Optimierende Compiler

Einleitung

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2010

Compiler

- Schnittstelle zwischen
  - Programmiersprache
  - Maschine

Programmiersprache Gut für Menschen handhabbar

- Smalltalk
- Java
- C++

Maschine Getrimmt auf

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

Compiler

- Schnittstelle zwischen
  - Programmiersprache
  - Maschine

Programmiersprache Gut für Menschen handhabbar

- Smalltalk
- Java
- C++

Maschine Getrimmt auf

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

Compiler

- 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

Entscheidet über dem Benutzer zugängliche Rechenleistung

Beispiel: Bildkompression auf Dothan CPU, 2GHz

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

Auswirkung von Compilern

Entscheidet über dem Benutzer zugängliche Rechenleistung

Beispiel: Bildkompression auf Dothan CPU, 2GHz

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

Auswirkung von Compilern

Entscheidet über dem Benutzer zugängliche Rechenleistung

Beispiel: Bildkompression auf Dothan CPU, 2GHz

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

Programmiersprachen

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
```

Programmiersprachen

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
```

Programmiersprachen

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

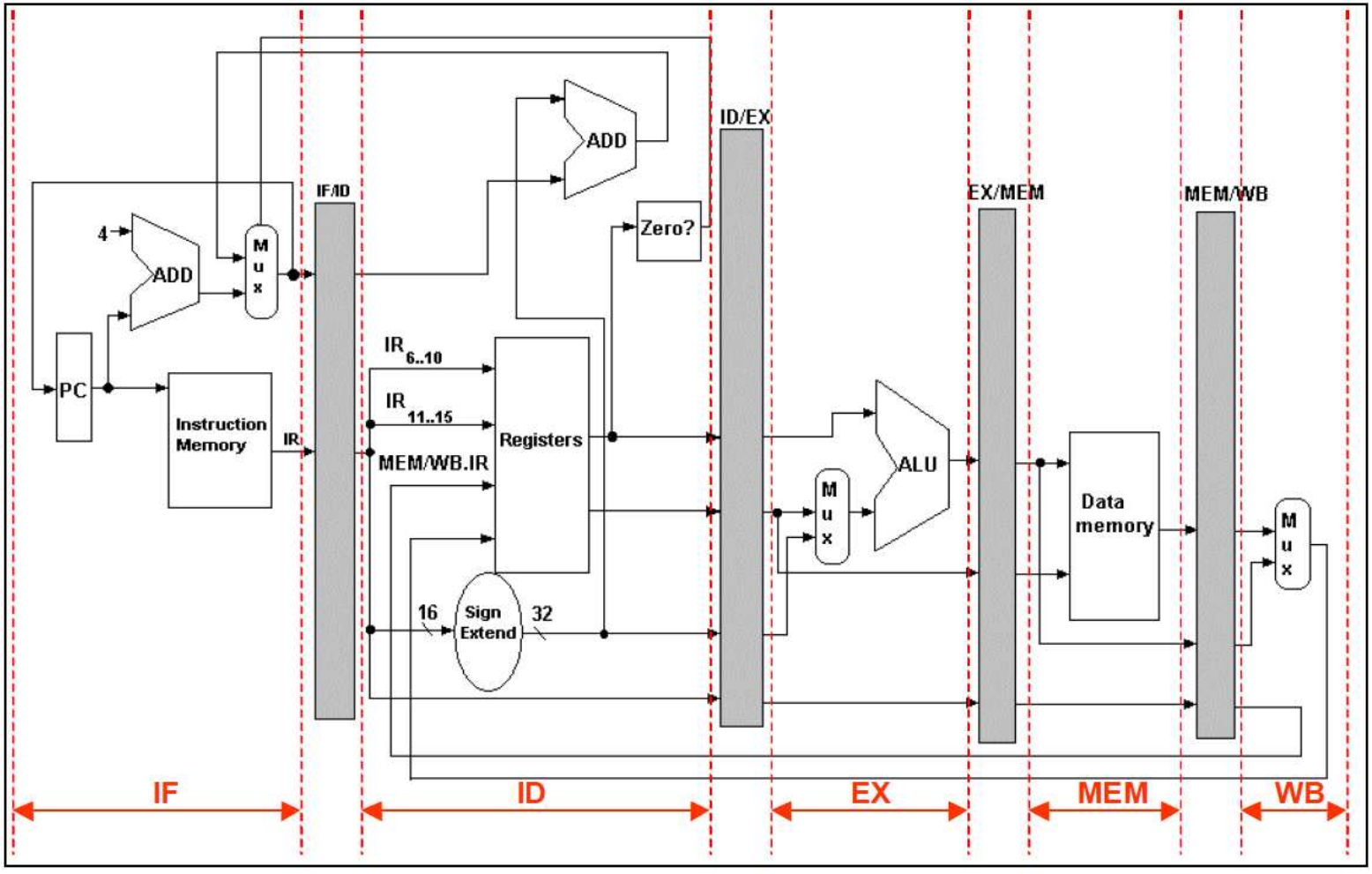
- Auf unteren Ebenen immer feinere Beschreibung
- Immer näher an Zielmaschine (Hardware)
- Details werden von Compiler hinzugefügt
  - Durch verschiedenste Algorithmen
    - Analyse von Programmeigenschaften
    - Verfeinerung der Beschreibung durch Synthese von Details

Unterschiedliche Abstraktionsebenen

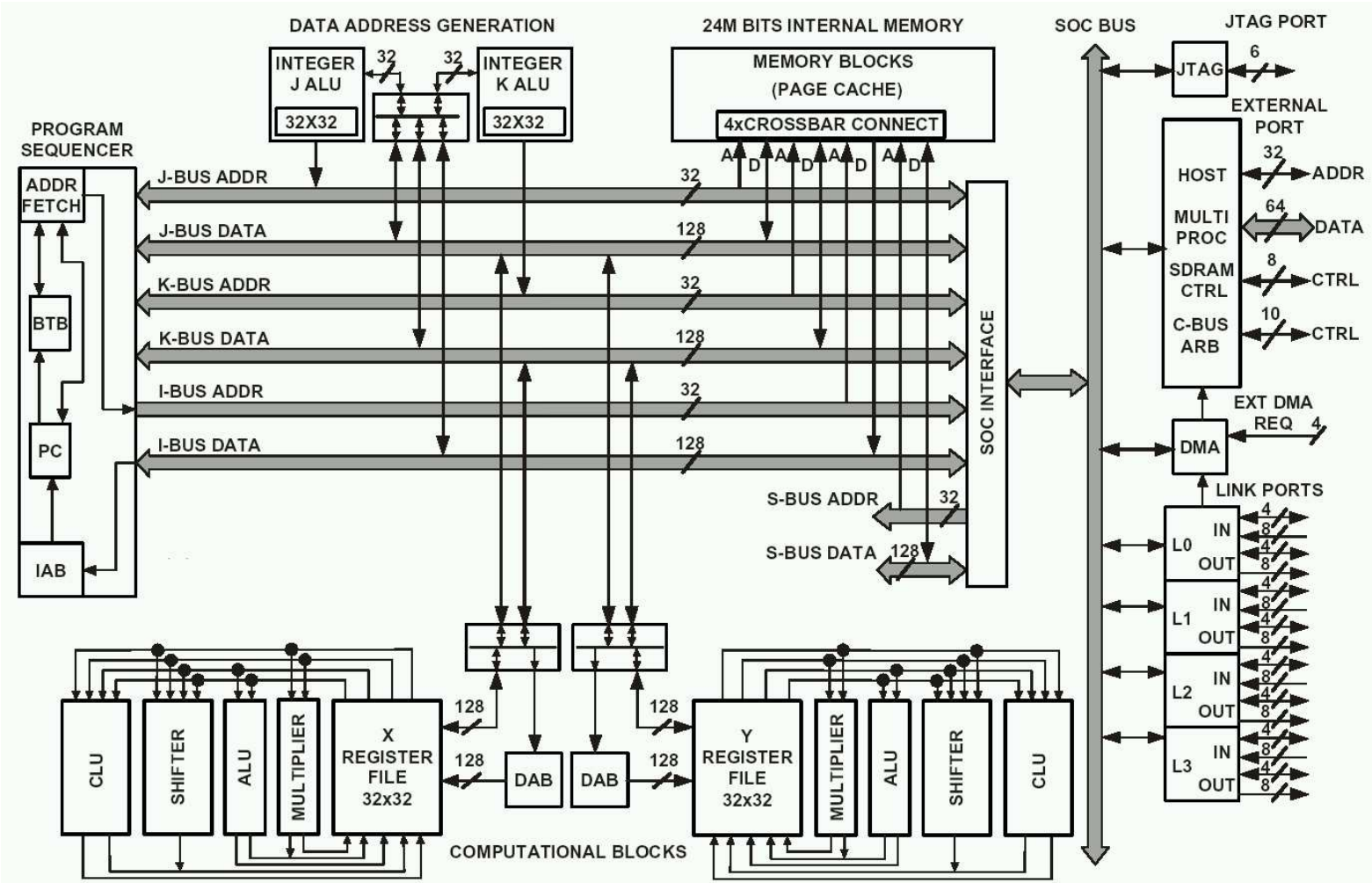
- Auf unteren Ebenen immer feinere Beschreibung
- Immer näher an Zielmaschine (Hardware)
- Details werden von Compiler hinzugefügt
  - Durch verschiedenste Algorithmen
    - Analyse von Programmeigenschaften
    - Verfeinerung der Beschreibung durch Synthese von Details

