Compiler I: Grundlagen

Einleitung

TECHNISCHE UNIVERSITÄT DARMSTADT

WS 2014/15

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Compiler

TECHNISCHE UNIVERSITÄT DARMSTADT

- Schnittstelle zwischen
  - Programmiersprache
  - Maschine

Programmiersprache Gut für Menschen handhabbar

- Smalltalk
- Java
- C++

Maschine Getrimmt auf

- Ausführungsgeschwindigkeit
- Preis/Chip-Fläche
- Energieverbrauch
- Nur selten: Leichte Programmierbarkeit

Auswirkung von Compilern

TECHNISCHE UNIVERSITÄT DARMSTADT

Entscheidet über dem Benutzer zugängliche Rechenleistung

Beispiel: Bildkompression auf Dothan CPU, 2GHz

Compiler	Ausführungszeit	Programmgröße
GCC 3.3.6	7,5 ms	13 KB
ICC 9.0	6,5 ms	511 KB

Programmiersprachen

TECHNISCHE UNIVERSITÄT DARMSTADT

Hohe Ebene:
Smalltalk, Java, C++

```
let
  var i : Integer;
in
  i := i + 1;
```

Mittlere Ebene:
Assembler

```
LOAD  R1, (i)
LOADI R2, 1
ADD   R1, R1, R2
STORE R1, (i)
```

Niedrige Ebene:
Maschinensprache

```
0110000100000110
0111001001000001
1011000100010010
1001000100000110
```

Unterschiedliche Abstraktionsebenen

TECHNISCHE UNIVERSITÄT DARMSTADT

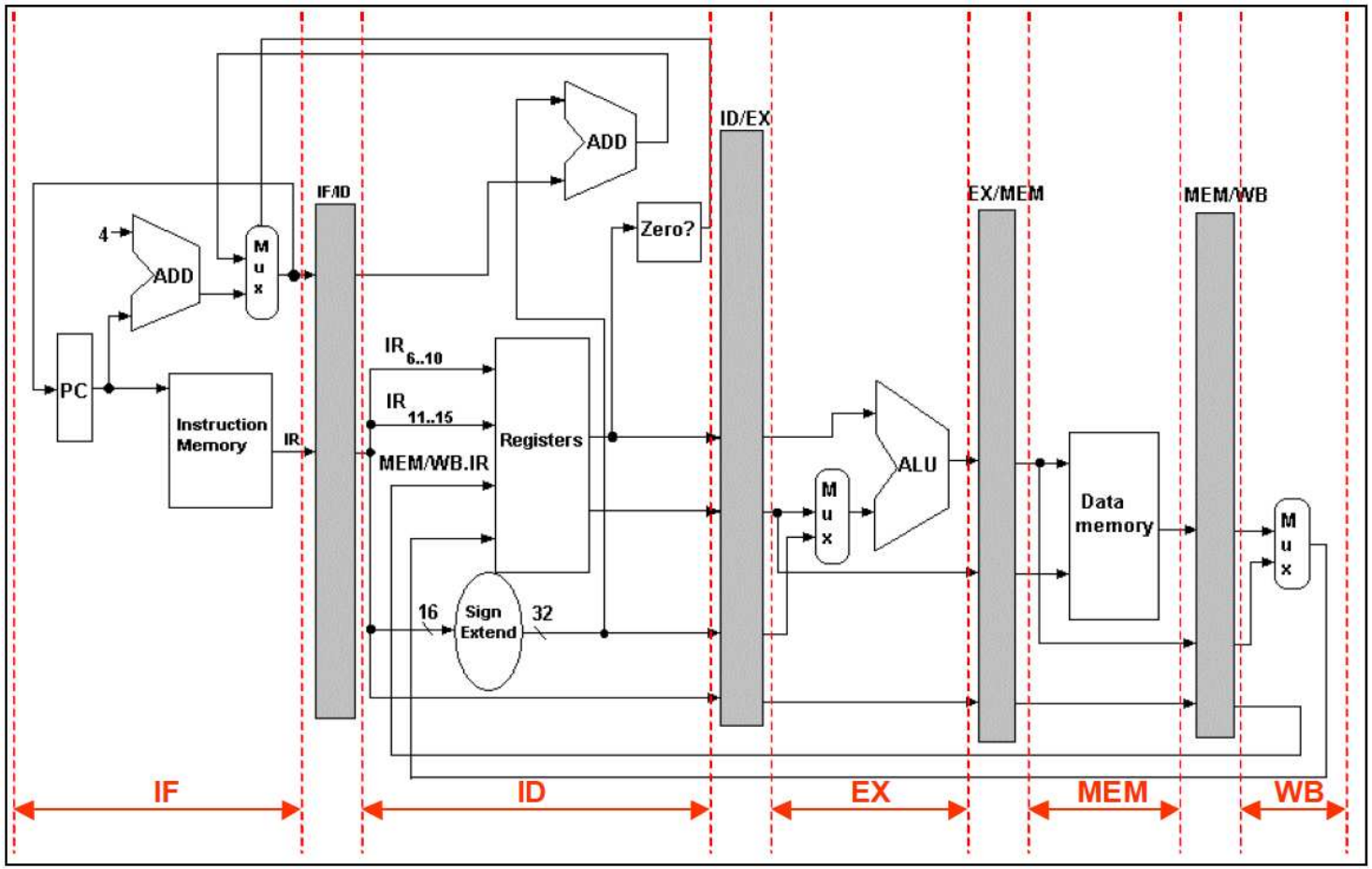
- Auf unteren Ebenen immer feinere Beschreibung
- Immer näher an Zielmaschine (Hardware)
- Details werden von Compiler hinzugefügt
  - Durch verschiedenste Algorithmen
    - Analyse von Programmeigenschaften
    - Verfeinerung der Beschreibung durch Synthese von Details

