Zeiger in Mini

Syntax

- Allozieren: `alloc(<exp>)`
- Laden: `*<exp>.(<exp>)`
- Speichern: `<exp>.(<exp>)<-<exp>`

Keine Adressarithmetik, dafür Offsets

Beispiel

```
let a = alloc(5)
  in a.(4) <- 3+5; *a.(4)
```

Wertet zu 8 aus

Parser für Adressen

Operationen werden zu Anwendung von Primitiva

```
exp: exp DOT LPAR exp RPAR BACK exp
    { App(Var "store", [ $1; $4; $7 ]) }
```

```
exp: MUL exp DOT LPAR exp RPAR
    { App(Var "load", [ $2; $5 ]) }
```

Adressen

Neuer Wert, Inhalt ist Index:

```
type value =  
  Int of int  
  | Addr of int  
  | Clos of dbexp * env  
  | Prim of (value list -> value)
```

Implementierung

Brauchen nur die drei Primitiva implementieren

... und natürlich den Speicher

Speicher ist Array mit willkürlichem Inhalt:

```
let memory_size = 100
```

```
let memory = Array.make memory_size (Int 42)
```

Einschub: Zuweisung in OCaml

OCaml unterstützt keine Zuweisung an Variablen

Aber `ref` erzeugt eine veränderbare Box

! dereferenziert den Inhalt der Box

:= verändert den Inhalt der Box

```
let x = ref 42
```

!x ergibt 42

x:=23 speichert 23 in der Box

!x ergibt nun 23

Allozieren

Verwalten Index auf nächste freie Speicherstelle:

```
let free_ptr = ref 0
```

```
let alloc size =
```

```
  let free = !free_ptr in
```

```
  if free + size >= memory_size
```

```
  then failwith "Out of memory";
```

```
  free_ptr := free+size;
```

```
  free
```

Primitivum:

```
  Prim (fun [Int size] -> Addr (alloc size))
```

Laden & Speichern

```
let load addr =  
  memory.(addr)
```

```
let store addr new_val =  
  memory.(addr)<-new_val
```

Primitiva:

```
Prim (fun [Addr addr;Int offset] ->  
  load (addr+offset))
```

```
Prim (fun [Addr addr;Int offset;new_val] ->  
  store (addr + offset) new_val;Int 42)
```

Zusammenfassung

- Grundbegriffe: Literal, Schlüsselwort, Variable, zusammengesetzte Form
- Konkrete und abstrakte Syntax
- Lexer und Parser
- Lexikalische Bindung und de-Bruijn-Indizes
- Auswertung und Interpreter
- Compiler und Compile-Zeit-Umgebung
- Zuweisung
- Primitiva
- Speicher

Nachtrag: Arrays in OCaml

Syntax ähnlich zu Records:

[|<expr>;... |] erzeugt ein Array

<expr> . (<expr>) greift auf Array zu

<expr> . (<expr>)<- <expr> speichert Wert in Array

Beispiel

```
let a = [ |1;2+3;4| ] erstellt Array
```

a.(1) ergibt 5

(a.(2)<-42).2 ergibt 42