Auswirkungen der Zielmaschine

Einfach: Hennessy & Patterson DLX

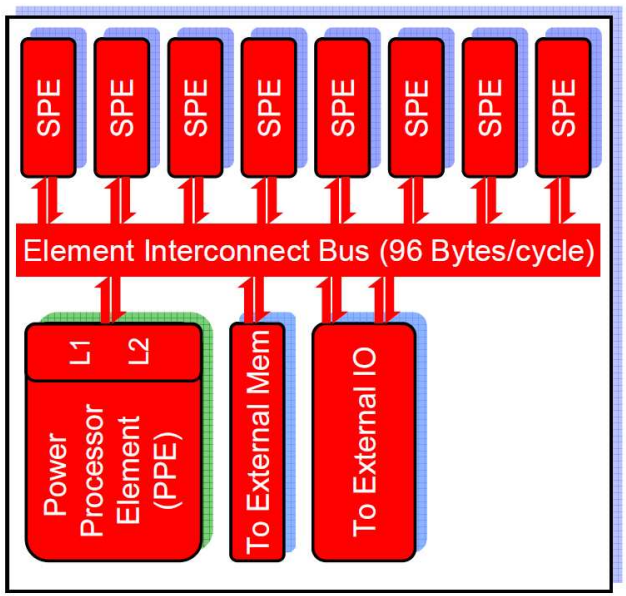


Komplizierter: Analog Devices TigerSHARC

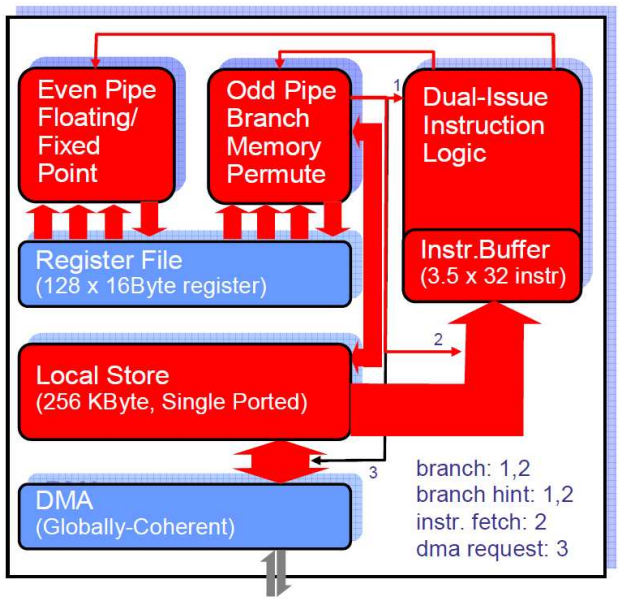


Problematisch: IBM/Sony Cell Processor

Übersicht



SPE



Spezialisierte Anforderungen

- 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**

Spezialisierte Anforderungen

- 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**

Spezialisierte Anforderungen

- 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

DOI:10.1145/1461928.1461946