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswirkungen der Zielmaschine

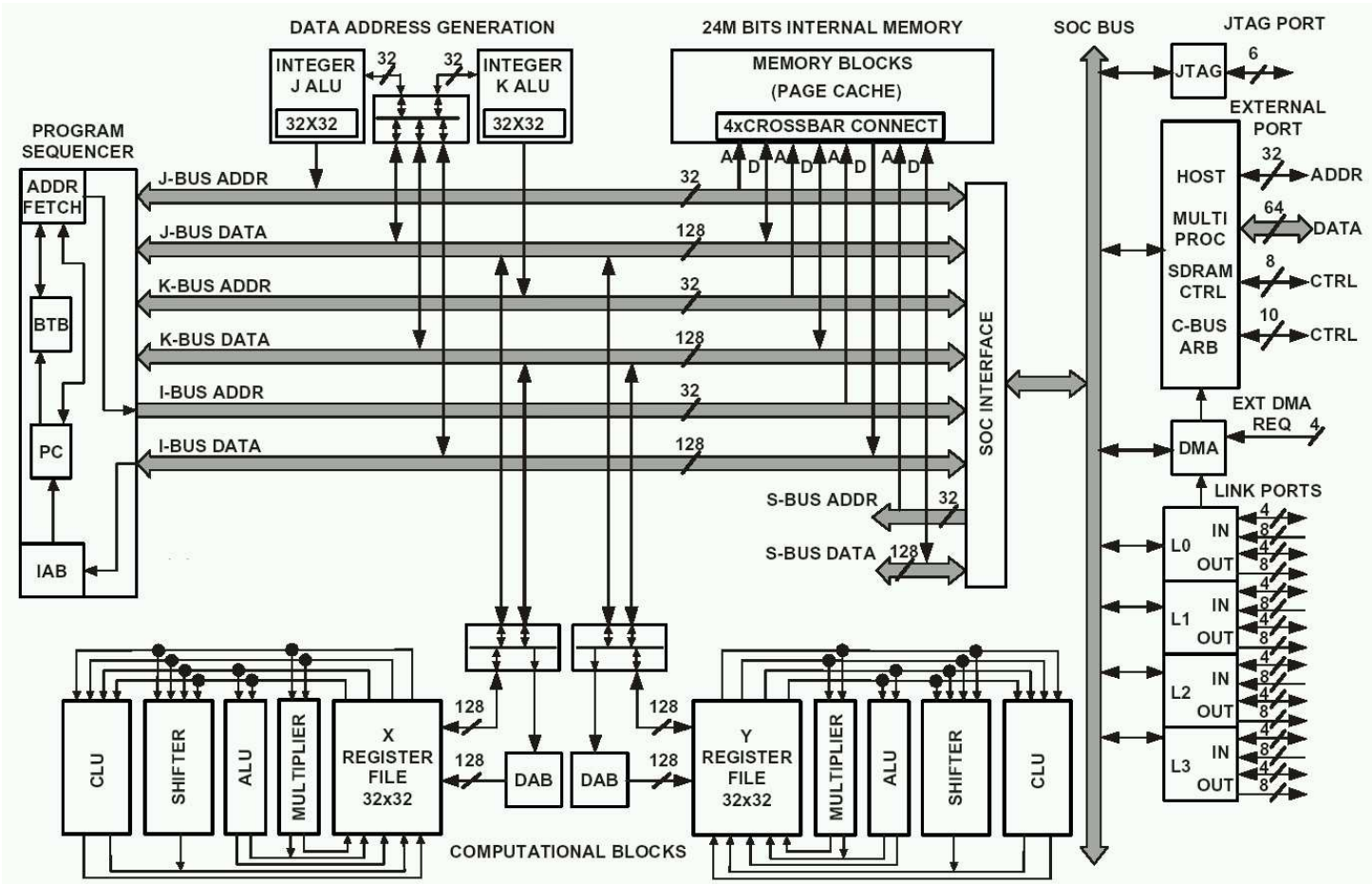
Einfach: Hennessy & Patterson DLX

TECHNISCHE UNIVERSITÄT DARMSTADT



Komplizierter: Analog Devices TigerSHARC

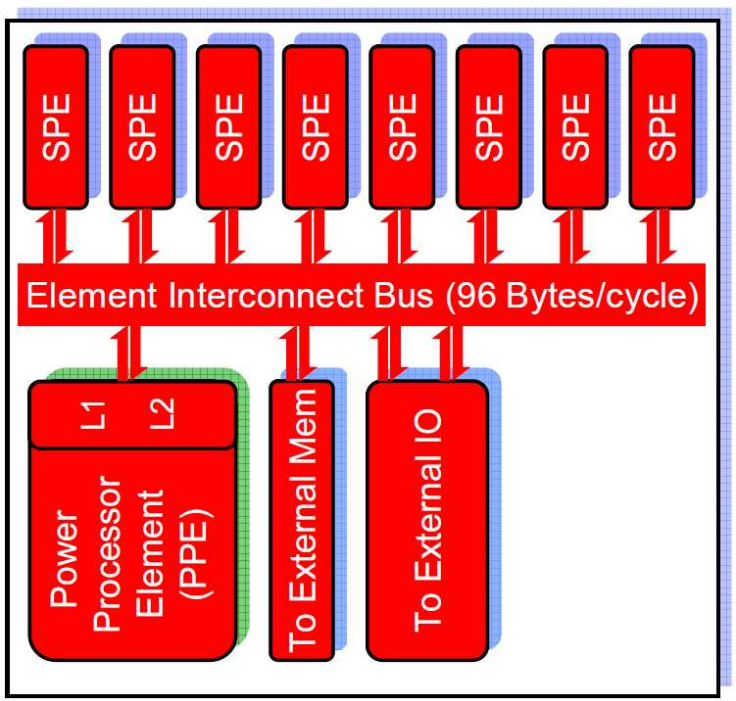
TECHNISCHE UNIVERSITÄT DARMSTADT



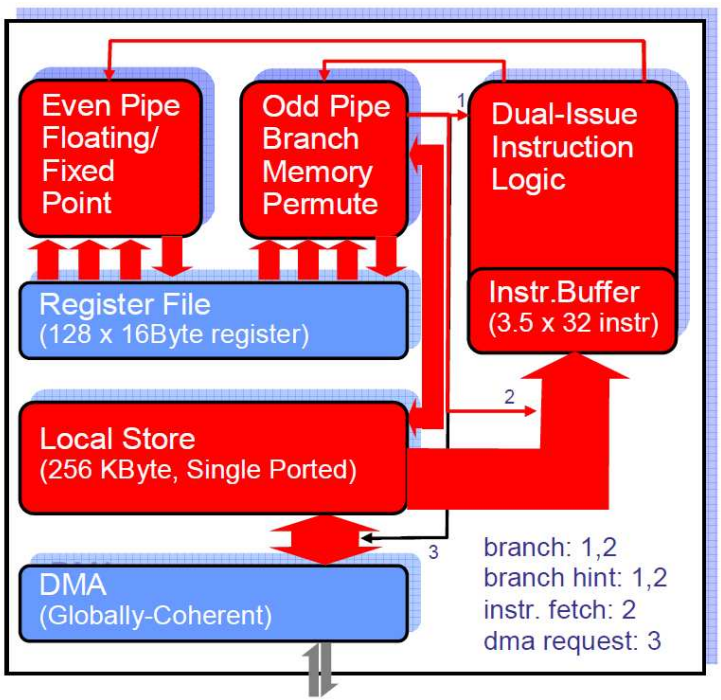
Problematisch: IBM/Sony Cell Processor

TECHNISCHE UNIVERSITÄT DARMSTADT

Übersicht



SPE



Spezialisierte Anforderungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Rechenleistung (hoch/niedrig)
- Datentypen (Gleitkomma, ganzzahlig, Vektoren)
- Operationen (Multiplikationen, MACs)
- Speicherbandbreite (parallele Speicherzugriffe)
- Energieeffizienz
- Platzbedarf

. . . können häufig nur durch spezialisierte Prozessoren erfüllt werden

➥**Benötigen passende Compiler**

Königsklasse: Paralleles Rechnen

TECHNISCHE UNIVERSITÄT DARMSTADT

DOI:10.1145/1461928.1461946

Research and education in compiler technology is more important than ever.

BY MARY HALL, DAVID PADUA, AND KESHAV PINGALI

Compiler Research: The Next 50 Years

Communications of the Association for Computing Machinery (CACM), Februar 2009

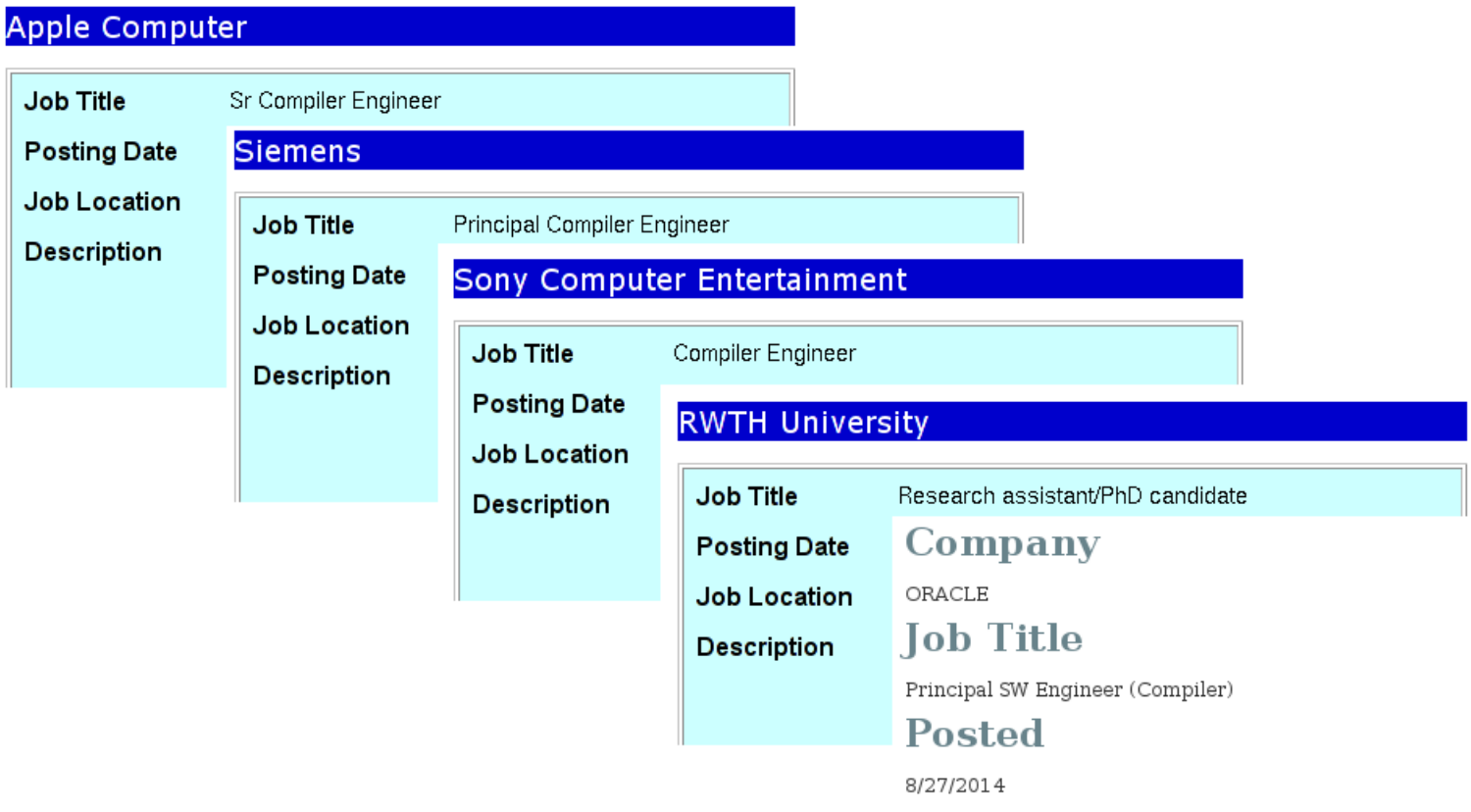
Technologie

TECHNISCHE UNIVERSITÄT DARMSTADT

- Taktfrequenz von Prozessoren nur mit Mühe steigerbar
- Trend weg von hochgetakteten Einzelprozessoren
- . . . hin zu vielen (aber langsameren) Prozessoren
- Wie parallele Strukturen programmieren?
- Erste praktische Ansätze
  - OpenMP: Mehr-Kern-CPUs
  - NVidia CUDA: GPUs ($\rightarrow$ PMPP Michael Gösele)
  - OpenCL: Heterogene Systeme (GPUs+CPUs)
- Aber noch wenig abstrakt
- *Keine* automatische Parallelisierung!

