Grundzüge der Informatik

V16. Interpreterbau

Dr. Thomas Kühne
Praktische Informatik
TU Darmstadt

Agenda

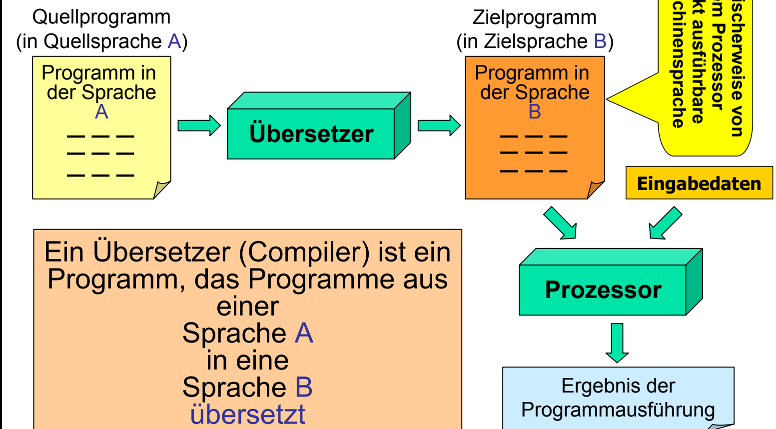
Definitionen
- Unterschied Übersetzer/Interpreter
- EBNF zur Definition der Sprachsyntax

Interpreterphasen
- Analyse (Scanner, Parser)
- Ausführung (AST, Symboltabelle)

Lernziele

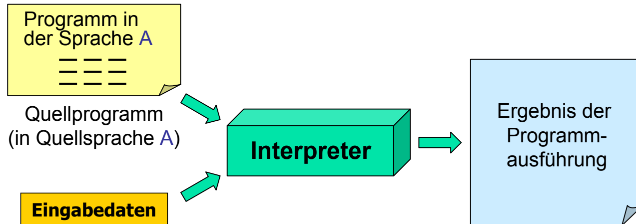
- Kennenlernen und Beherschen von Konzepten im Interpreterbau (EBNF, AST, Speicher, usw.)
- Verstehen der Analysephasen eines Interpreters
- Die Konzepte „Abstrakter Syntaxbaum“ und „Speicher“ zur Ausführung von Programmen nutzen können

Was ist ein Übersetzer?



Was ist ein Interpreter?

Ein Interpreter ist ein Programm, das Programme aus einer Programmiersprache A interpretiert, d.h., ausführt



- Direkte Ausführung des Quellcodes
  - Zwischenschritte (z.B., Transformation in Bytecode) sind üblich

Warum *Interpreter*-bau?

Vorteile

- Im Gegensatz zum Compiler ist keine Synthesephase erforderlich
  - Zielsprache B muß nicht erlernt werden
  - Die Generierung des Programms in der Zielsprache B ist nicht notwendig
- Direkte Ausführung
  - keine zwischengeschaltete Übersetzungsphase
  - Laufzeitumgebung für Zielsprache B wird nicht benötigt

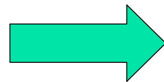
Nachteile

- Effizienzverlust
  - Ausführungsgeschwindigkeit deutlich langsamer
  - spielt hier keine Rolle

Interpreterphasen

Ein großer Schritt...

```
int a=0;

while (a<5)
{
  a=1+2*a;
}
```



a=7

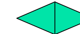
- Das Problem wird beherrschbar, wenn man es in Phasen zerlegt:
  - Analysephase (Quelle in ausführbare Form überführen)
  - Ausführungsphase (Interpretation der ausführbaren Form)