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

BY MARY HALL, DAVID PADUA, AND KESHAV PINGALI

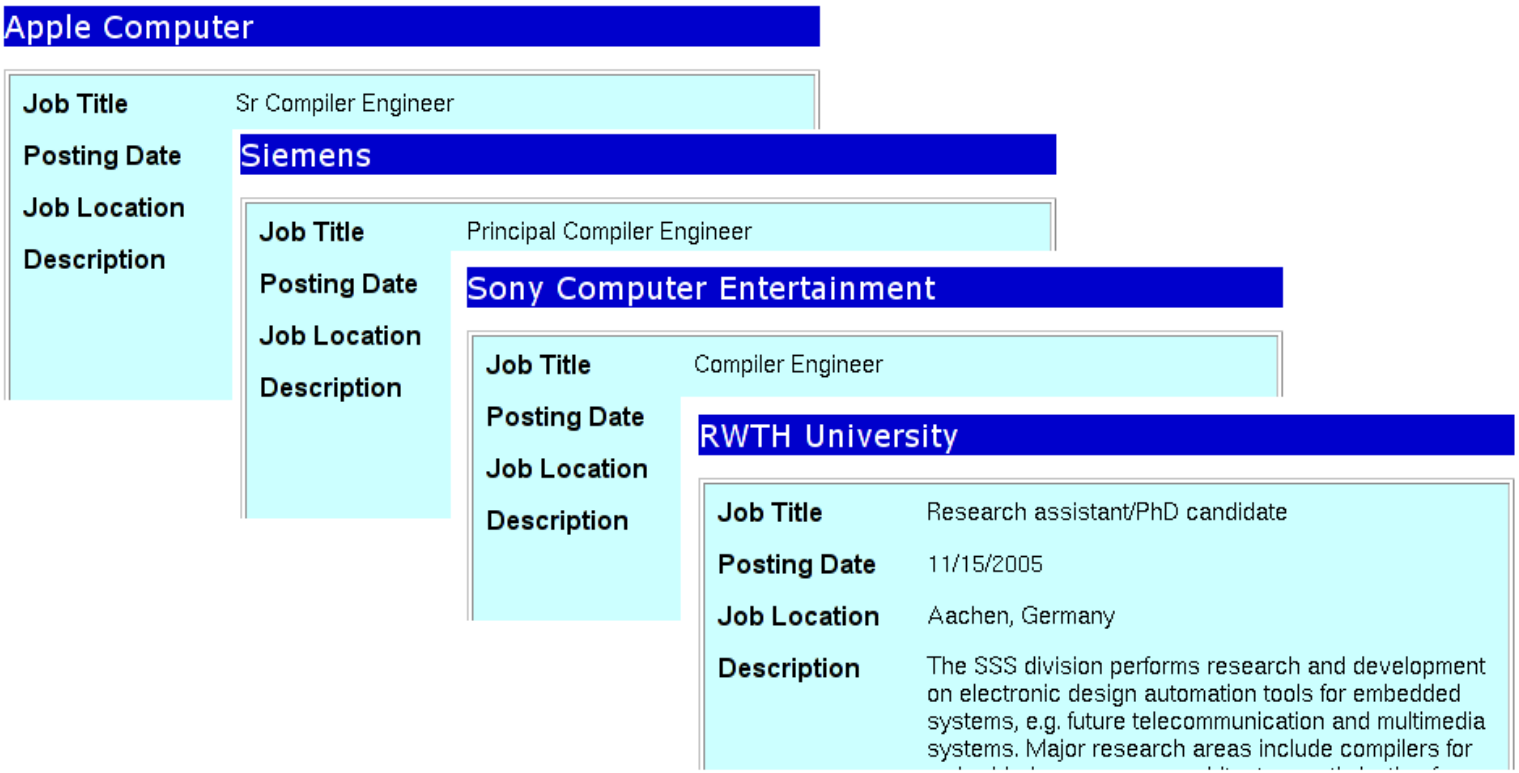
Compiler Research: The Next 50 Years

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

Technologie

- 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!

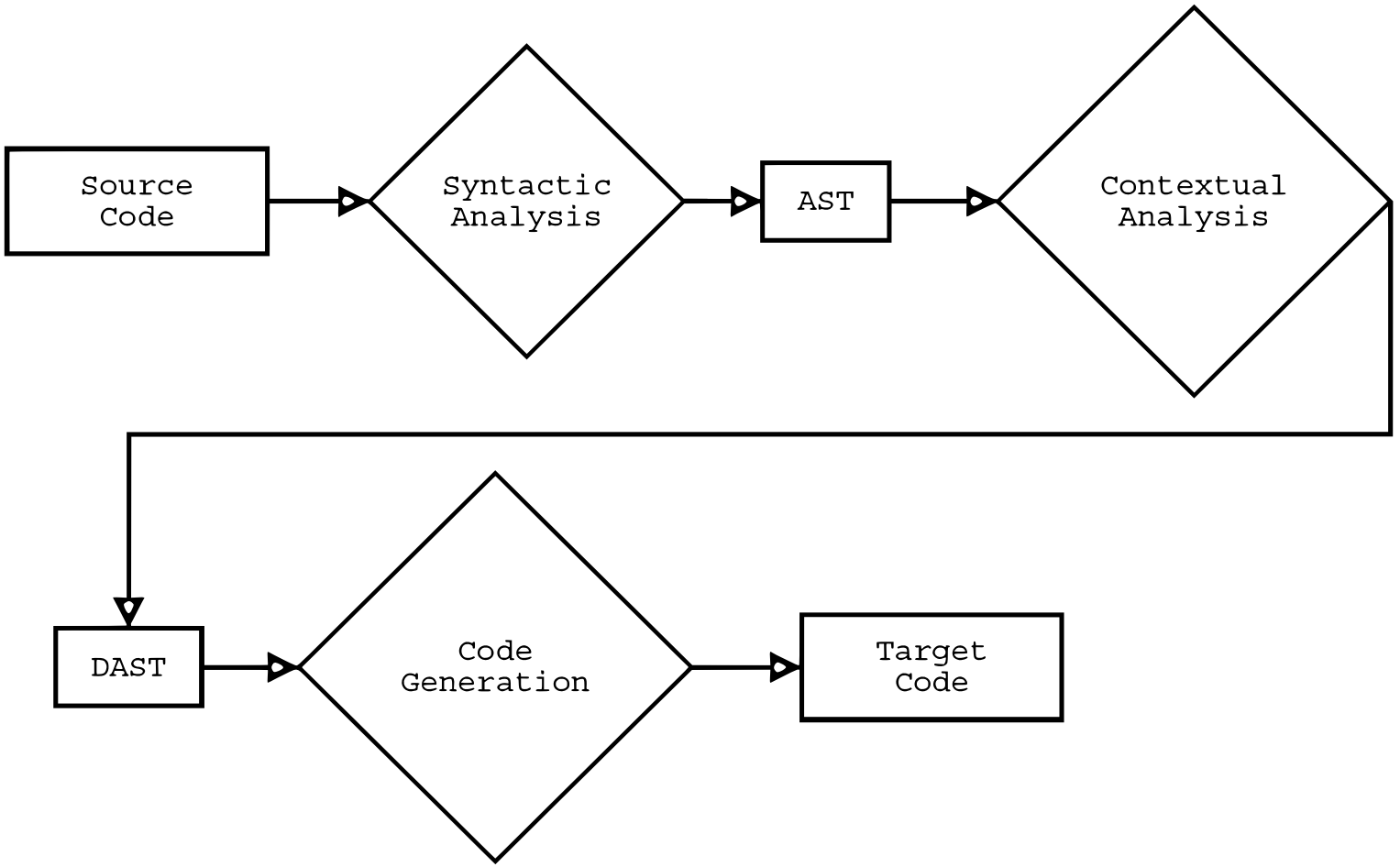
Spezialisten gesucht



Offene Stellen auf **`www.compilerjobs.com`** bis 04/2010: 43

Aufbau von Compilern

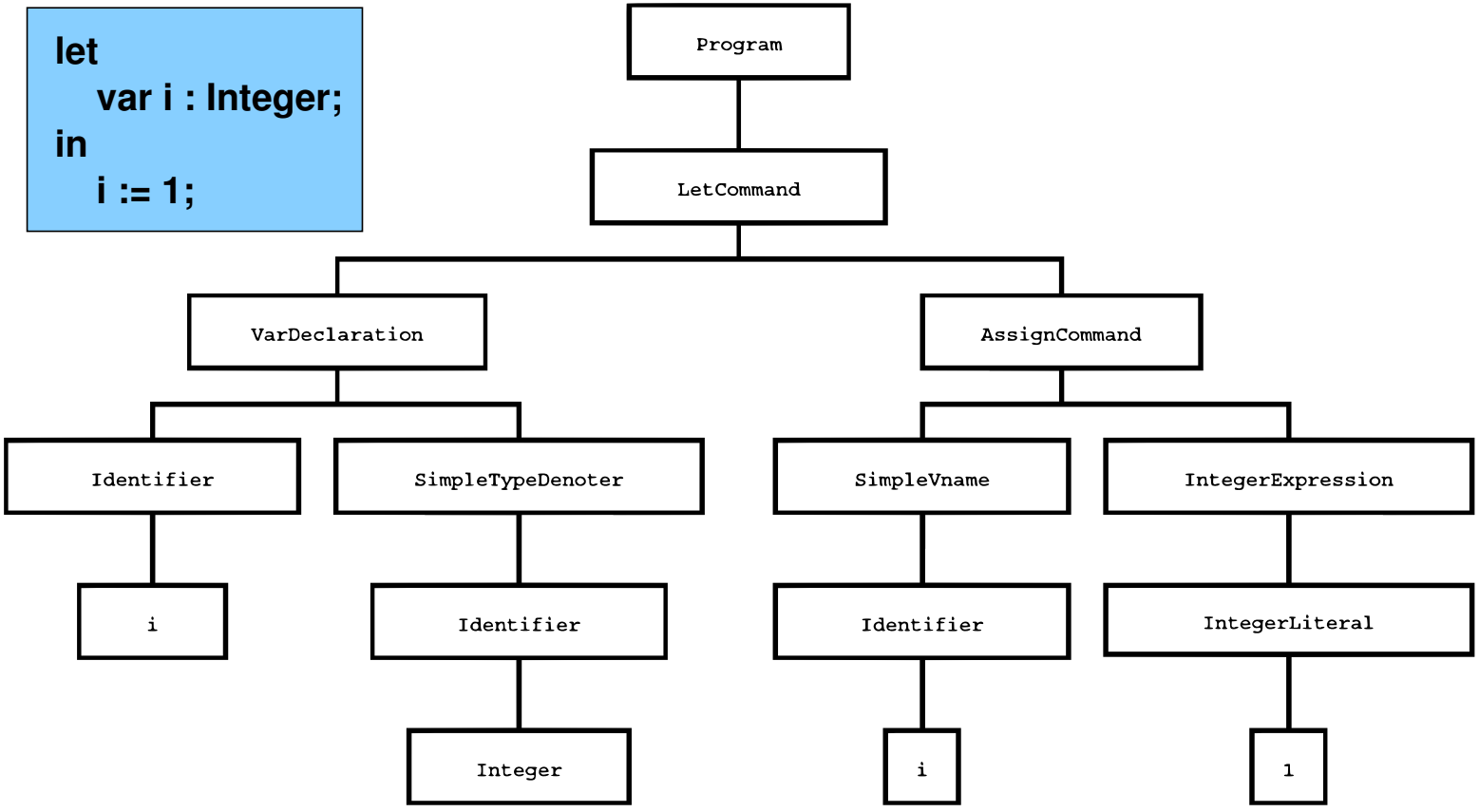
Vorgehen: Bearbeitung in mehreren Phasen