Spezialisten gesucht

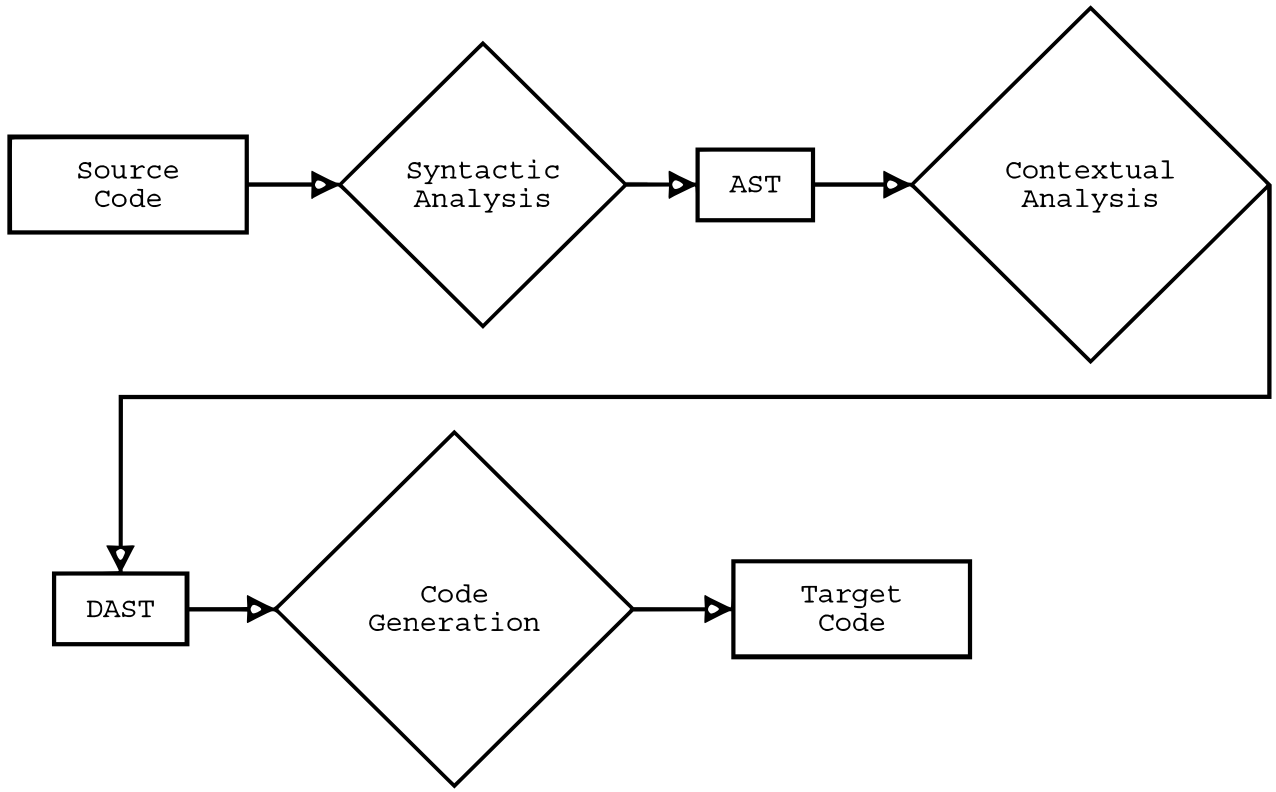
TECHNISCHE UNIVERSITÄT DARMSTADT



Aufbau von Compilern

Vorgehen: Bearbeitung in mehreren Phasen

TECHNISCHE UNIVERSITÄT DARMSTADT



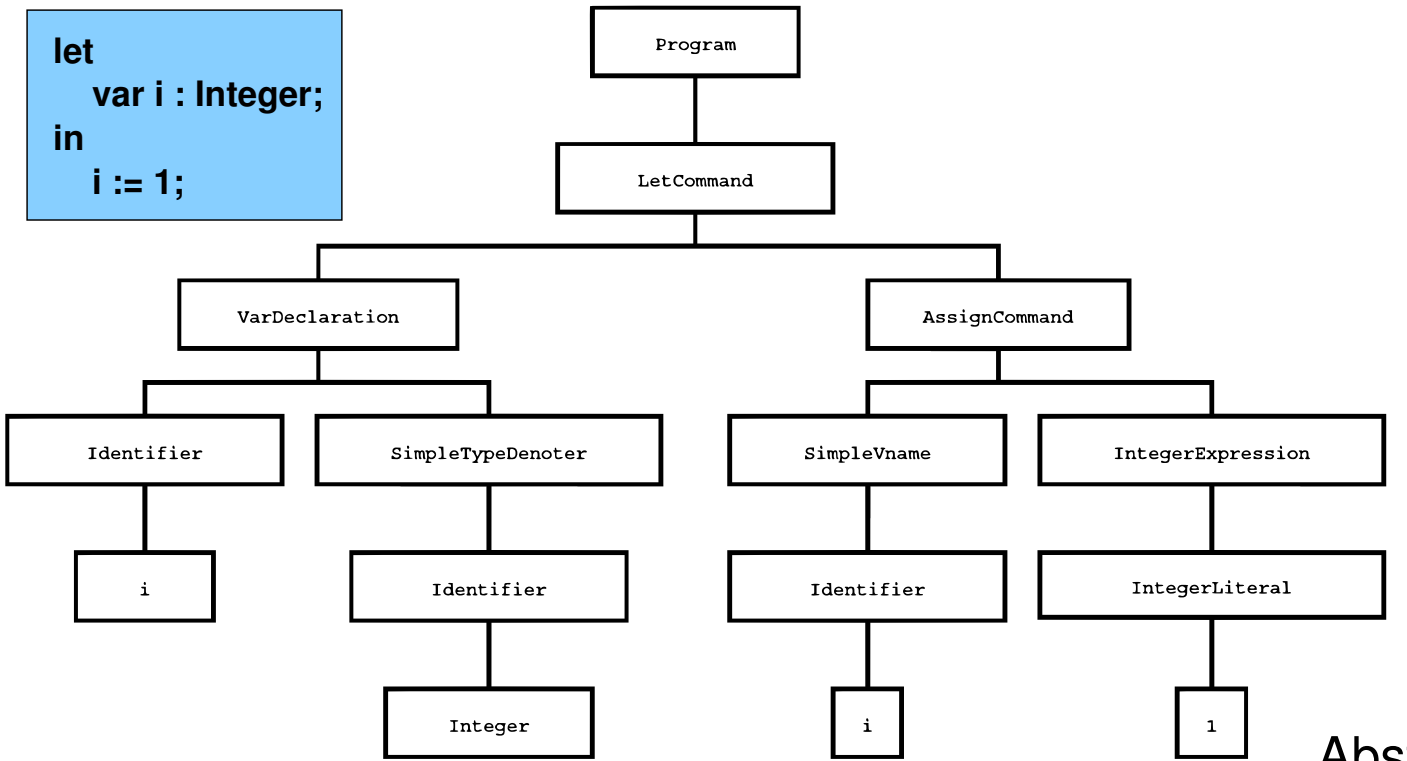
Zwischendarstellung(en) für den Informationsaustausch

Syntaxanalyse

TECHNISCHE UNIVERSITÄT DARMSTADT

- Überprüfung ob Programm Syntaxregeln gehorcht
- Speichern des Programmes in geeigneter Darstellung

```
let
   var i : Integer;
in
   i := 1;
```