Analysephasen

Problem

- Das Quellprogramm ist nur eine Aneinander-reihung von Zeichen

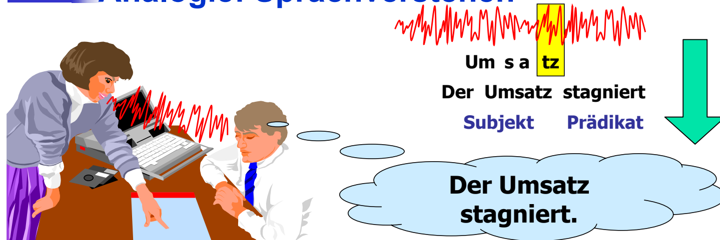
```
int a=0;

while (a<5)
{
  a=1+2*a;
}
```



```
'i' 'n' 't' ' ' 'a' '=' '0'
';' '\n'
'w' 'h' 'i' 'l' 'e' ' ' '('
'a' '<' '5' ')' '\n'
'{' '\n' 'a' '=' '1' '+' '2'
'*'  'a' ';' '\n' '}'
```

- Das Problem wird beherrschbar, wenn man es in Phasen zerlegt:
  - lexikalische Analyse
  - syntaktische Analyse
  - semantische Analyse

Analogie: Sprachverstehen



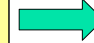
- Die Schallwellen (Quelle) müssen in eine mentale Repräsentation des Inhalts übersetzt werden
  - zunächst werden Schallwellen als Phoneme erkannt
  - die Phoneme werden zu Wörtern verbunden
  - die Wörter werden als Satz erkannt
  - der Satz wird überprüft und interpretiert

Analysephasen

Lexikalische Analyse

- Zunächst werden Zeichen als Lexeme erkannt

```
'i' 'n' 't' ' ' 'a' '=' '0'
';' '\n'
'w' 'h' 'i' 'l' 'e' ' ' '('
'a' '<' '5' ')' '\n'
'{' '\n' 'a' '=' '1' '+' '2'
'*' 'a' ';' '\n' '}'
```



```
<int> <a> <=> <0>
<;> <while> <a>
<5> <)> <{> <a>
<=> <1> <+> <2>
<*> <a> <;>
<}>
```

- Aufgabe des Lexers (Teil des Scanners)
  - ignoriert "whitespace" (Space, Zeilenende, Kommentare)
  - erkennt Schlüsselworte, Bezeichner, Operatoren, Klammern und einfache Symbole
  - erkennt nur Zahlenbestandteile (<12> <.> <45>)

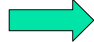
Analysephasen

Lexikalische Analyse

Token:
Typ = Zahl
Inhalt = 5

- Die Lexeme werden zu Wörtern verbunden

```
<int> <a> <=> <0>
<;> <while> <a>
<5> <)> <{> <a>
<=> <1> <+> <2>
<*> <a> <;>
<}>
```



```
keyword(int) id(a) simple(=)
num(0) simple(;) keyword(while)
id(a) num(5) bracket())
bracket({) id(a) simple(=)
num(1) op(+) num(2) op(*) id(a)
simple(;) bracket(})
```

- Aufgabe des Scanners
  - klassifiziert Lexeme in Schlüsselworte, Bezeicher, usw.
  - produziert einen Tokenstrom
  - Zahlentoken sind komplett (num(12.45))

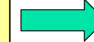
Analysephasen

Syntaktische Analyse

AST-Knoten:
Typ = Deklaration
Inhalt = Typ int, Bez. a, Wert 0

- Wortfolgen werden als Satz erkannt

```
keyword(int)
id(a) simple(=)
num(0) simple(;)
keyword(while)
id(a) num(5)
bracket()) ...
```

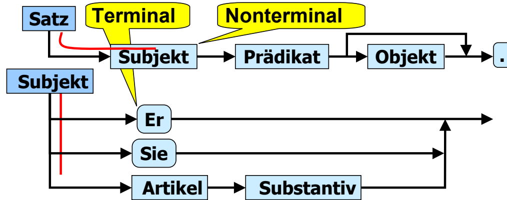


```
declaration(int, a, 0)
while(lowerThan(a, 5),
      assign(a, plus(1, mul(2, a))))
     )
```