Zwischendarstellung(en) für den Informationsaustausch

Syntaxanalyse

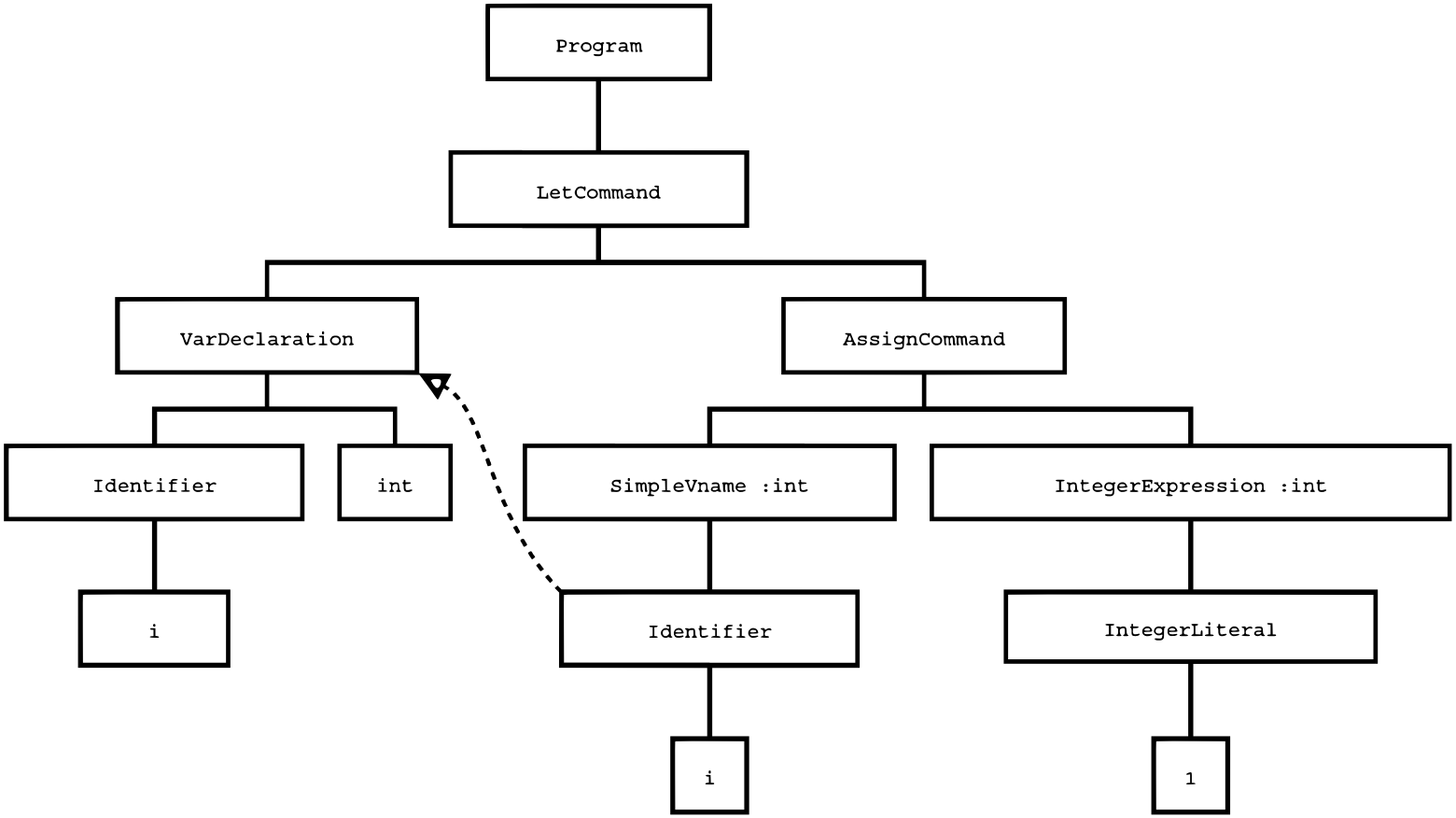
- Überprüfung ob Programm Syntaxregeln gehorcht
- Speichern des Programmes in geeigneter Darstellung



Abstrakter Syntaxbaum

Kontextanalyse - Identifikation

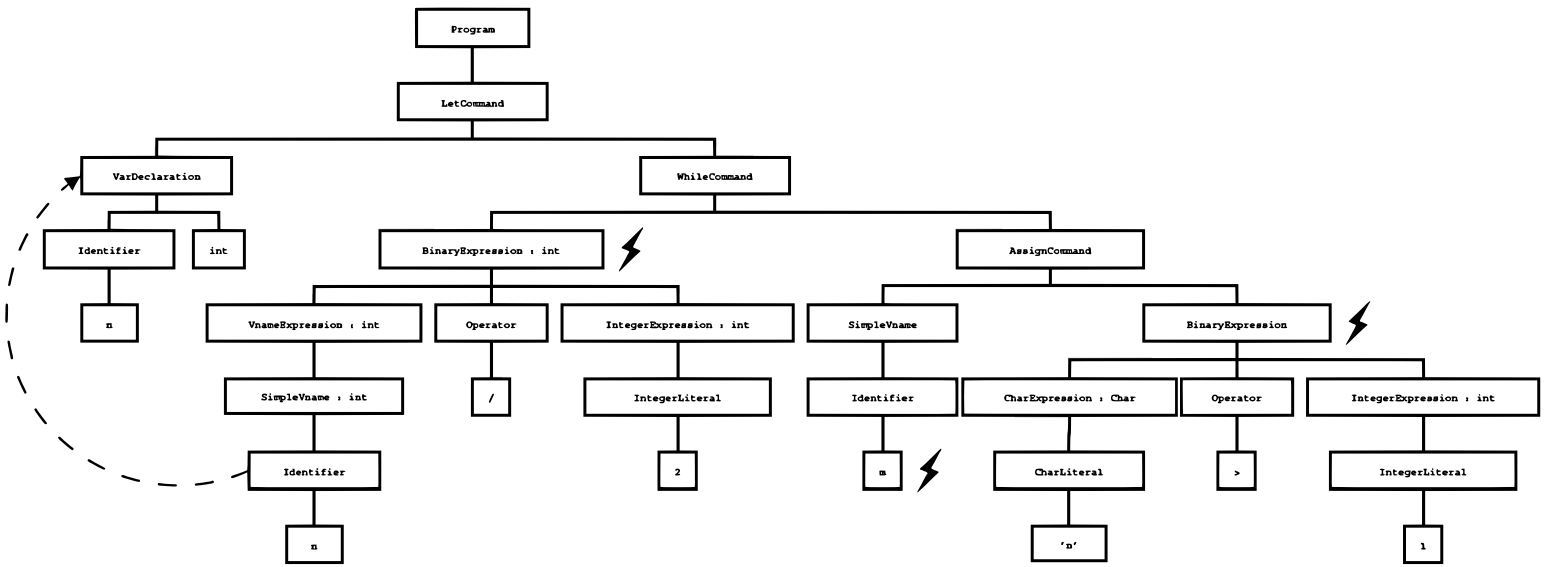
- Ordne Variablen ihren Deklarationen zu
- Berechne Typen von Ausdrücken