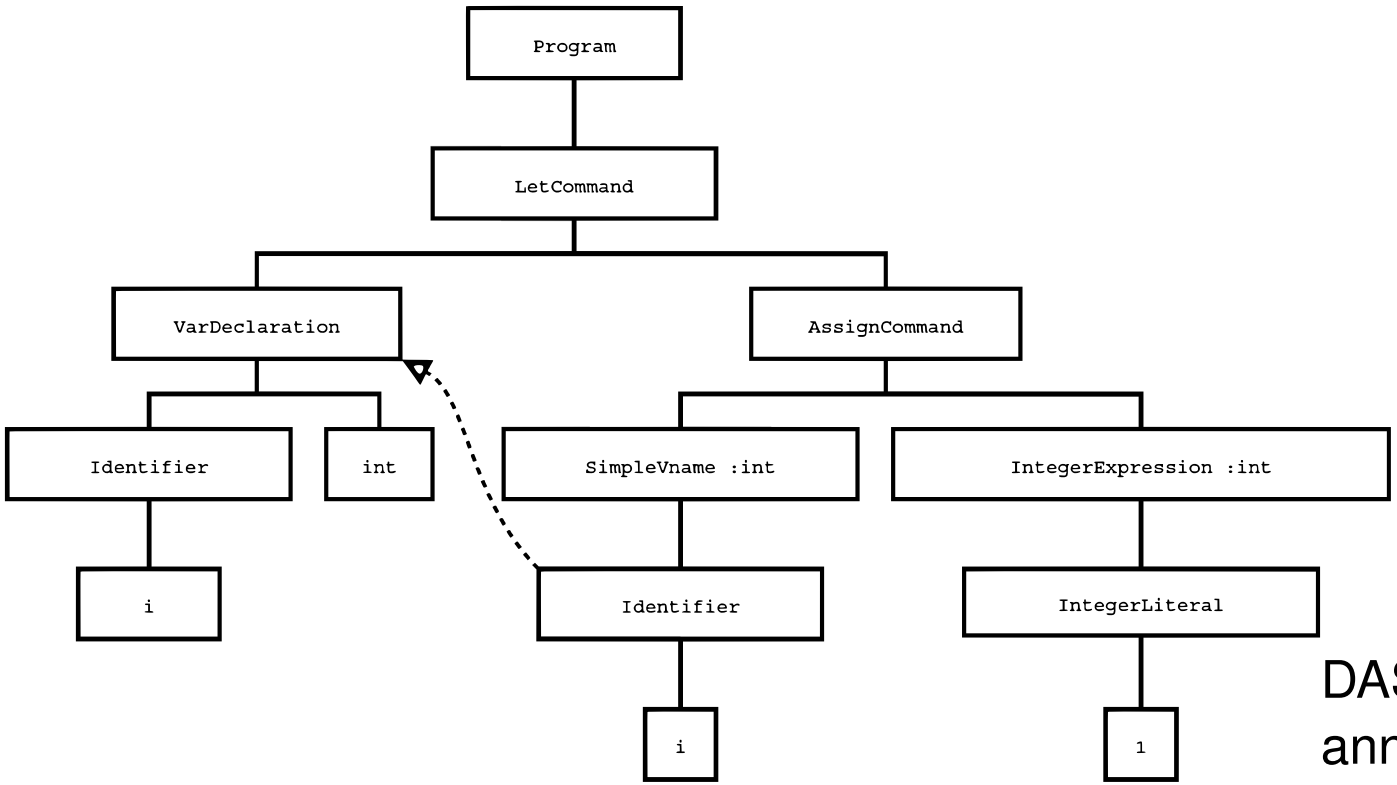


Abstrakter Syntaxbaum

Kontextanalyse - Identifikation

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ordne Variablen ihren Deklarationen zu
- Berechne Typen von Ausdrücken

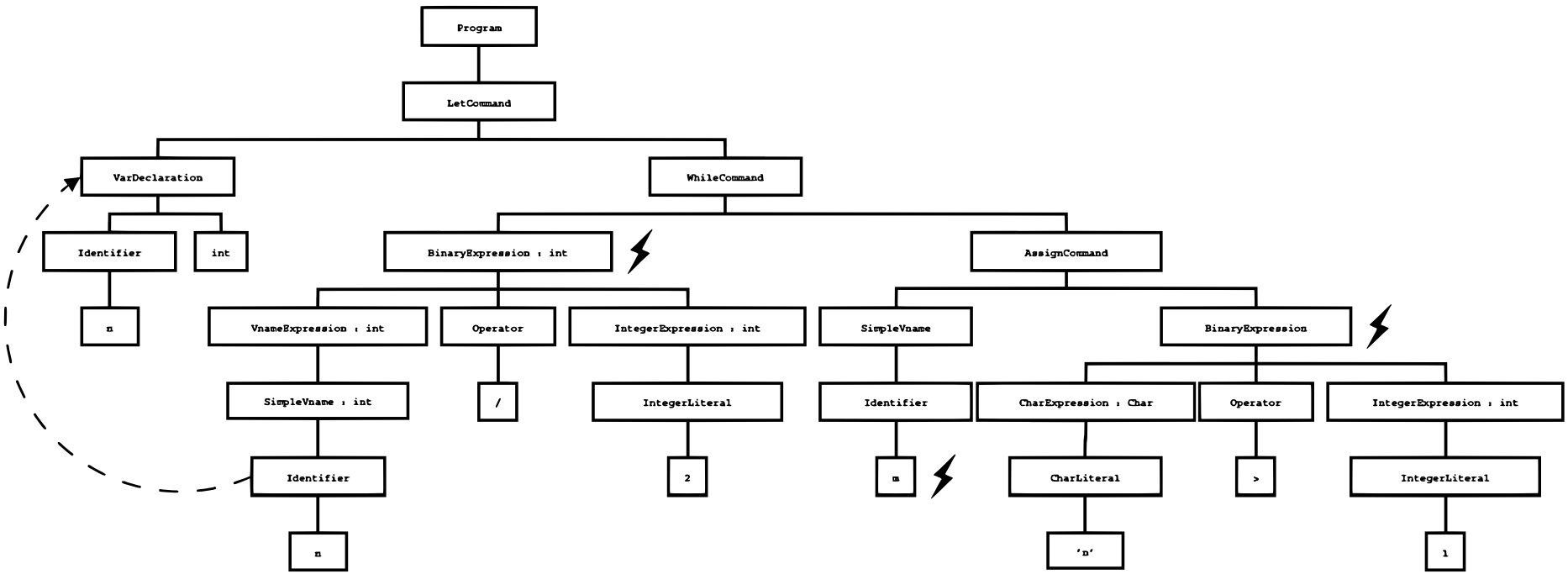


DAST: Dekorierter bzw. annotierter AST

Kontextanalyse - Überprüfung

TECHNISCHE UNIVERSITÄT DARMSTADT

Erkenne Fehler in Variablen- und Typzuordnung

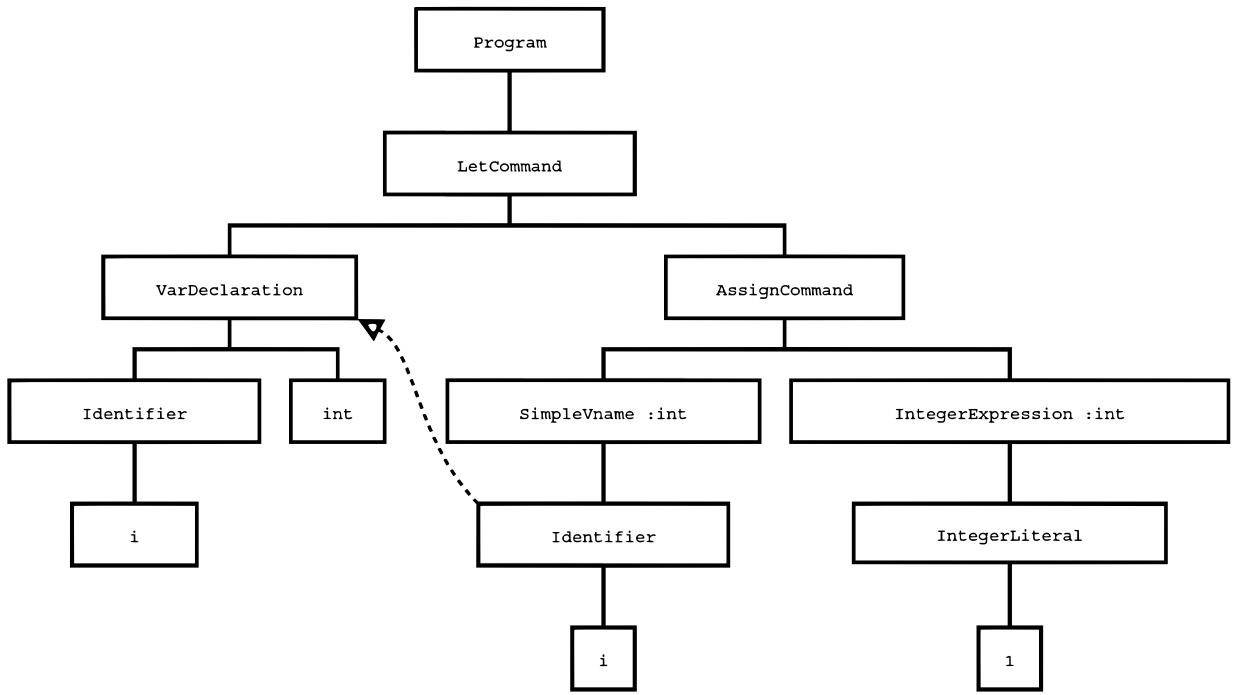


Code-Erzeugung 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Programm ist syntaktisch und kontextuell korrekt
- Übersetzung in Zielsprache
  - Maschinensprache
  - Assembler
  - C
  - Andere Hochsprache
- Ordne DAST-Teilen Instruktionen der Zielsprache zu
- Handhabung von Variablen
  1. Deklaration: Reserviere Speicherplatz für eine Variable
  2. Verwendung: Referenziere immer den zugeordneten Speicherplatz

