

Einführung in den Compilerbau

Lena Thuy Trang Vo

Wintersemester 2024/25

Inhaltsverzeichnis

1	Einführung	2
1.1	Auswirkung von Compilern	2
1.2	Programmiersprachen	2
1.3	Unterschiedliche Abstraktionsebenen	2
2	Auswirkungen der Zielmaschine	3
2.1	Einfach: Henessy & Patterson DLX	3
2.2	Komplizierter: Analog Devices TigerSHARC	3
2.3	Problematisch: IBM/Sony Cell Processor	4
2.4	Spezialisierte Anforderungen an CPUs	4
2.5	Technologie	5
3	Aufbau von Compilern	5
3.1	Syntaxanalyse	6
3.2	Kontextanalyse	6
3.3	Code-Erzeugung	7
4	Optimierung	7
4.1	Optimierender Compiler	7
4.2	Beispiele für Optimierung	8
5	Syntax	8
5.1	Kontextuelle Einschränkungen	9
5.2	Semantik	9
5.3	Art der Spezifikation	9
5.4	Syntax	10
5.5	Syntax durch Mengenbeschreibung	10
5.6	Reguläre Ausdrücke (REs)	10
5.7	Kontextfreie Grammatiken (CFGs)	10
6	(Mini-)Triangle	11
6.1	Terminologie	11
6.2	Syntaxbäume	11
6.3	Konkrete und abstrakte Syntax	12
6.4	AST als Zwischendarstellung	13

Einführung

Schnittstelle zwischen Programmiersprache und Maschine

- **Programmiersprache** ist gut für den Menschen handhabbar
 - Smalltalk
 - Java
 - C++
- **Maschine** ist getrimmt auf
 - Ausführungsgeschwindigkeit
 - Preis/Chip-Fläche
 - Energieverbrauch
 - nur selten: leichte Programmierbarkeit

Auswirkung von Compilern

- entscheidet über dem Benutzer **zugängliche** Rechenleistung
- Compiler entscheiden maßgeblich über die **Effizienz und Leistung** eines Programms auf der Zielhardware
- beeinflussen nicht nur die **Geschwindigkeit der Programmausführung**, sondern auch den **Speicherbedarf** und die **Energieeffizienz**

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

```
01100001000000110
01110010010000001
1011000100010010
10010001000000110
```

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**
- Höhere Ebenen **erleichtern die Entwicklung komplexer Logik**, während niedrigere Ebenen **maximale Kontrolle** über die Hardware bieten

Auswirkungen der Zielmaschine

Einfach: Hennessy & Patterson DLX

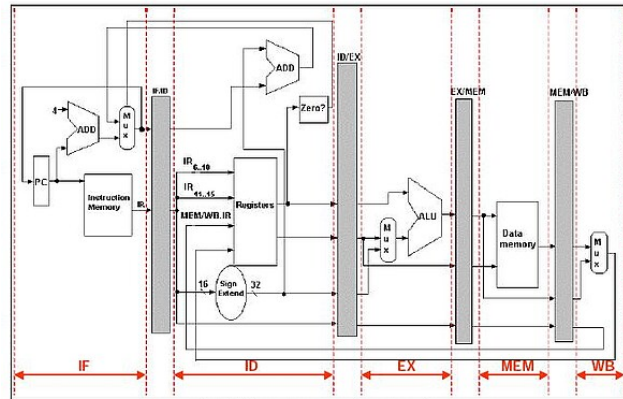


Figure 1: DLX processor architectural schematic

Abbildung 1: DLX-Prozessor

- ist eine **Reduced Instruction Set Computer** (RISC) Architektur, die auf **Einfachheit** und **Effizienz** ausgelegt ist
- basiert auf einem **Load/Store-Design**, bei dem **alle Operationen auf Register beschränkt** sind mit Ausnahme von Lade- und Speicheroperationen
- die DLX-Architektur verwendet eine **32-Bit-Datenbreite** und bietet 32 allgemeine Register, die jeweils 32 Bit breit sind
- implementiert eine **fünfstufige Instruktionspipeline**, die aus den Phasen **Instruction Fetch** (IF), **Instruction Decode** (ID), **Execution** (EX), **Memory Access** (MEM) und **Write Back** (WB) besteht