DAST: Dekorierter bzw. annotierter AST

Kontextanalyse - Überprüfung

Erkenne Fehler in Variablen- und Typzuordnung



Code-Erzeugung 1

- 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 1

- 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

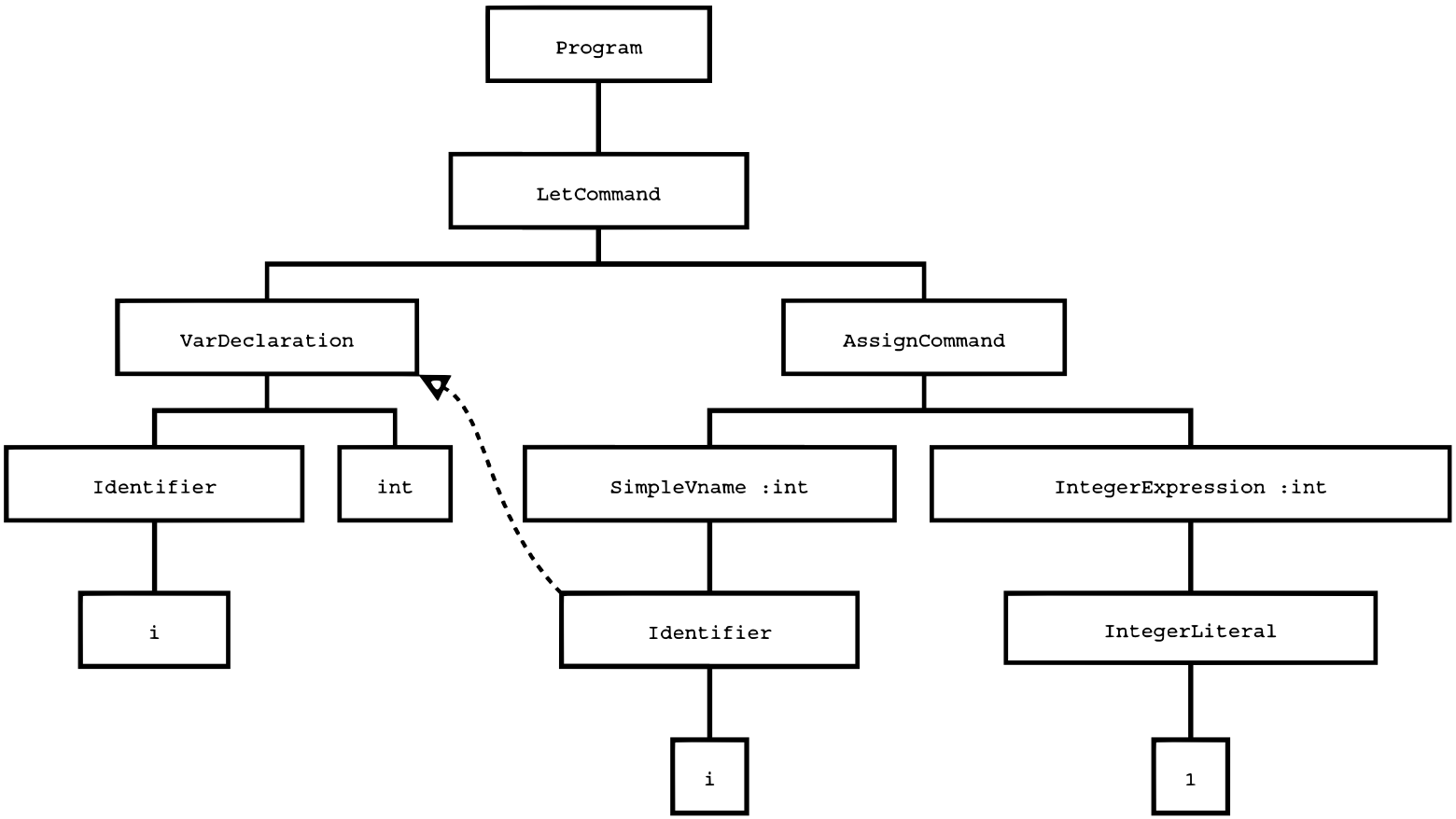
Code-Erzeugung 1

- 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 1

- 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
```

Optimierung

Optimierender Compiler

- 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

Optimierender Compiler

- 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

Optimierender Compiler

- 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

Optimierender Compiler

- 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

Optimierender Compiler

- 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

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;
}
```

Beispiele für Optimierung

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;
}
```

Beispiele für Optimierung

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;
}
```

Beispiele für Optimierung

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

Dozent
Andreas Koch koch@esa.informatik.tu-darmstadt.de
Sprechstunde Mi 14:00-15:00, E103

Web-Seite

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

Material: Grundlage der Vorlesung

Grundlagen von Compilern: Fast vollständig

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

Optimierung: Auszugsweise

Engineering a Compiler
von Keith Cooper und Linda Torczon, Elsevier 2004

Advanced Compiler Design and Implementation
von Steven Muchnick, Morgan-Kaufman 1997

Material: Grundlage der Vorlesung

Grundlagen von Compilern: Fast vollständig

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

Optimierung: Auszugsweise

Engineering a Compiler
von Keith Cooper und Linda Torczon, Elsevier 2004

Advanced Compiler Design and Implementation
von Steven Muchnick, Morgan-Kaufman 1997

Übersichtswerk

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 1
Erstmals in 2010

- Vorlesung
  - Theoretische und algorithmische Grundlagen
  - Semesterbegleitende Prüfung gegen Ende der Vorlesungszeit

- Praktikum
  - Praktische Implementierung
  - Eng verzahnt mit Vorlesung
  - Besuch auch zum Verständnis der Theorie dringend empfohlen

Veranstaltungskonzept 1
Erstmals in 2010

- Vorlesung
  - Theoretische und algorithmische Grundlagen
  - Semesterbegleitende Prüfung gegen Ende der Vorlesungszeit

- Praktikum
  - Praktische Implementierung
  - Eng verzahnt mit Vorlesung
  - Besuch auch zum Verständnis der Theorie dringend empfohlen

Veranstaltungskonzept 2

- Anfangs zweimal die Woche Vorlesungen
- Dann fährt Praktikum hoch
- Vorlesung nun nur noch einmal die Woche

Praktikum

- Erweitern eines bestehenden Compilers
- Einfache Programmiersprache: Triangle
- Zielmaschine: Simulierte Stack-Maschine
- Programmierung überwiegend in Java

Veranstaltungskonzept 2

- Anfangs zweimal die Woche Vorlesungen
- Dann fährt Praktikum hoch
- Vorlesung nun nur noch einmal die Woche

Praktikum

- Erweitern eines bestehenden Compilers
- Einfache Programmiersprache: Triangle
- Zielmaschine: Simulierte Stack-Maschine
- Programmierung überwiegend in Java

Praktikum

- Wird in Dreiergruppen durchgeführt
- Beginnt nach Einführung der Kontextanalyse (geplant KW 18)
- Anmeldung in Vorlesung
- Erste Aufgabe noch recht kurz (ca. 1 Woche Bearbeitungszeit), folgende länger
- Derzeit geplant vier Aufgaben und eine Korrekturphase

Umfang des Praktikums

Programmgrößen aus 2008 in Zeilen Java-Code

6849
15927
17565

Benotung des Praktikums

- Viel Freiheit bei der Realisierung
- Keine starren Bewertungsrichtlinien
  - Analog zu Bachelor- oder Master-Arbeit o.ä.