Code-Erzeugung 2




```
0:  PUSH      1          ; Platz fuer 'i' schaffen, Adresse 0[SB]
1:  LOADL     1          ; Wert 1 auf Stack legen
2:  STORE (1) 0[SB]      ; ein Datenwort vom Stack nach Adresse 0[SB] schreiben
3:  POP (0)   1          ; Platz von 'i' wieder aufgeben
4:  HALT                 ; Ausfuehrung beenden
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Optimierung

Optimierender Compiler

TECHNISCHE UNIVERSITÄT DARMSTADT

- Front-End: Syntaktische/kontextuelle Analyse
- Back-End: Code-Erzeugung
- Middle-End: Transformation von Zwischendarstellungen
  - Intermediate Representation (IR)
  - Keine direkte Code-Erzeugung aus Front-End IR
  - Verwendet in der Regel zusätzliche interne Darstellungen

Ziel: Verbesserung des erzeugten Codes in Bezug auf bestimmte Gütemaße

Beispiele für Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Constant-Folding

```
x = (2+3) * y
```

```
x = 5*y
```

Common-Subexpression Elimination

```
x = 5 * a + b;
y = 5 * a + c;
```

```
t = 5 * a;
x = t + b;
y = t + c;
```

Strength Reduction

```
for (i=0; i <= j; ++i) {
   a[i*3] = 42;
}
```

```
int t = 0;
for (i=0; i <= j; ++i) {
   a[t] = 42;
   t = t + 3;
}
```

Loop-invariant Code Motion

```
int t;
for (i=0; i <= j; ++i) {
   t = x * y;
   a[i] = t * i;
}
```

```
int t = x * y;
for (i=0; i <= j; ++i) {
   a[i] = t * i;
}
```

Organisatorisches

Dozent

TECHNISCHE UNIVERSITÄT DARMSTADT

Dozent

Andreas Koch — koch@esa.informatik.tu-darmstadt.de

Sprechstunde — i.d.R. Mi 14:00-15:00, E101

Assistent

Jens Huthmann — jh@esa.informatik.tu-darmstadt.de

Sprechstunde — i.d.R. Mi 14:00-15:00, E121

Web-Seite

`http://www.esa.cs.tu-darmstadt.de`

Material: Grundlage der Vorlesung

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlagen von Compilern: Fast vollständig

Programming Language Processors in Java
von David Watt und Deryck Brown, Prentice-Hall 2000

Auszugsweise noch weiteres Material, z.B. zum ANTLR Parsergenerator.

Übersichtswerk

TECHNISCHE UNIVERSITÄT DARMSTADT

Guter allgemeiner Überblick, aber im Detail mit anderen Schwerpunkten als bei uns

Compilers, 2. Auflage (!)
von Aho, Sethi, Ullmann, Lam, Addison-Wesley 2006

Auch auf Deutsch verfügbar

Veranstaltungskonzept

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vorlesung
  - Theoretische und algorithmische Grundlagen
  - Semesterbegleitende Prüfung als Klausur gegen Ende der Vorlesungszeit
- Optionale Hausaufgaben zur Wissensvertiefung
  - Kompakte Übungsaufgaben
  - Fünf Aufgabenblätter
  - Abgaben in Gruppen möglich
  - Keine größeren Programmierprojekte mehr
    $\rightarrow$ Praktikum zu Compiler II im WS 14/15
  - Stattdessen Überprüfung des eigenen Lernerfolgs

Aufbau der Veranstaltung 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Überblick über Front-End[1] (ca. 3 Wochen)

- Lexing/Parsing
- Zwischendarstellungem

Überblick über Middle-End[1] (ca. 2 Wochen)

- Semantische / Kontextanalyse

Überblick über Back-End[1] (ca. 4 Wochen)

- Laufzeitorganisation
- Code-Erzeugung

[1]: Diese Teile lehnen sich an an die Veranstaltungen

- IMT3052 von Ivar Farup, Universität Gjøvik, Norwegen
- Vertalerbouw von Theo Ruys, Universität Twente, Niederlande

Aufbau der Veranstaltung 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Weiterführende Themen

- Verwendung von Front-End-Generatoren (ca. 2-3 Wochen)
- Ausblick (mögliche Themen z.B.: Java Bytecodes, Compilierung für FPGAs)

Syntax

Syntax oder Grammatik

TECHNISCHE UNIVERSITÄT DARMSTADT

Beschreibt die Satzstruktur von korrekten Programmen

- `n := n + 1;`
  Syntaktisch korrektes Statement in Triangle
- “Ein Kreis hat zwei Ecken.”
  Syntaktisch korrekte Aussage in Deutsch

Kontextuelle Einschränkungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Dazu gehören Regeln für den Geltungsbereich (*scope*) und den Typ von Aussagen.

- `n` muß bei Auftreten des Statements passend deklariert sein.
- Kreise haben im allgemeinen keine Ecken.
  Hier passen die Typen offenbar nicht.

Semantik

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Bedeutung einer Anweisung/Aussage in einer Sprache. Wird bei Programmiersprachen häufig beschrieben . . .

Operationell Welche Schritte laufen ab, wenn das Programm gestartet wird?

Denotational Abbildung von Eingaben auf Ausgaben

Art der Spezifikation

TECHNISCHE UNIVERSITÄT DARMSTADT

- Für alle drei Teile
  1. Syntax
  2. Kontextuelle Einschränkungen
  3. Semantik
- . . . gibt es jeweils zwei Spezifikationsarten
  - Formal
  - Informal

Triangle-Spezifikation

- Formale Syntax (reguläre Ausdrücke, EBNF)
- Informale kontextuelle Einschränkungen
- Informale Semantik

Syntax

TECHNISCHE UNIVERSITÄT DARMSTADT

Eine **Sprache** ist eine Menge von **Zeichenketten** aus einem **Alphabet**

- Wie diese Menge angeben?
- Bei endlichen Sprachen: Einfach Elemente aufzählen
- Geht nicht bei unendlichen Sprachen
- Mögliche Vorgehensweisen
  1. Mathematische Mengennotation
  2. Reguläre Ausdrücke
  3. Kontextfreie Grammatik

Syntax durch Mengenbeschreibung

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele für die beschriebenen Zeichenketten

- $L = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ beschreibt **a**, **b**, **c**
- $L = \{\mathbf{x}^n | n > 0\}$ beschreibt **x**, **xx**, **xxx**, ...
- $L = \{\mathbf{x}^n\mathbf{y}^m | n > 0, m > 0\}$ beschreibt **xxy**, **xyy**, **xxxyy**, ...
- $L = \{\mathbf{x}^n\mathbf{y}^n | n > 0\}$ beschreibt **xy**, **xxyy**, ..., aber z.B. nicht **xxy**

Offensichtlich keine sonderlich nützliche und gut zu handhabende Spezifikationsform für komplexere Sprachen.

Reguläre Ausdrücke (REs)

TECHNISCHE UNIVERSITÄT DARMSTADT

Erweitere Zeichenketten aus dem Alphabet um Operatoren

- $|$ zeigt Alternativen an
- $*$ zeigt Null oder mehr Vorkommen des vorangehenden Zeichens an
- ε ist die leere Zeichenkette
- $(\ldots)$ erlauben die Gruppierung von Teilausdrücken durch Klammerung

Beispiele

- $L = \mathbf{a}|\mathbf{b}|\mathbf{c}$ ergibt **a**, **b**, **c**
- $L = \mathbf{ab}*$ ergibt **a**, **ab**, **abb**, ...
- $L = (\mathbf{ab})*$ ergibt die leere Zeichenkette ε, **ab**, **abab**, **ababab**, ...
- $L = \mathbf{a}(\mathbf{b}|\varepsilon)$ ergibt **a** oder **ab**

Mächtigkeit von REs?

TECHNISCHE UNIVERSITÄT DARMSTADT

Kann man die Menge aller regulären Ausdrücke selber durch einen regulären Ausdruck beschreiben?

Nein! REs sind daher ungeeignet zur Beschreibung der Syntax komplexer Programmiersprachen

. . . also Weitersuchen nach geeigneter Beschreibungsform für Syntax

Aber: REs sind trotzdem innerhalb eines Compilers nützlich
(siehe PLPJ Kapitel 4, Scanner).

Kontextfreie Grammatiken (CFGs) 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Eine kontextfreie Grammatik besteht aus

- Einer Menge von Terminalsymbolen T aus Alphabet
- Einer Menge von Nicht-Terminalsymbolen N
- Einem Startsymbol $S \in N$
- Einer Menge von Produktionen P
  - Beschreiben, wie Nicht-Terminalsymbole aus Terminalsymbolen zusammengesetzt sind.

Kontextfreie Grammatiken (CFGs) 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Produktionen in Backus-Naur-Form (BNF)

Nicht-Terminal ::= **Zeichenkette** aus Terminal und Nicht-Terminalsymbolen

Produktionen in Extended BNF (EBNF)

Nicht-Terminal ::= **RE** aus Terminal und Nicht-Terminalsymbolen

BNF Beispiel 1

TECHNISCHE UNIVERSITÄT DARMSTADT

$T = \{\mathbf{x}, \mathbf{y}\}$, $N = \{\mathbf{S}, \mathbf{B}\}$, $S = \mathbf{S}$, $P = \{$

$$
\begin{array}{lcll}
\mathbf{S} & ::= & \mathbf{xS} & (1)\\
\mathbf{S} & ::= & \mathbf{yB} & (2)\\
\mathbf{S} & ::= & \mathbf{x} & (3)\\
\mathbf{B} & ::= & \mathbf{yB} & (4)\\
\mathbf{B} & ::= & \mathbf{y} \quad \} & (5)
\end{array}
$$

Ist die Zeichenkette **xxyyy** Element der durch T, N, S, P beschriebenen Sprache?

$$\mathbf{S} \rightarrow \mathbf{xS} \rightarrow \mathbf{xxS} \rightarrow \mathbf{xxyB} \rightarrow \mathbf{xxyyB} \rightarrow \mathbf{xxyyy}$$

➥Ja, da sie sich aus S herleiten läßt.

BNF Beispiel 2

TECHNISCHE UNIVERSITÄT DARMSTADT

$T = \{\mathbf{x}, \mathbf{y}\}$, $N = \{\mathbf{S}, \mathbf{B}\}$, $S = \mathbf{S}$, $P = \{$

$$\begin{aligned}
\mathbf{S} &::= \mathbf{xS} && (6)\\
\mathbf{S} &::= \mathbf{yB} && (7)\\
\mathbf{S} &::= \mathbf{x} && (8)\\
\mathbf{B} &::= \mathbf{yB} && (9)\\
\mathbf{B} &::= \mathbf{y} \quad \} && (10)
\end{aligned}$$

Ist die Zeichenkette **xy** Element der durch T, N, S, P beschriebenen Sprache?

$$\mathbf{S} \rightarrow \mathbf{xS} \rightarrow \mathbf{xyB} \rightarrow ?$$

➥Nein, da sie sich nicht aus S herleiten läßt.

Mehrdeutigkeit in BNF

TECHNISCHE UNIVERSITÄT DARMSTADT

Gegeben seien die Produktionen:

$$\mathbf{S} \quad ::= \quad \mathbf{S{+}S} \tag{11}$$

$$\mathbf{S} \quad ::= \quad \mathbf{x} \tag{12}$$

Wie läßt sich die Zeichenkette **x+x+x** herleiten?

$$\mathbf{S} \rightarrow \mathbf{S{+}S} \rightarrow \mathbf{x{+}S} \rightarrow \mathbf{x{+}S{+}S} \rightarrow \mathbf{x{+}x{+}S} \rightarrow \mathbf{x{+}x{+}x}$$

Aber auch anders:

$$\mathbf{S} \rightarrow \mathbf{S{+}S} \rightarrow \mathbf{S{+}x} \rightarrow \mathbf{S{+}S{+}x} \rightarrow \mathbf{S{+}x{+}x} \rightarrow \mathbf{x{+}x{+}x}$$

Eindeutige CFG

TECHNISCHE UNIVERSITÄT DARMSTADT

Für sinnvolle praktische Anwendungen müssen CFGs eindeutig sein.

Eindeutige Produktionen für die gleiche CFG:

$$\mathbf{S} \quad ::= \quad \mathbf{x+S} \tag{13}$$

$$\mathbf{S} \quad ::= \quad \mathbf{x} \tag{14}$$

TECHNISCHE UNIVERSITÄT DARMSTADT

(Mini-) Triangle

(Mini-) Triangle

TECHNISCHE UNIVERSITÄT DARMSTADT

- Pascal-artige Sprache als Anschauungsobjekt
- Compiler-Quellcode auf Web-Page
- In der Vorlesung: Mini-Triangle
  - Weiter vereinfachte Version
  - Z.B. keine Definition von Unterfunktionen