Komplizierter: Analog Devices TigerSHARC

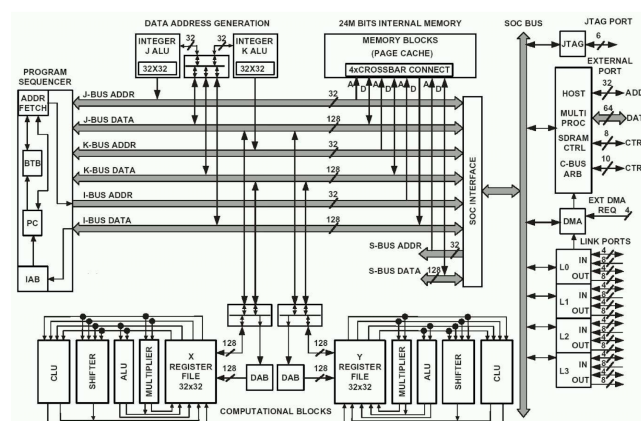


Abbildung 2: Analog Devices TigerSHARC

- ist ein leistungsstarker **digitaler Signalprozessor** (DSP), der für anspruchsvolle Anwendungen entwickelt wurde

- kann bis zu vier Instruktionen pro Taktzyklus ausführen was eine hohe Parallelität bei der Verarbeitung ermöglicht
- ist für Hochleistungs-Multiprocessing-Anwendungen ausgelegt, wie z.B. Motorsteuerung, Energiemanagement, Prozesssteuerung und Sicherheitssysteme
- Kombination aus hoher Ausführungsgeschwindigkeit, flexibler Datenverarbeitung und robuster Speicherverwaltung

Problematisch: IBM/Sony Cell Processor

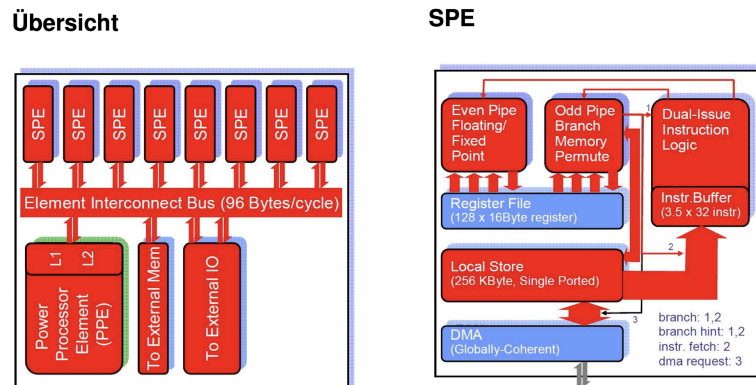


Abbildung 3: IBM/Sony Cell Processor

- ist eine innovative Prozessorarchitektur, die von IBM, Sony und Toshiba entwickelt wurde
- der zentrale Steuerprozessor basiert auf der 64-Bit-PowerPC-Architektur und kann zwei Threads gleichzeitig verarbeiten
- parallele Verarbeitung
- in Ps3

Spezialisierte Anforderungen an CPUs

Je nach Anwendungsgebiet mehr oder weniger wichtig...

- **Rechenleistung** (hoch/niedrig):
 - Anwendungen wie KI, Gaming oder CAD erfordern hohe Rechenleistung, während einfache Büroanwendungen weniger anspruchsvoll sind
- **Datentypen** (Gleitkomma, ganzzahlig, Vektoren):
 - wissenschaftliche Berechnungen benötigen oft Gleitkommaoperationen, während Grafikverarbeitung von Vektorenoperationen profitiert
- **Operationen** (Multiplikationen, MACs):
 - DSPs und Grafikprozessoren nutzen häufig MACs für schnelle Berechnungen in Signalverarbeitung und Rendering
- **Speicherbandbreite** (parallele Speicherzugriffe):
 - hohe Speicherbandbreite ist entscheidend für Anwendungen die große Datenmengen verarbeiten wie Videobearbeitung oder Simulationen

- **Energieeffizienz:**
 - in mobilen Geräten entscheidend, um die Akkulaufzeit zu maximieren
- **Platzbedarf** (für den Prozessorchip):
 - in eingebetteten Systemen und mobilen Geräten ist der Platzbedarf des Prozessors ein wichtiger Faktor

... können häufig nur durch spezialisierte Prozessoren erfüllt werden

→ **benötigen passende Compiler**

Technologie

- Taktfrequenz von Prozessoren nur **mit Mühe steigerbar**
- Trend geht weg von **hochgetakten Einzelprozessoren** hin zu vielen, aber langsameren Prozessoren
- um die Vorteile von Mehrkern-CPUs und GPUs zu nutzen, wurden verschiedene **parallele Programmiermodelle** entwickelt:
 - **OpenMP:** Mehr-Kern-CPUs
 - **NVidia CUDA:** GPUs
 - **OpenCL:** Heterogene Systeme (GPUs + CPUs, experimentell auch schon FPGAs)
- aber noch wenig abstrakt
 - Entwickler müssen **explizit angeben**, wie Aufgaben parallelisiert werden sollen, was **komplex** und **fehleranfällig** sein kann
- keine automatische Parallelisierung!