- Aufgabe des Parsers
  - faßt Tokens zu Satzbestandteilen zusammen
  - produziert einen abstrakten Syntaxbaum
  - Grundlage ist eine Grammatikbeschreibung, die festlegt an welcher Stelle welche Phrasen legal sind

Grammatikbeschreibung

Analogie Spracherkennung



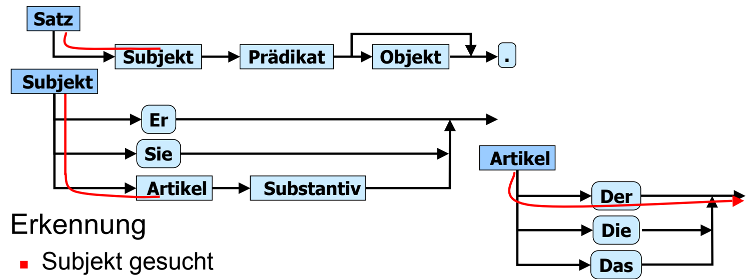
- Erkennung
  - Subjekt gesucht
    - "Der ..." ≠ "Er" ≠ "Sie" => Artikel gefordert

"Der Umsatz stagniert."

Satz(..., ..., ...)

Grammatikbeschreibung

Analogie Spracherkennung



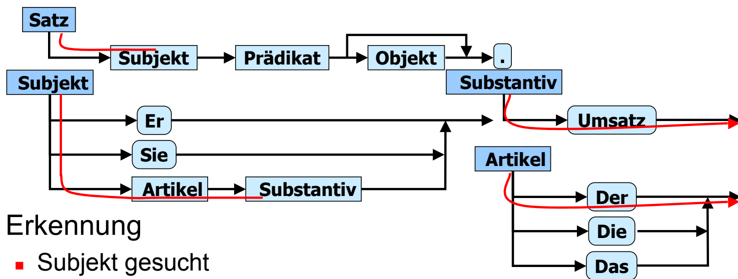
- Erkennung
  - Subjekt gesucht
    - Artikel gesucht

"Der Umsatz stagniert."

Satz(Artikel(...), ..., ...)

Grammatikbeschreibung

Analogie Spracherkennung



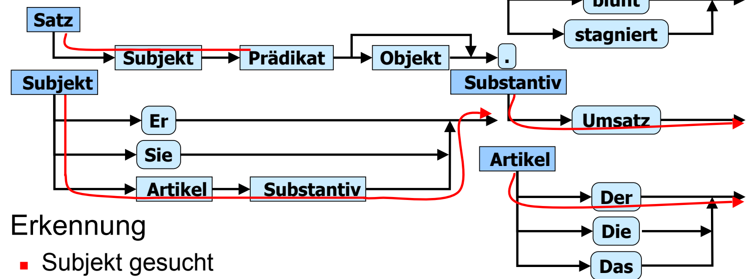
- Erkennung
  - Subjekt gesucht
    - Artikel gesucht
    - Substantiv gesucht

"Umsatz stagniert."

Satz(Artikel("Der"), ..., ...)

Grammatikbeschreibung

Analogie Spracherkennung



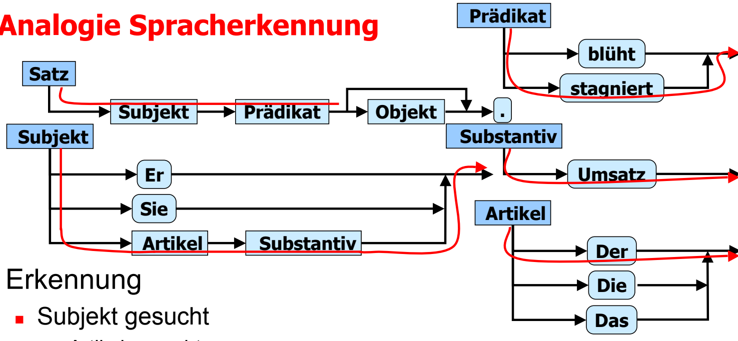
- Erkennung
  - Subjekt gesucht
    - Artikel gesucht
    - Substantiv gesucht
  - Prädikat gesucht

"stagniert."