```
let
   const MAX ~ 10;
   var n: Integer
in begin
   getint(var n);
   if (n>0) /\ (n<=MAX) then
      while n > 0 do begin
         putint(n); puteol();
         n := n-1
      end
   else
end
```

Lokale Deklarationen

Konstante (häßliches “~”!)

Variable kann in `getint` verändert werden

Folge von Anweisungen zwischen `begin`/`end`

`else` ist erforderlich (darf aber leer sein)

Produktionen der Mini-Triangle CFG 1

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Program ::= single-Command
single-Command ::= empty
        | V-name := Expression
        | Identifier ( Expression )
        | if Expression then single-Command
                          else single-Command
        | while Expression do single-Command
        | let Declaration in single-Command
        | begin Command end
Command ::= single-Command
          | Command ; single-Command
...
```

Produktionen der Mini-Triangle CFG 2

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Expression
  ::= primary-Expression
    | Expression Operator primary-Expression
primary-Expression
  ::= Integer-Literal
    | V-name
    | Operator primary-Expression
    | ( Expression )
V-name ::= Identifier
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
Integer-Literal ::= Digit
                  | Integer-Literal Digit
Operator ::= + | - | * | / | < | > | =
```

Produktionen der Mini-Triangle CFG 3

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Declaration
   ::= single-Declaration
     | Declaration ; single-Declaration
single-Declaration
   ::= const Identifier ~ Expression
     | var Identifier : Type-denoter
Type-denoter ::= Identifier
```

Produktionen der Mini-Triangle CFG 4

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Comment ::= ! CommentLine eol
CommentLine ::= Graphic CommentLine
Graphic ::= any printable character or space
Digit  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Terminologie

TECHNISCHE UNIVERSITÄT DARMSTADT

Phrase Von einem gegebenen Nicht-Terminalsymbol herleitbare Kette von Terminalsymbolen.
Z.B. Expression-Phrase, Command-Phrase ...

Satz *S*-Phrase, wobei *S* das Startsymbol der CFG ist

Beispiel:

```
let                     (1)
  var y : Integer       (2)
in                      (3)
  y := y + 1            (4)
```

- Das gesamte Program ist ein *Satz* der CFG
- Zeile 2 ist eine single-Declaration-Phrase
- Zeile 4 ist eine single-Command-Phrase

Syntaxbäume

TECHNISCHE UNIVERSITÄT DARMSTADT

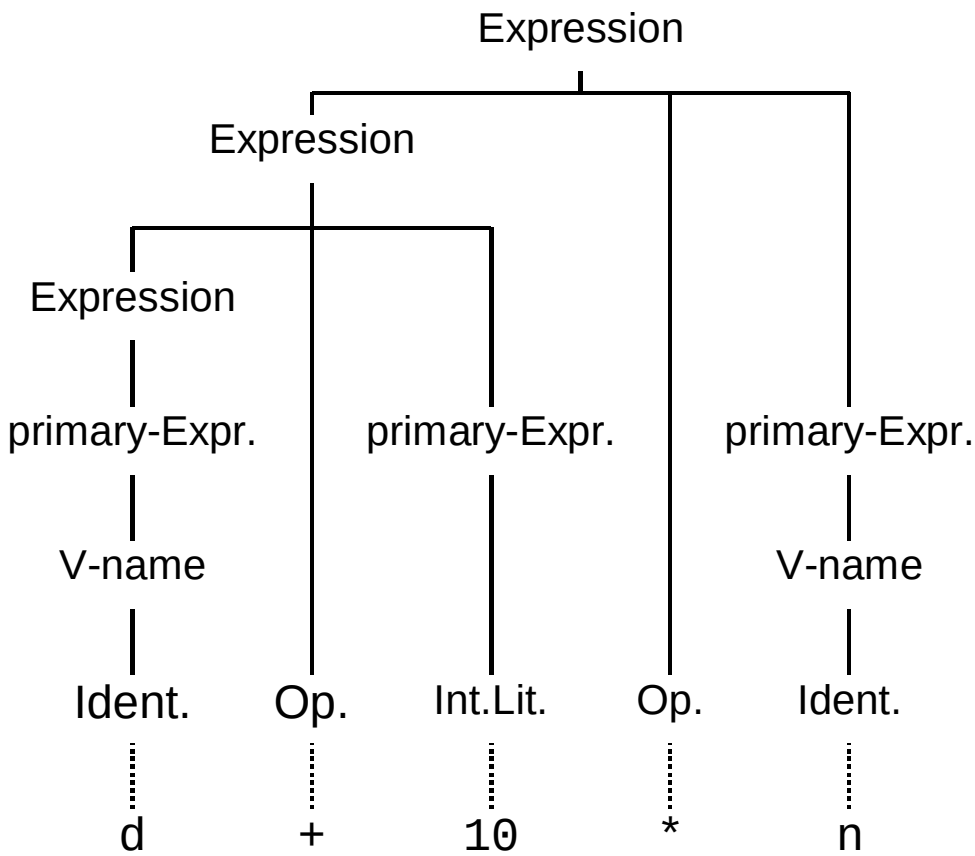
Ein Syntaxbaum ist ein geordneter, markierter Baum bei dem

- ...die Blätter mit Terminalsymbolen markiert sind
- ...die inneren Knoten mit Nicht-Terminalsymbolen markiert sind
- ...jeder innere Knoten **N** (von links nach rechts) die Kinder $\mathbf{X}_1, \ldots, \mathbf{X}_n$ hat, entsprechend der Produktion $\mathbf{N} := \mathbf{X}_1 \ldots \mathbf{X}_n$

Ein *N*-Baum ist ein Baum mit einem *N* Nicht-Terminalsymbol am Wurzelknoten.