Anwendungsgebiet: Machine Learning

- Parallelisierung ist besonders wichtig im Bereich des maschinellen Lernens
- Frameworks wie TensorFlow XLA und Glow nutzen parallele Berechnungen, um komplexe Modelle effizienter zu trainieren und auszuführen

Aufbau von Compilern

- Vorgehen: Bearbeitung in **mehreren Phasen**
- **Zwischendarstellungen** werden verwendet, um Informationen zwischen diesen Phasen auszutauschen

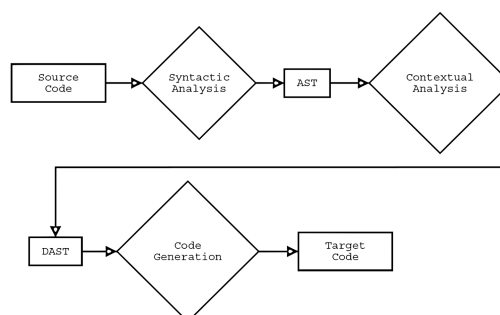


Abbildung 4: Vorgehen

Syntaxanalyse

- folgt auf die lexikalische Analyse und **überprüft**, ob der **Quellcode** den **Syntaxregeln** der jeweiligen Programmiersprache entspricht
- stellt sicher, dass der Quellcode grammatikalisch korrekt ist
- wenn der Code den Regeln entspricht, wird ein **Syntaxbaum** erstellt
 - dieser Baum repräsentiert die **hierarchische Struktur** des Codes und dient als Grundlage für die nachfolgenden Phasen wie die **semantische Analyse** und **Optimierung**
 - Baum dient als **Zwischendarstellung für den Informationsaustausch** zwischen den verschiedenen Phasen des Compilers.

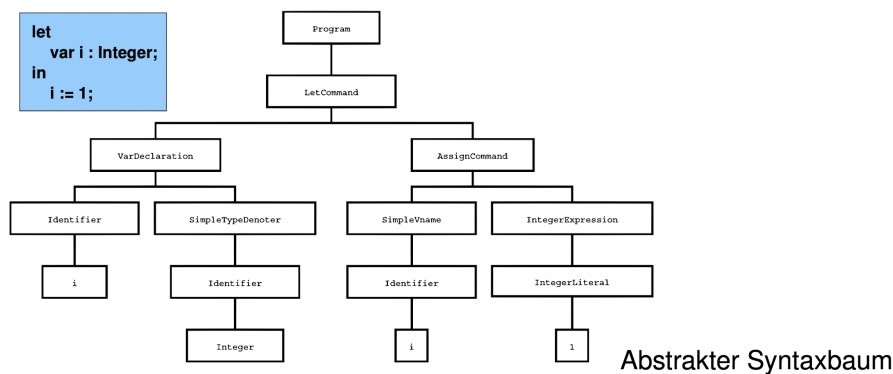


Abbildung 5: Syntaxbaum

Kontextanalyse

- auch als **semantische Analyse** bekannt
- überprüft, ob alle verwendeten Variablen **korrekt deklariert** wurden
- stellt sicher, dass jede Variable vor ihrer Verwendung definiert ist und dass **keine Mehrfachdeklarationen** im selben Gültigkeitsbereich vorhanden sind
- bestimmt die **Datentypen aller Ausdrücke** im Programm.
 - **Überprüfung von Typkompatibilität** bei Operationen und Zuweisungen, um sicherzustellen, dass beispielsweise keine Ganzzahlen mit Zeichenketten addiert werden