Satz(Artikel("Der"), Substantiv("Umsatz"), ...)

Grammatikbeschreibung

Analogie Spracherkennung



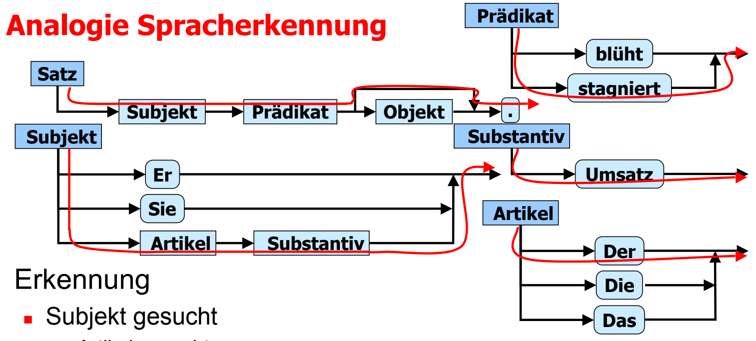
- Erkennung
  - Subjekt gesucht
    - Artikel gesucht
    - Substantiv gesucht
  - Prädikat gesucht

"stagniert."

Satz(Artikel("Der"), Substantiv("Umsatz"), ...)

Grammatikbeschreibung

Analogie Spracherkennung



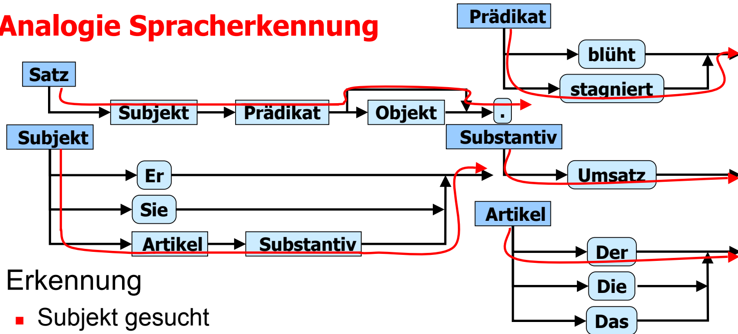
- Erkennung
  - Subjekt gesucht
    - Artikel gesucht
    - Substantiv gesucht
  - Prädikat gesucht
  - Objekt nicht vorhanden

"."

Satz(Artikel("Der"), Substantiv("Umsatz")), Prädikat("stagniert"),...)

Grammatikbeschreibung

Analogie Spracherkennung



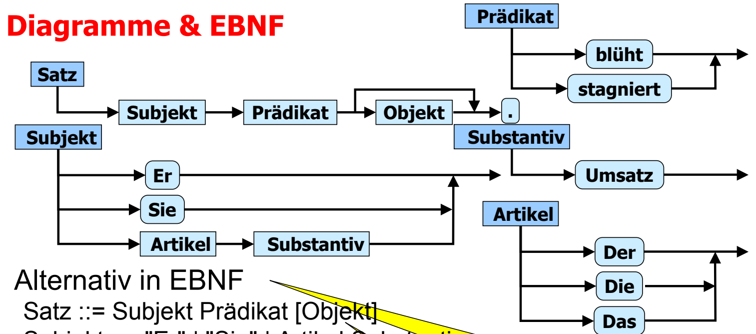
- Erkennung
  - Subjekt gesucht
    - Artikel gesucht
    - Substantiv gesucht
  - Prädikat gesucht
  - Objekt nicht vorhanden

" "

Satz(Artikel("Der"), Substantiv("Umsatz")), Prädikat("stagniert"))

Grammatikbeschreibung

Diagramme & EBNF



- Alternativ in EBNF

  Satz ::= Subjekt Prädikat [Objekt]
  Subjekt ::= "Er" | "Sie" | Artikel Substantiv
  Prädikat ::= "blüht" | "stagniert"
  Substantiv ::= "Umsatz"
  Artikel ::= "Der" | "Die" | "Das"