Expression-**Baum für** `d + 10 * n`

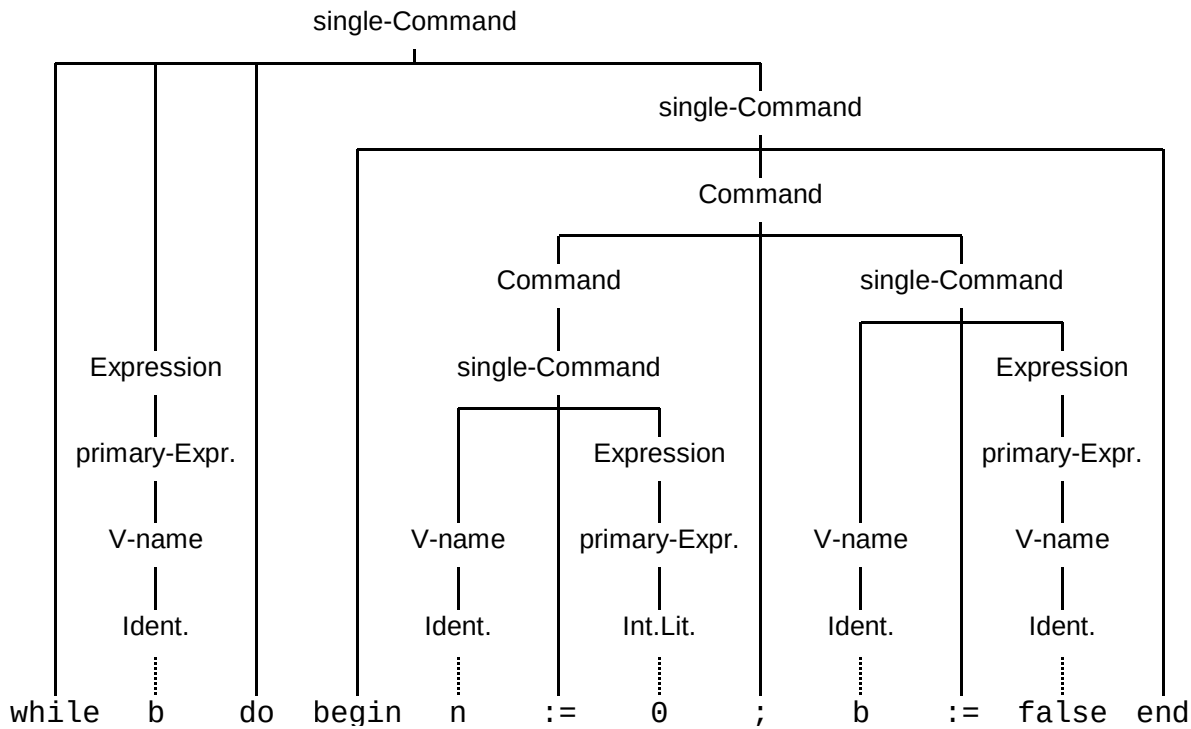
TECHNISCHE UNIVERSITÄT DARMSTADT



single-Command**-Baum**

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while b do begin
  n := 0;
  b := false
end
```



Konkrete und abstrakte Syntax

TECHNISCHE UNIVERSITÄT DARMSTADT

- Grammatik spezifiziert präzise syntaktische Details
  - **`do`**, **`:=`**, ...

➥Konkrete Syntax, wichtig für das Verfassen korrekter Programme

- Konkrete Syntax hat aber *keinen* Einfluß auf Semantik der Programme
  - **`V = E`**
  - **`V := E`**
  - **`set V = E`**
  - **`assign E to V`**
  - **`V` ← `E`**
- ...können alle das gleiche bedeuten: Eine Zuweisung von **E** nach **V**

➥Für weitere Verarbeitung Darstellung vereinfachen!

Abstrakte Syntax 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Modelliert nur essentielle Information
- Idee: Orientierung an der Subphrasen-Struktur der Produktionen
- Beispiel: **`V-name := Expression`** hat zwei Subphrasen
  1. **`V-name`**
  2. **`Expression`**

Abstrakte Syntax 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Schlüsselworte, Begrenzer wie **`do`**, **`:=`** sind irrelevant
- Unterscheidungen zwischen
  - **`Command`** und **`single-Command`**
  - **`Declaration`** und **`single-Declaration`**
  - **`Expression`** und **`primary-Expression`**
- . . . ... sind nur für das Erkennen des Programmes relevant, nicht zur Darstellung seiner Semantik.

➥Alle dafür unwichtigen Details weglassen!

Auszug aus der abstrakten Syntax 1

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Command
  ::= V-name := Expression                    AssignCmd
    | Identifier ( Expression )               CallCmd
    | if Expression then Command
                    else Command              IfCmd
    | while Expression do Command             WhileCmd
    | let Declaration in Command              LetCmd
    | Command ; Command                       SequentialCmd
```

Auszug aus der abstrakten Syntax 2

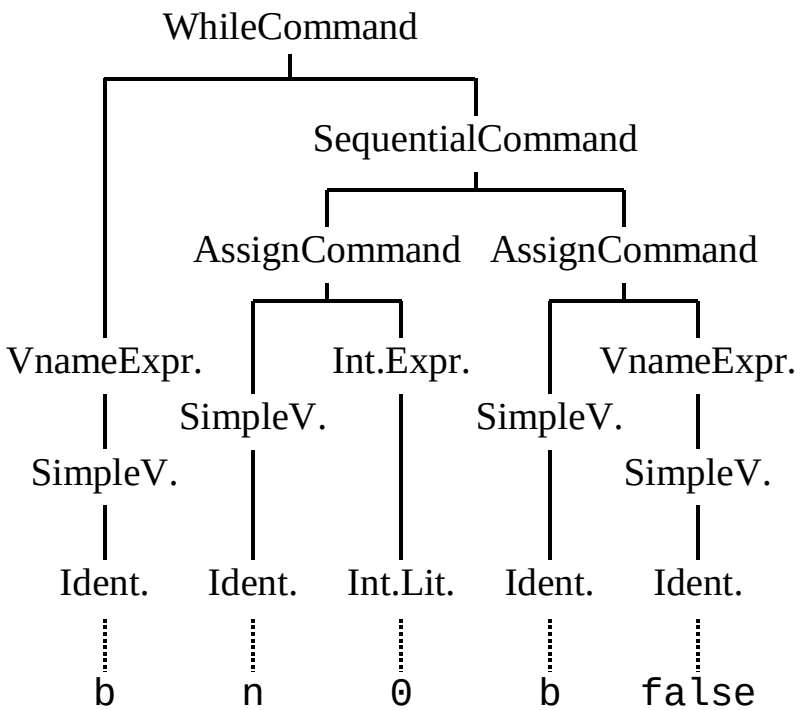
TECHNISCHE UNIVERSITÄT DARMSTADT

```
Expression                                          
   ::= Integer-Literal                              IntegerExp
     | V-name                                       VnameExp
     | Operator Expression                          UnaryExp
     | Expression Op Expression                     BinaryExp
V-name::= Identifier                                SimpleVName
```

