Konzepte von Programmiersprachen

Kapitel 3: Ausdrücke

Phillip Heidegger

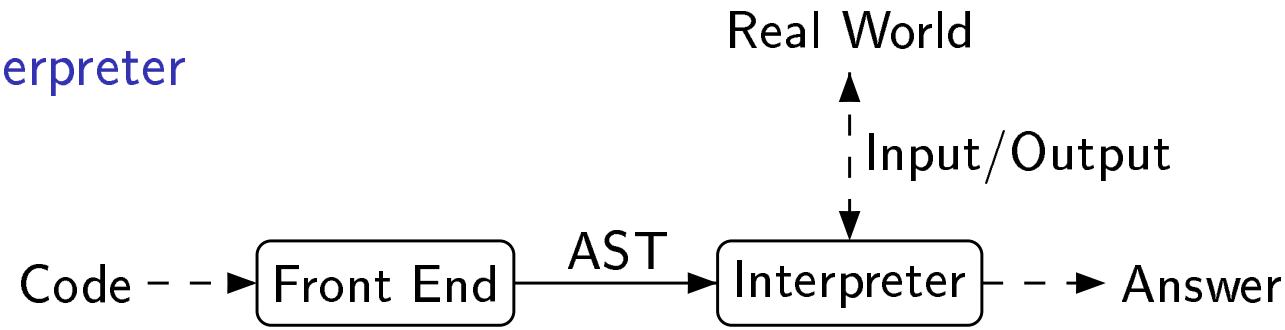
Universität Freiburg, Deutschland

SS 2009

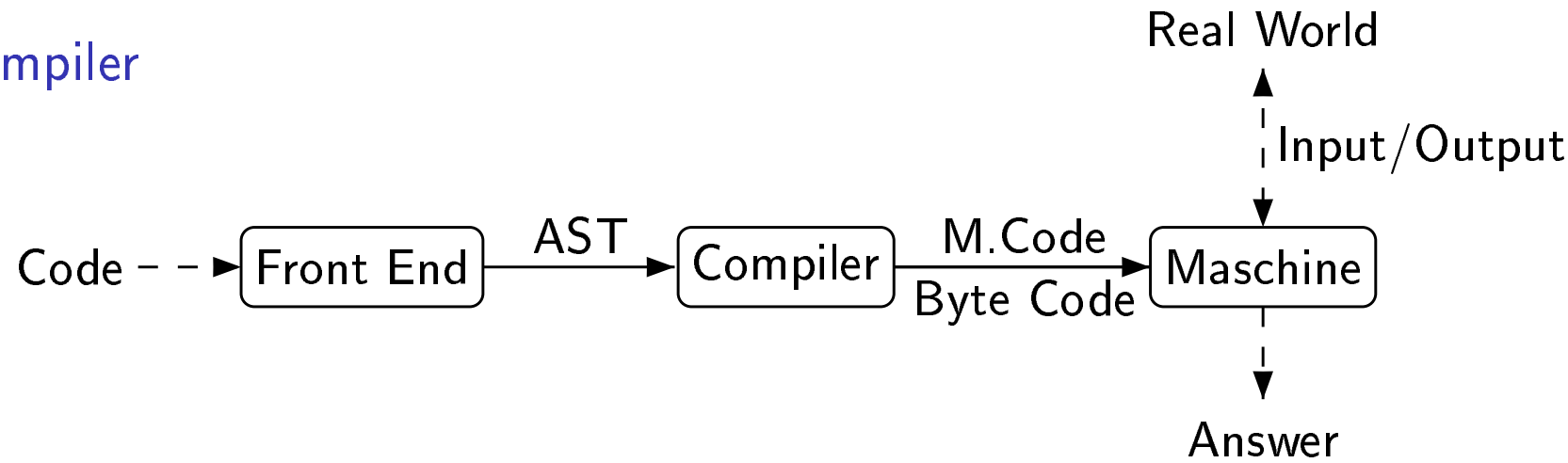
Inhalt

Interpreter – Compiler

Interpreter



Compiler



Syntax – Grammatik

$$
\begin{array}{rcl}
\textit{Programm} & ::= & \textit{Expression} \\
\textit{Expression} & ::= & \textit{Number} \\
& | & -(\textit{Expression}, \textit{Expression}) \\
& | & \texttt{zero?}(\textit{Expression}) \\
& | & \texttt{if } \textit{Expression} \texttt{ then } \textit{Expression} \texttt{ else } \textit{Expression} \\
& | & \textit{Identifier} \\
& | & \texttt{let } \textit{Identifier} = \textit{Expression} \texttt{ in } \textit{Expression} \\
\textit{Identifier} & ::= & x \mid y \mid \ldots
\end{array}
$$

Syntax – Beispiele

Beispiel 1:

```
let x = 5 in
let y = 2 in
  -(x,y)
```

Beispiel 2:

```
let x = 5 in
let y = 2 in
  if zero? (x) then y else -(x,y)
```

Syntax – Beispiele

Beispiel 3:

```
let y = 2 in
let x = 1 in
let y = 0 in
  if zero? (y) then x else 10
```

Syntax – Datentypen für Ausdrücke / AST

```
(define-datatype expression expression?
  (const-exp (num number?))
  (diff-exp (exp1 expression?) (exp2 expression?))
  (zero?-exp (exp1 expression?))
  (if-exp (exp1 expression?)
          (exp2 expression)
          (exp3 expression?))
  (var-exp (var identifier?))
  (let-exp (var identifier?)
           (exp1 expression?)
           (body expression?)))
```

Semantik – Werte

Die Menge der Werte *Val* besteht aus:

- Boolean
- Integer

d.h. $\mathit{Val} = \mathit{Boolean} + \mathit{Integer}$.