Extended Backus Naur Form

Optional

Alternative

Grammatikbeschreibung

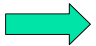
EBNF für die Sprache vom Beispiel (Auszug)

Programm ::= {Statement}
Statement ::= Declaration | WhileStat | AssignStat | Block
Declaration ::= Type Identifier [Initialization]
Initialization ::= "=" Expression
WhileStat ::= "while" "(" Expression ")" Statement
AssignStat ::= Identifier "=" Expression
Block ::= "{" {Statement} "}"
Type ::= "int" | "boolean"

"Niedriger Einstieg" ohne Klassen oder Methoden

Wiederholung (0..n)

```
int a=0;

while (a<5)
{
  a=1+2*a;
}
```



```
program(statements(
        declaration(int, a, 0),
        while(Expression,
          assign(a, Expression)
        )
      )
```

Grammatikbeschreibung

EBNF für die Sprache vom Beispiel (Auszug)

Expression ::= AddExpr [("<" | ">") AddExpr]
AddExpr ::= MultExpr [("+" | "-") AddExpr]
MultExpr ::= UnaryExpr [("*" | "/") MultExpr]
UnaryExpr ::= Literal | Identifier

Warum so kompliziert?

Priorität der Operatoren muß beachtet werden!

`1+2*a;` → ! → `plus(2, mul(2, a))`

Der Entwurf der Grammatik ermöglicht es Ausdrücke ohne überflüssige Klammern hinschreiben zu können

`a<2*5;` → ! → `lower(a, mul(2, 5))`
`a<2*5;` → `mul(lower(a, 2), 5))`

Parser

Erkennen von Phrasen anhand einer Grammatik

- Nichtterminale der Grammatik werden zu Funktionen, die Phrasen erkennen
- Alternativen werden der Reihe nach erprobt
- In jeder einzelnen Regel werden Nichtterminale zu Funktionsaufrufen und die Terminale werden geprüft, ob sie in der Eingabe stehen

AssignStat ::=
Identifier "=" Expression

Funktion erkennt Zuweisung

Statement ::=
Declaration | WhileStat |
AssignStat | Block

Überprüfen der Eingabe nach diesen Möglichkeiten

WhileStat ::=
"while" "(" Expression ")"
Statement

Funktionsaufruf

Muß in der Eingabe vorkommen

Analysephasen

Sematische Analyse

- Der Satz wird auf Konsistenz überprüft

vom Typ "int"? **vom Typ "boolean"?**

```
declaration(int, a, 0)
while(lowerThan(a, 5),
      assign(a, plus(1, mul(2, a))))
)
```

wurde "a" bereits deklariert? **Addition auf Ergebnistyp definiert?**

- Aufgabe des Typecheckers
  - überprüft die inhaltliche Wohlgeformtheit des Satzes
  - Analyse ergibt "true" oder "false" + Fehlerbeschreibung
  - kann auch vollständig in die Ausführungsphase verlegt werden => späteres Erkennen von Fehlern