Grundidee

- Brauchbar kommentierte
- Brauchbar dokumentierte
- im Wesentlichen funktionierende

Lösung der Aufgabenstellung

➥ **Note 2,0**

- Kleinere Schwächen sind akzeptabel
  - Einbußen in Lösungsqualität, Rechenzeit, Speicher, . . .

Grundidee

- Brauchbar kommentierte
- Brauchbar dokumentierte
- im Wesentlichen funktionierende

Lösung der Aufgabenstellung
➥**Note 2,0**

- Kleinere Schwächen sind akzeptabel
  - Einbußen in Lösungsqualität, Rechenzeit, Speicher, . . .

Grundidee

- Brauchbar kommentierte
- Brauchbar dokumentierte
- im Wesentlichen funktionierende

Lösung der Aufgabenstellung
➥**Note 2,0**

- Kleinere Schwächen sind akzeptabel
  - Einbußen in Lösungsqualität, Rechenzeit, Speicher, . . .

Aber Luft nach oben Richtung Note 1,0

Beispielsweise für:

- Sehr gute eigene Algorithmen und Datenstrukturen
- Umfassende Kommentierung und Dokumentation
- Sehr gute Lösungsqualität
- Sehr kurze Rechenzeiten
- Niedriger Speicherverbrauch
- ...

Aufbau der Veranstaltung 1

Überblick über Front-End[1] (ca. 5 Wochen)

- Lexing/Parsing
- Zwischendarstellungem

Überblick über Middle-End (ca. 4 Wochen)

- Analysen
- Optimierungen

Überblick über Back-End[1] (ca. 5 Wochen)

- Laufzeitorganisation
- Code-Erzeugung

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

- IMT3052 von Ivar Farup, Universität Gjøvik, Norwegen
- Vertalerbouw 2004 von Theo Ruys, Universität Twente, Niederlande

Aufbau der Veranstaltung 1

Überblick über Front-End[1] (ca. 5 Wochen)

- Lexing/Parsing
- Zwischendarstellungem

Überblick über Middle-End (ca. 4 Wochen)

- Analysen
- Optimierungen

Überblick über Back-End[1] (ca. 5 Wochen)

- Laufzeitorganisation
- Code-Erzeugung

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

- IMT3052 von Ivar Farup, Universität Gjøvik, Norwegen
- Vertalerbouw 2004 von Theo Ruys, Universität Twente, Niederlande

Aufbau der Veranstaltung 1

Überblick über Front-End[1] (ca. 5 Wochen)

- Lexing/Parsing
- Zwischendarstellungem

Überblick über Middle-End (ca. 4 Wochen)

- Analysen
- Optimierungen

Überblick über Back-End[1] (ca. 5 Wochen)

- Laufzeitorganisation
- Code-Erzeugung

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

- IMT3052 von Ivar Farup, Universität Gjøvik, Norwegen
- Vertalerbouw 2004 von Theo Ruys, Universität Twente, Niederlande

Aufbau der Veranstaltung 1

Überblick über Front-End[1] (ca. 5 Wochen)

- Lexing/Parsing
- Zwischendarstellungem

Überblick über Middle-End (ca. 4 Wochen)

- Analysen
- Optimierungen

Überblick über Back-End[1] (ca. 5 Wochen)

- Laufzeitorganisation
- Code-Erzeugung

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

- IMT3052 von Ivar Farup, Universität Gjøvik, Norwegen
- Vertalerbouw 2004 von Theo Ruys, Universität Twente, Niederlande

Aufbau der Veranstaltung 2

Nicht behandelt werden

- Entwurf von Programmiersprachen
- Übersetzung von objektorientierten Sprachen
- Komplexe Laufzeitsysteme
  - Garbage Collection
  - Just-In-Time Compiler
    ➥VL Virtuelle Maschinen, Dr. Haupt (Mi+Fr)

Syntax

Syntax oder Grammatik
PLPJ, Kapitel 1

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

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

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

Semantik

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

Semantik

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

- 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

Art der Spezifikation

- 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

Art der Spezifikation

- 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

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

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

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

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.

Syntax durch Mengenbeschreibung

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)

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**

Reguläre Ausdrücke (REs)

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**, **abababab**, ...
- $L = \mathbf{a}(\mathbf{b}|\varepsilon)$ ergibt **a** oder **ab**

Mächtigkeit von REs?

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).

Mächtigkeit von REs?

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).

Mächtigkeit von REs?

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).

Mächtigkeit von REs?

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

Eine kontextfreie Grammatik besteht aus

- Einer Menge von Terminalsymbolen T aus dem 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

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

Kontextfreie Grammatiken (CFGs) 2

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

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

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

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 1

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

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

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 1

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

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

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

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

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

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.

BNF Beispiel 2

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

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

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.

BNF Beispiel 2

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

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

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

Gegeben seien die Produktionen:

$$\mathbf{S} ::= \mathbf{S{+}S} \quad (11)$$

$$\mathbf{S} ::= \mathbf{x} \quad (12)$$

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:

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

Mehrdeutigkeit in BNF

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:

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

Mehrdeutigkeit in BNF

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:

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

Mehrdeutigkeit in BNF

Gegeben seien die Produktionen:

$$\mathbf{S} ::= \mathbf{S+S} \quad (11)$$

$$\mathbf{S} ::= \mathbf{x} \quad (12)$$

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

Für sinnvolle praktische Anwendungen müssen CFGs eindeutig sein.

Eindeutige Produktionen für die gleiche CFG:

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

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

(Mini-) Triangle

(Mini-) Triangle

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

Lokale Deklarationen

```
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
```

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

```
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

```
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

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

Produktionen der Mini-Triangle CFG 4

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

Terminologie

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

Terminologie

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

Terminologie

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

Terminologie

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

Terminologie

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

Terminologie

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

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.