Beispiel abstrakte Syntax

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while b do begin
  n := 0;
  b := false
end
```



AST als Zwischendarstellung 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- AST ist eine weit verbreitete Form der IR
- High-level IR
- Sehr nah an der Eingabesprache
- Gut geeignet für weitreichende Analysen und Transformationen
  - Unabhängig von Architektur der Zielmaschine
  - Verschieben von Anweisungen
  - Änderungen der Programmstruktur

AST als Zwischendarstellung 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Schlechter geeignet für maschinennahe Analysen und Transformationen

- Ausnutzung von Maschinenregistern
- Ausnutzung von speziellen Maschinenbefehlsfolgen

➥Hier: Konzentration auf maschinenunabhängige Ebene

- (D)AST ist Hauptrepräsentation
- Für einzelne Bearbeitungsschritte: Andere IRs

TECHNISCHE UNIVERSITÄT DARMSTADT

Kontextuelle Einschränkungen

Kontextuelle Einschränkungen: Geltungsbereiche

TECHNISCHE UNIVERSITÄT DARMSTADT

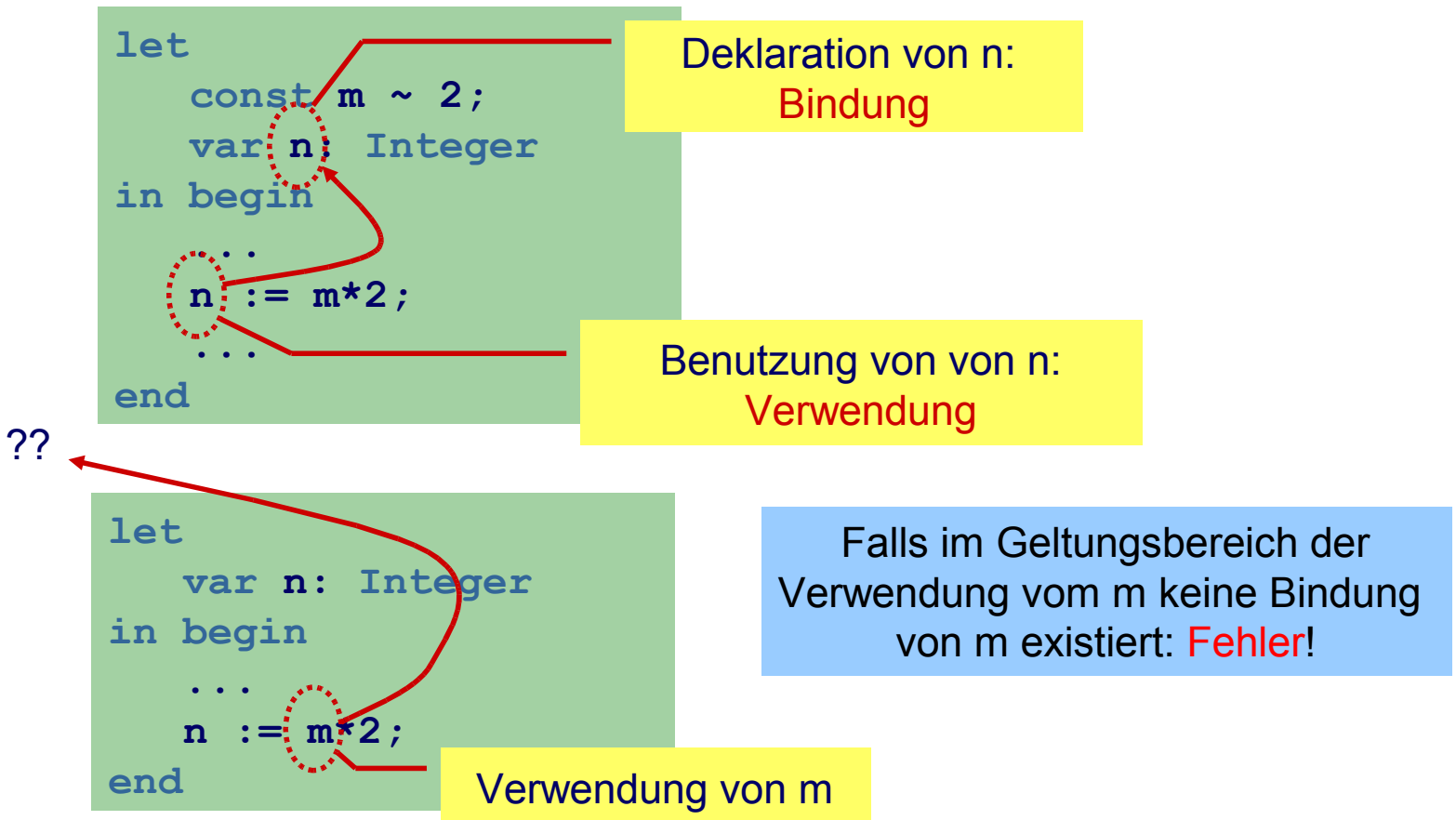
Syntaktische Korrektheit reicht nicht aus für sinnvolle Übersetzung

Geltungsbereiche (Scope)

- Betreffen *Sichtbarkeit* von Bezeichern
- Jeder verwendete Bezeichner muss vorher *deklariert* werden
  - . . . nicht bei allen Programmiersprachen
- Deklaration ist sog. *bindendes Auftreten* des Bezeichners
- Benutzung ist sog. *verwendendes Auftreten* des Bezeichners
- Aufgabe: Bringe jede Verwendung mit genau der einen passenden Bindung in Zusammenhang

Beispiele Geltungsbereiche

TECHNISCHE UNIVERSITÄT DARMSTADT



Kontextuelle Einschränkungen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

Typen

- Jeder Wert hat einen Typ
- Jede Operation
  - . . . hat Anforderungen an die Typen der Operanden
  - . . . hat Regeln für den Typ des Ergebnisses

. . . auch nicht bei allen Programmiersprachen.

- Hier: statische Typisierung (zur Compile-Zeit)
- Alternativ: dynamische Typisierung (zur Laufzeit)

Beispiele Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
let
   var n: Integer
in begin
   ...
   while n > 0 do
      n := n-1;
   ...
end
```

Typregel für $E_1 > E_2$ (GreaterOp):
Wenn E_1 und E_2 beide vom Typ int, dann ist Ergebnis vom Typ bool.

Typregel für **while** E **do** C (WhileCmd):
E muss vom Typ boolean sein.

Typregel für $V := E$ (AssignCmd):
Die Typen von V und E müssen äquivalent sein.

Typregel für $E_1 - E_2$ (SubOp):
Wenn E_1 und E_2 beide vom Typ int, dann ist Ergebnis vom Typ int.

TECHNISCHE UNIVERSITÄT DARMSTADT

Semantik

Semantik 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Semantik beschreibt die Bedeutung eines Programmes zur Ausführungszeit.
Allgemeine Terminologie:

Anweisungen

. . . werden ausgeführt. Mögliche Seiteneffekte:

- Ändern der Werte von Variablen
- Ein-/Ausgabeoperationen

Semantik 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Ausdrücke

. . . werden ausgewertet (evaluiert), um ein Ergebnis zu erhalten.

- Die Evaluation kann in einigen Sprachen auch Seiteneffekte haben.

Deklarationen

. . . werden elaboriert um eine Bindung vorzunehmen. Mögliche Seiteneffekte:

- Allokieren von Speicherplatz
- Initialisieren von Speicherplatz

Beispiele Semantik von (Mini)Triangle 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Beschreibung orientiert sich am AST.

AssignCmd `V := E`

1. Der Ausdruck `E` wird evaluiert um einen Wert v zu erhalten
2. v wird an die Variable `V` zugewiesen

BinaryExp $\texttt{E}_1$ `op` $\texttt{E}_2$

1. Der Ausdruck $\texttt{E}_1$ wird evaluiert um einen Wert v_1 zu erhalten
2. Der Ausdruck $\texttt{E}_2$ wird evaluiert um einen Wert v_2 zu erhalten
3. Die Werte v_1 und v_2 werden mit dem Operator `op` zu einem Wert v_3 verknüpft.
4. v_3 ist das Ergebnis der BinaryExp

Beispiele Semantik von (Mini)Triangle 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Declaration `var I : T`

1. Der Bezeichner `I` wird an eine Variable vom Typ `T` gebunden
2. Es wird ein für `T` passender Speicherbereich bereitgestellt
3. Der Speicherbereich ist *nicht* initialisiert
4. Der Geltungsbereich für `I` ist der eingeschlossene Block (LetCmd)
5. Am Ende des Blockes wird die Bindung aufgehoben
6. . . . und der Speicherbereich wieder freigegeben

Triangle

TECHNISCHE UNIVERSITÄT DARMSTADT

- In Vorlesung: Mini-Triangle
  - Stark vereinfacht
  - Z.B. Keine Unterprogramme (Prozeduren/Funktionen)
- Im praktischen Teil: Triangle
  - Pascal-artige Sprache
  - Arrays, Records, Prozeduren, Funktionen
  - Parameterübergabe durch Wert oder Referenz
  - Prozeduren/Funktionen als Parameter erlaubt
  - Ausdrücke haben *keine* Seiteneffekte
- Beschreibung in PLPJ, Anhang B

Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Überblick
- Organisation
- Material in PLPJ, Kapitel 1
  - Syntax (konkrete und abstrakte)
  - Kontextuelle Einschränkungen
  - Semantik
  - AST als IR
  - (Mini-)Triangle