u. U. nie, falls der entsprechende Code nie erreicht wird

Ausführungsphase

Interpretation des Programms

- Der Satz wird interpretiert

Speicher: Abbildung von Bezeichnern auf Werte

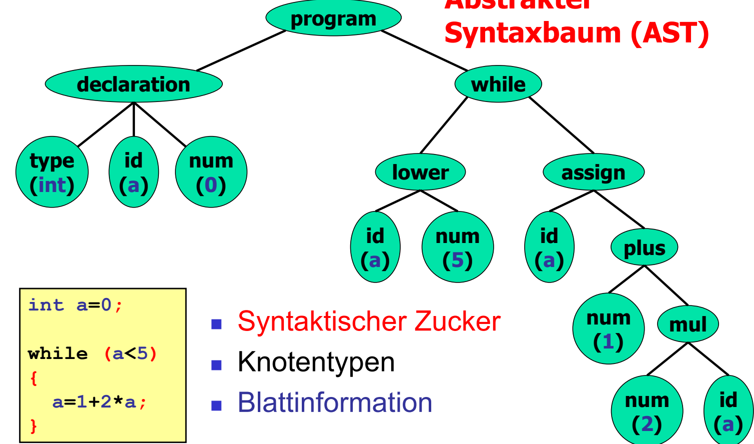
```
declaration(int, a, 0)
while(lowerThan(a, 5),
      assign(a, plus(1, mul(2, a))))
     )
```

```
a=0
a=1 (= 1+2*0)
a=3 (= 1+2*1)
a=7 (= 1+2*3)
```

- Aufgabe des Interpreters
  - interpretiert die einzelnen AST Knoten
  - Effekt ist die Veränderung der Symboltabelle, bzw., Seiteneffekte wie Text- oder grafische Ausgaben
  - muß (mindestens) Laufzeitfehler abfangen

Ausführungsphase

Abstrakter Syntaxbaum (AST)

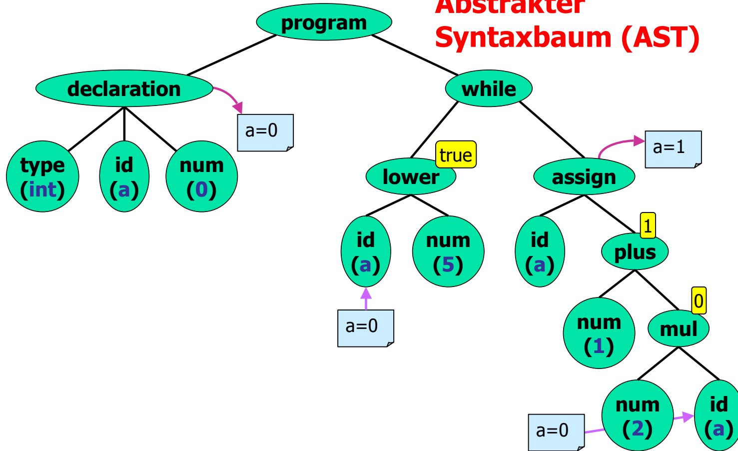



```
int a=0;

while (a<5)
{
  a=1+2*a;
}
```

- Syntaktischer Zucker
- Knotentypen
- Blattinformation

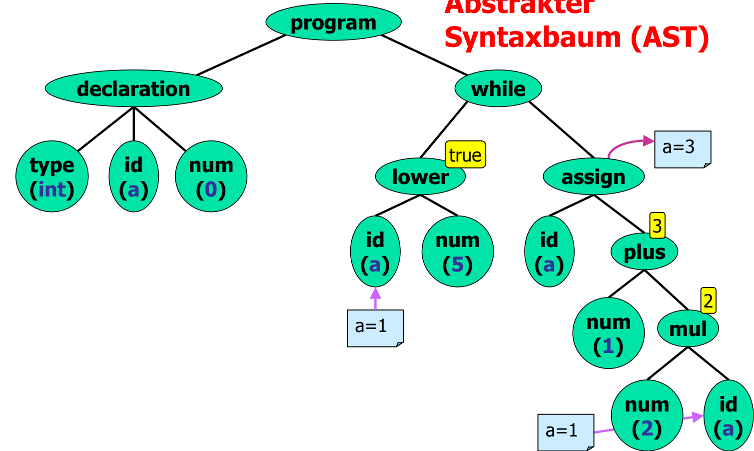
Ausführungsphase

Abstrakter Syntaxbaum (AST)