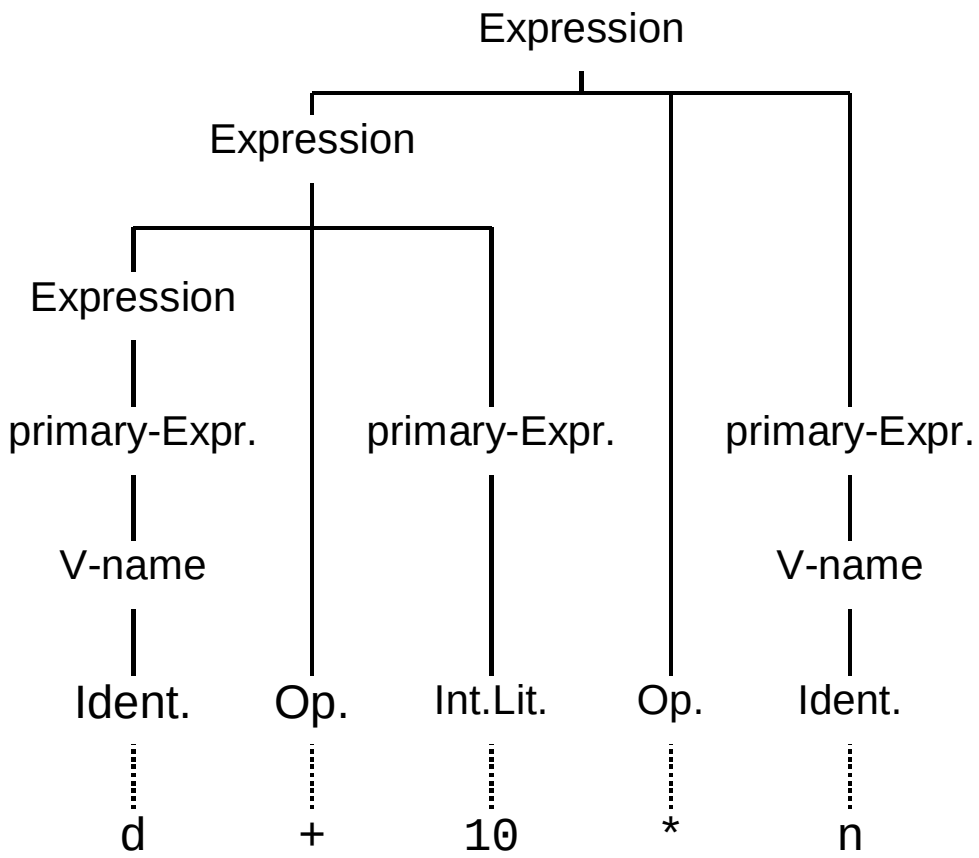
Syntaxbäume

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 $\mathbf{N} := \mathbf{X}_1 \ldots \mathbf{X}_n$

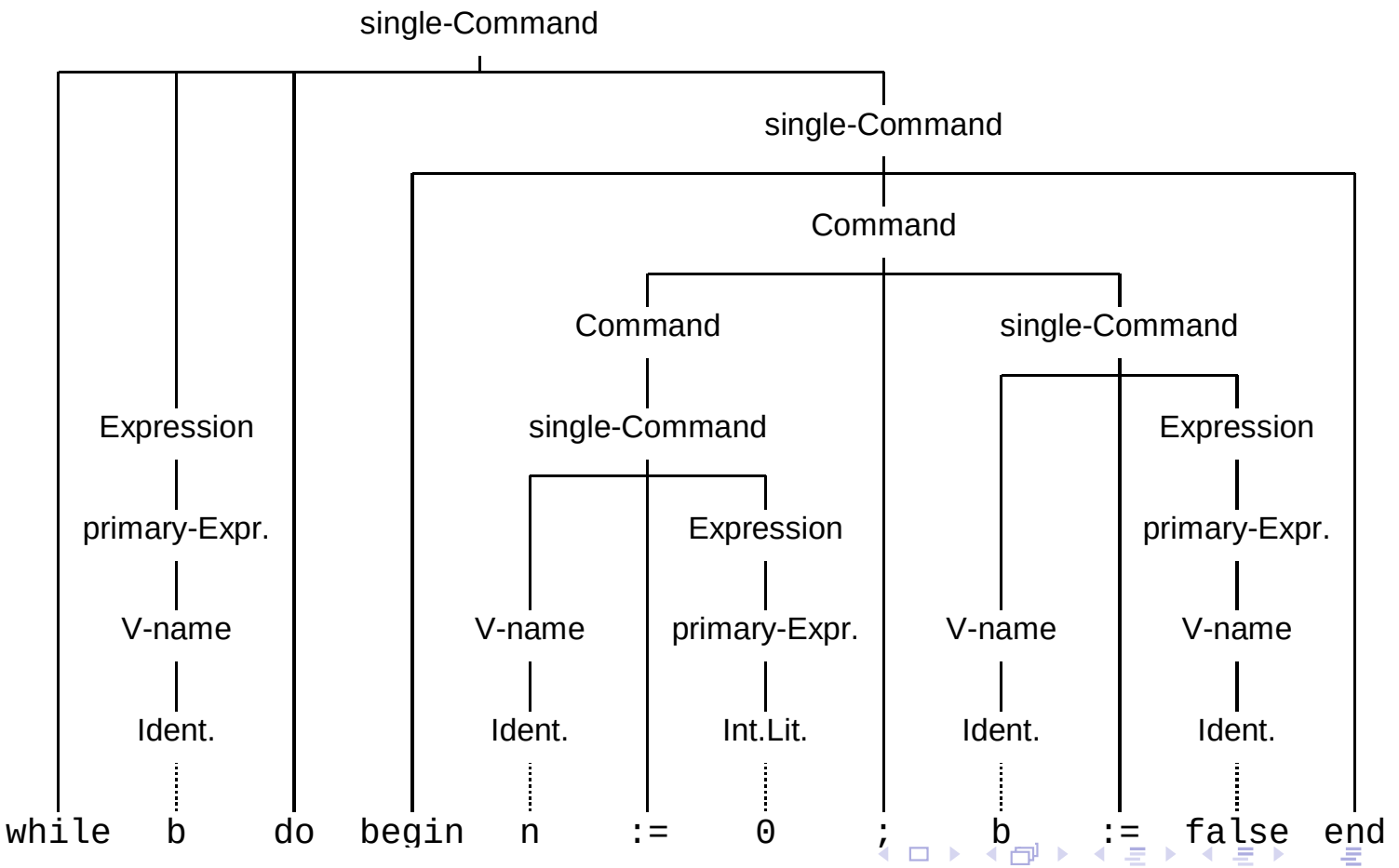
Ein N-Baum ist ein Baum mit einem N Nicht-Terminalsymbol am Wurzelknoten.

Expression-Baum für `d + 10 * n`



single-Command-Baum

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



Konkrete und abstrakte Syntax

- 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!

Konkrete und abstrakte Syntax

- 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!

Konkrete und abstrakte Syntax

- 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

- 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

- 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!