Interface für die Werte

`num-val`	:	$\mathsf{Int} \longrightarrow \mathsf{Val}$
`bool-val`	:	$\mathsf{Bool} \longrightarrow \mathsf{Val}$
`val->num`	:	$\mathsf{Val} \longrightarrow \mathsf{Int}$
`val->bool`	:	$\mathsf{Val} \longrightarrow \mathsf{Bool}$

Semantik – Umgebung

Eine Umgebung ρ hat den Typ:

$$\rho : \mathit{Indetifier} \rightarrow \mathit{Val}$$

Hierbei gelte:

- [] steht für die leere Umgebung
- $[x = v]\rho$ steht für die Umgebung, in der x dem Wert *val* zugeordnet ist, d.h. für `(extend-env x v` ρ`)`

Ausdrücke

Interface für Ausdrücke

`const-exp`	:	$Int \longrightarrow Exp$
`zero?-exp`	:	$Exp \longrightarrow Exp$
`if-exp`	:	$Exp \times Exp \times Exp \longrightarrow Exp$
`diff-exp`	:	$Exp \times Exp \longrightarrow Exp$
`var-exp`	:	$Var \longrightarrow Exp$
`let-exp`	:	$Var \times Exp \times Exp \longrightarrow Exp$
`value-of`	:	$Exp \times Env \longrightarrow Val$

Spezifikation von value-of (1/4)

Konstanten

```
(value-of (const-exp n) p) = (num-val n)
```

Variablen

```
(value-of (var-exp x) p) = (apply-env p x)
```

Diff Ausdruck

```
(value-of (diff-exp e1 e2) p)
= (num-val
    (- (val->num (value-of e1 p))
       (val->num (value-of e2 p))))
```

Spezifikation von value-of (1/4) – Beispiel

Abkürzung von `<<-(x,5)>>` für AST, der den Ausdruck darstellt, d.h.

```
<<-(x,5)>> = (exp-diff (var-exp "x")
                       (const-exp 5))
```

Beispiel:

```
  (value-of <<-(x,5)>> [x=3])
= (num-val (- (val-num (value-of <<x>> [x=3]))
              (val-num ((value-of <<5>> [x=3])))))
= (num-val (- (val->num (apply-env [x=3] x))
              (val->num (num-val 5))))
= (num-val (- (val->num (num-val 3))
              (val->num (num-val 5))))
= (num-val (- 3 5)) = (num-val -2)
```

Spezifikation von value-of (2/4)

$$\frac{\texttt{(value-of e p)} = \texttt{v}}{\texttt{(value-of (zero?-exp e) p)} = \begin{cases} \texttt{(bool-val \#t)} & \textit{if } \texttt{(val-num v)} = 0 \\ \texttt{(bool-val \#f)} & \textit{if } \texttt{(val-num v)} \neq 0 \end{cases}}$$

Spezifikation von value-of (3/4)

$$\frac{\texttt{(value-of e1 p)} = \texttt{v}}{\texttt{(value-of (if-exp e1 e2 e3) p)} = \begin{cases} \texttt{(value-of e2 p)} & \textit{if } \texttt{(val-bool v)} = \texttt{\#t} \\ \texttt{(value-of e3 p)} & \textit{else} \end{cases}}$$

Spezifikation von value-of (4/4)

$$\frac{\texttt{(value-of e p)} = \texttt{v}}{\texttt{(value-of (let x = e in body) p)} = \texttt{(value-of body [x = v]p)}}$$

Implementierung von value-of

```
; value-of : exp x env -> val
(define value-of
  (lambda (exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (var-exp (var) (aplly-env env var))
      (diff-exp (e1 e2)
        (let ((v1 (value-of e1 env))
              (v2 (value-of e2 env)))
          (let ((n1 (val->num v1))
                (n2 (val->num v2)))
            (num-val (- n1 n2)))))
      (zero?-exp (e)
        (let ((n (val->num (value-of e env))))
          (if (zero? n) (bool-var #t) (bool-var #f))))
      ...)))
```

Implementierung von value-of

```
; value-of : exp x env -> val
(define value-of
  (lambda (exp env)
    (cases expression exp
      ...
      (if-exp (e1 e2 e3)
        (let ((v1 (value-of e1 env)))
          (if (val->bool v1)
            (value-of e2 env)
            (value-of e3 env))))
      (let-exp (x e body)
        (let ((v (value-of e env)))
          (value-of body
            (extend-env x v env)))))))
```