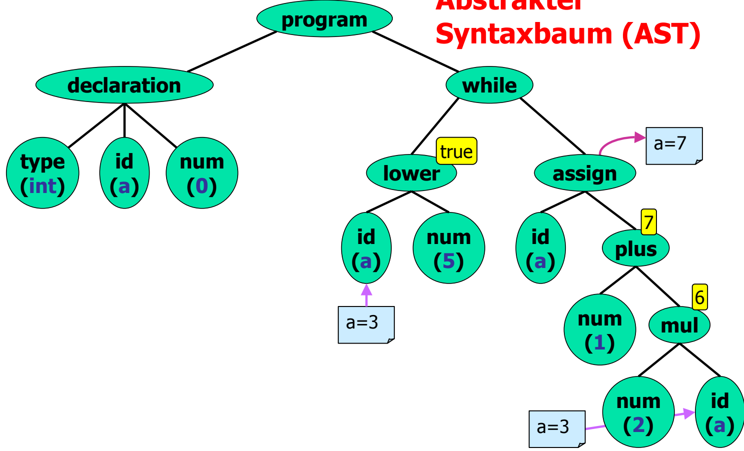


Ausführungsphase

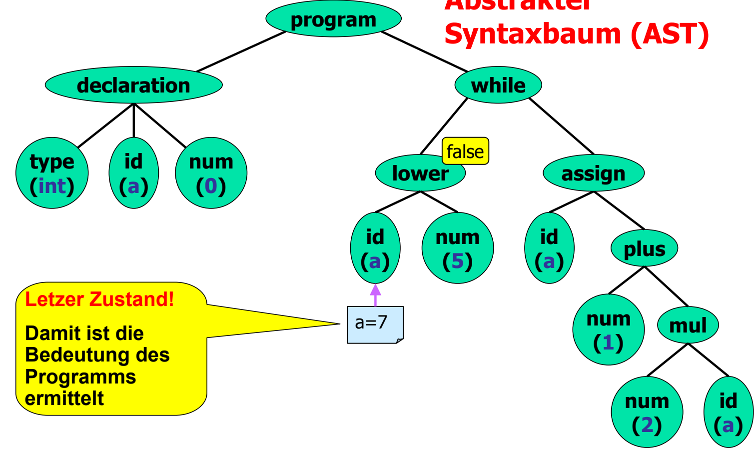
Abstrakter Syntaxbaum (AST)



Ausführungsphase



Ausführungsphase



Fehlertypen

Programmfehler

- lexikalische Fehler: unbekannte Worte

```
...
Karel = neu Robot(...);
weil (a<5,3) {a=a+1};
...
```

Werden durch Scanner entdeckt

- Syntaxfehler: falsche Anordnung von Worten

```
...
move();
task {
        new Robot(0, 1, North, 0);
    }
...
```

Werden durch Parser entdeckt

Fehlertypen

Programmfehler

Werden durch semantische Analyse entdeckt

- Wohlgeformtheitsfehler (hier: Typfehler)
  - das Programm ist syntaktisch aber nicht semantisch korrekt

```
int Karel = new Robot(1, 1, West, 0);
```

- Laufzeitfehler (hier: Error-shutoff)
  - die Programmausführung muß unterbrochen werden

Werden durch Ausführung entdeckt

```
...
Robot Karel = new Robot(1, 1, West, 0);
Karel.move();
```

Fehlertypen

Programmfehler

- Intentionsfehler
  - Programm läuft, aber mit unerwünschtem Ergebnis

Diese Art von Fehler kann von einem Interpreter nicht entdeckt werden und ist bzgl. des Interpreters auch nicht als Fehler anzusehen!

Es sei denn: Das Programm enthält Zusicherungen, welche die Absicht spezifizieren. Der Interpreter kann dann die Zusicherungen zur Laufzeit überprüfen und Abweichungen melden.

Interpreterbau: Zusammenfassung

- Interpretieren von Programmen erfolgt in Phasen
  - Analysephase
  - Ausführungsphase
- Analysephase unterteilt sich in
  - lexikalische Analyse
  - syntaktische Analyse
- Bei der Ausführung
  - enstehen Seiteneffekte (Ausgaben)
  - wird der Speicherzustand verändert