Abstrakte Syntax 2

- 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

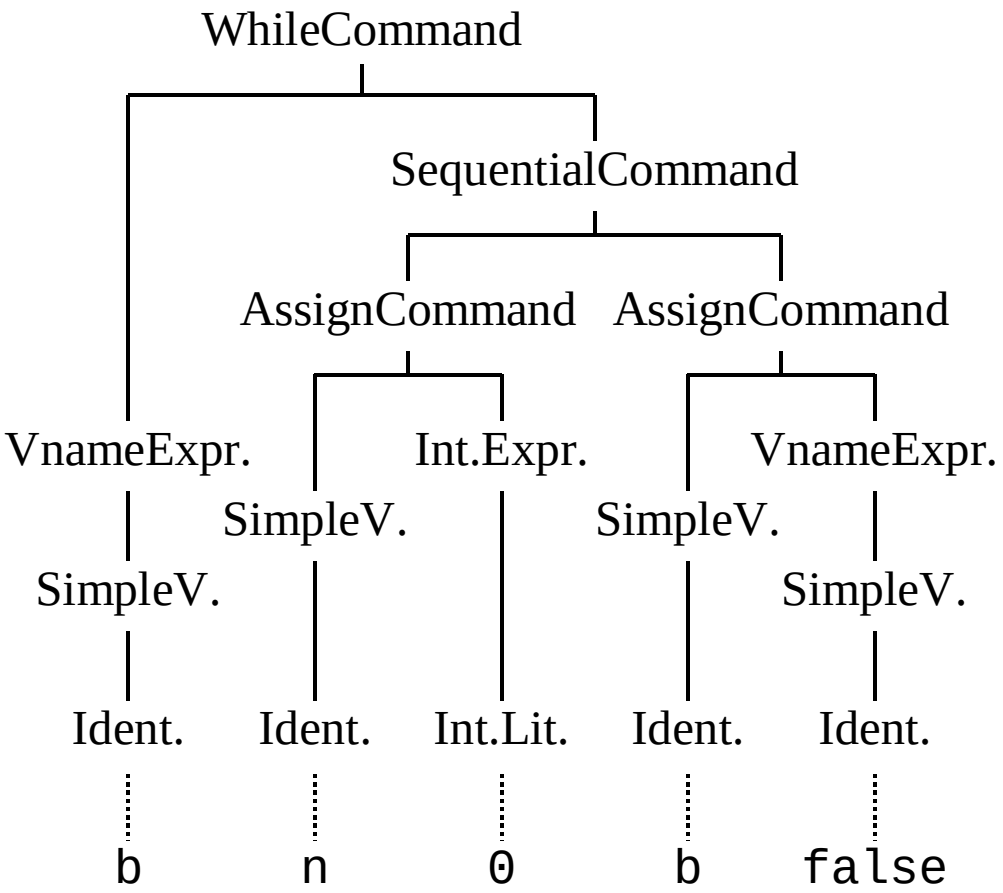

```
Command
 ::= V-name := Expression                    AssignCmd
   | Identifier ( Expression )               CallCmd
   | if Expression then Command
                       else Command          IfCmd
   | while Expression do Command             WhileCmd
   | let Declaration in Command            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

Auszug aus der abstrakten Syntax 2


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

Beispiel abstrakte Syntax

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



AST als Zwischendarstellung 1

- AST ist eine weit verbreite 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

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

AST als Zwischendarstellung 2

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

Kontextuelle Einschränkungen

Kontextuelle Einschränkungen: Geltungsbereiche

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

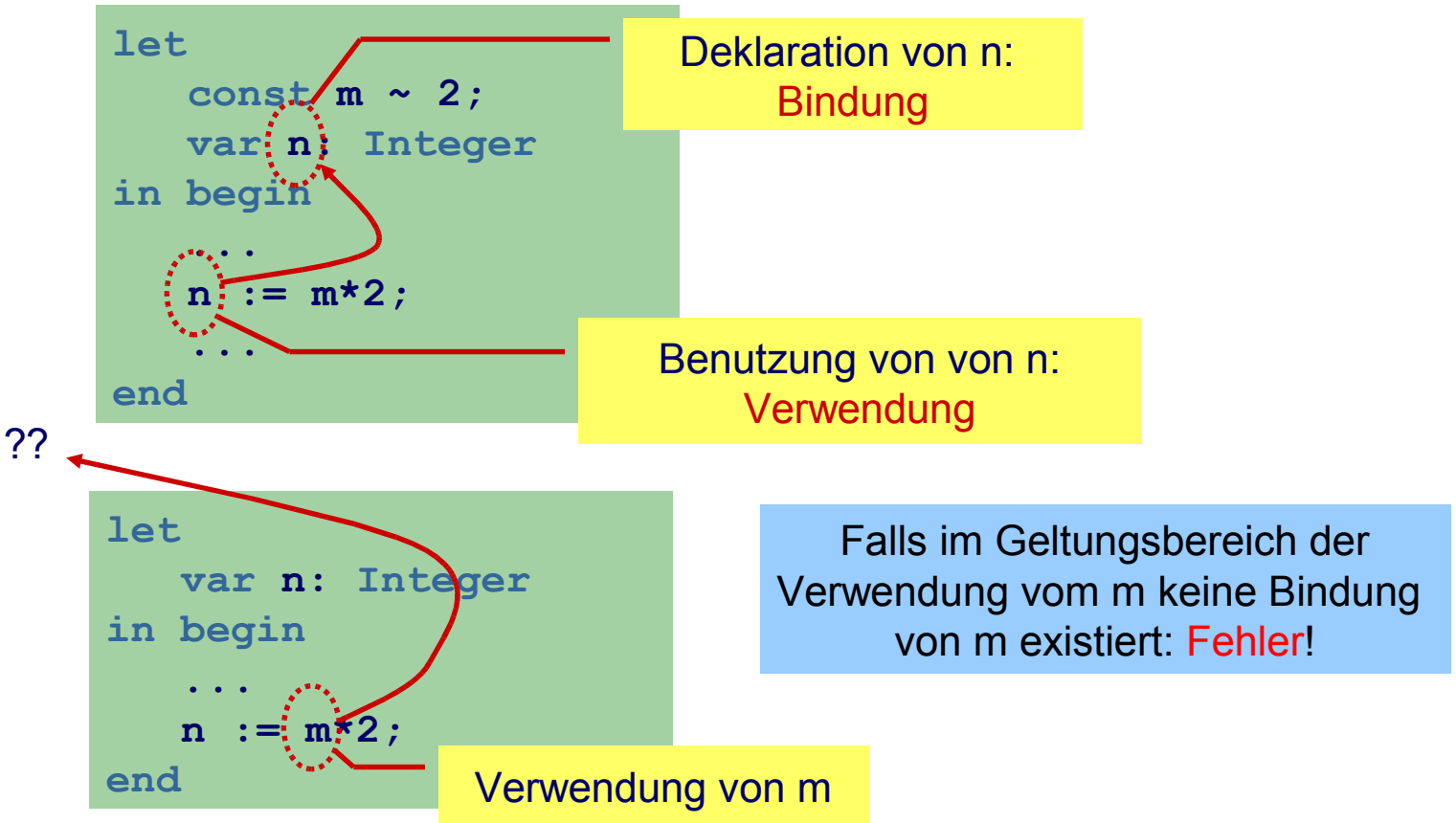
Kontextuelle Einschränkungen: Geltungsbereiche

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



Kontextuelle Einschränkungen: Typen

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)

Kontextuelle Einschränkungen: Typen

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


```
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.

Semantik

Semantik 1

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

Anweisungen

. . . werden ausgeführt. Mögliche Seiteneffekte:

- Ändern der Werte von Variablen
- Ein-/Ausgabeoperationen

Semantik 1

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

Anweisungen

. . . werden ausgeführt. Mögliche Seiteneffekte:

- Ändern der Werte von Variablen
- Ein-/Ausgabeoperationen

Semantik 2

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

Semantik 2

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

Semantik 2

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

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 1

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 1

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 1

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 1

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

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 1

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

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 1

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 1

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

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

Beispiele Semantik von (Mini)Triangle 2

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

Beispiele Semantik von (Mini)Triangle 2

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

Beispiele Semantik von (Mini)Triangle 2

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

Beispiele Semantik von (Mini)Triangle 2

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

Beispiele Semantik von (Mini)Triangle 2

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

- 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
  - Für Optimierungen nur Untermenge!
- Beschreibung in PLPJ, Anhang B

Triangle

- 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
  - Für Optimierungen nur Untermenge!
- Beschreibung in PLPJ, Anhang B

Triangle

- 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
  - Für Optimierungen nur Untermenge!
- Beschreibung in PLPJ, Anhang B

Triangle

- 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
  - Für Optimierungen nur Untermenge!
- Beschreibung in PLPJ, Anhang B

Zusammenfassung

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