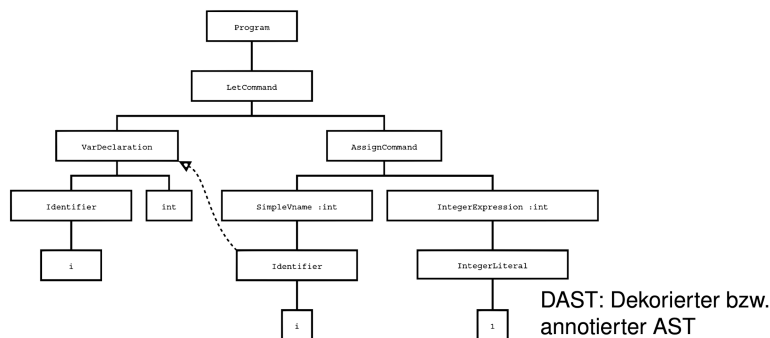


Abbildung 6: DAST

Code-Erzeugung

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

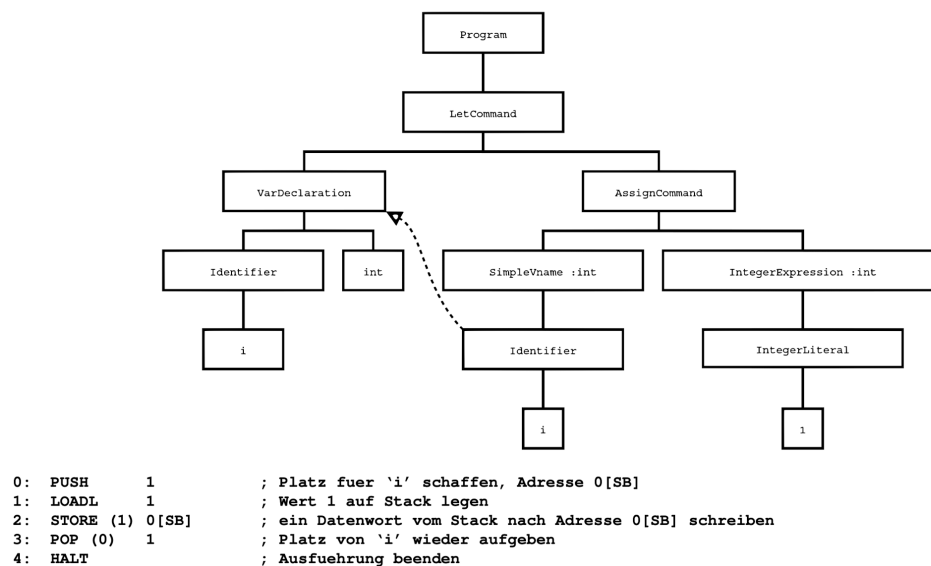
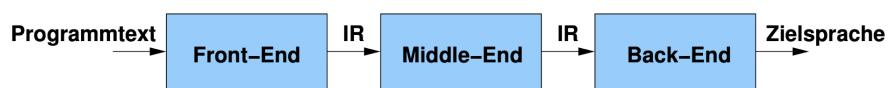


Abbildung 7: Code-Erzeugung

Optimierung

Optimierender Compiler



- **Front-End**: Syntaktische/kontextuelle Analyse
 - überprüft den Quellcode auf **syntaktische** und **semantische Korrektheit**
 - oder wird in eine **Intermediate Representation (IR)** umgewandelt, die als zentrale Datenstruktur dient

- **Middle-End:** Transformation von Zwischendarstellungen
 - hier werden **maschinenunabhängige Optimierungen** auf der IR durchgeführt
 - keine direkte Code-Erzeugung aus Front-End IR
 - verwendet in der Regel zusätzliche interne Darstellungen
- **Back-End:** Code-Erzeugung
 - optimierte IR wird in **Maschinencode** oder eine andere **Zielsprache** übersetzt

Hauptzweck eines optimierenden Compilers ist es, den erzeugten Code hinsichtlich **Ausführungszeit**, **Speicherverbrauch** oder **Energieeffizienz** zu verbessern.

Beispiele für Optimierung

Constant-Folding

$x = (2+3) * y$ $x = 5 * y$

- konstante Ausdrücke werden zur Compile-Zeit berechnet, um die Laufzeit zu reduzieren

Common-Subexpression Elimination

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

- identifiziert und eliminiert wiederholte Berechnungen, indem sie das Ergebnis einer Berechnung speichert und wiederverwendet.

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;
}

- ersetzt teure Operationen durch günstigere, um die Effizienz zu steigern

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;
}

- verschiebt Berechnungen, die innerhalb einer Schleife konstant bleiben, aus der Schleife heraus, um sie nur einmal auszuführen.

Syntax

Beschreibt die Satzstruktur von korrekten Programmen

- $n := n + 1$
 - syntaktisch korrektes Statement in Triangle
- Ein Kreis hat zwei Ecken:
 - syntaktisch korrekte Aussage, ist grammatikalisch korrekt, da sie den Regeln der deutschen Satzstruktur folgt, auch wenn sie inhaltlich falsch ist

Kontextuelle Einschränkungen

Definition

Geltungsbereich (Scope)

- Geltungsbereich bestimmt, wo im Programm eine Variable sichtbar und zugänglich ist
- hilft, Namenskollisionen zu vermeiden, indem er es ermöglicht, dass derselbe Name in verschiedenen Teilen des Programms unterschiedliche Bedeutungen hat

Definition

Typüberprüfung

- Beispielsweise muss eine Variable n vor ihrer Verwendung deklariert werden und kompatible Typen haben

Semantik

operationell:

- beschreibt die Bedeutung eines Programms **in Bezug auf die Schritte**, die **während der Programmausführung** ablaufen
- legt fest, wie ein **Programmzustand in einen anderen** übergeht

denotational:

- **bildet Eingaben auf Ausgaben ab** und abstrahiert von den konkreten Ausführungsschritten

Art der Spezifikation

Für alle drei Teile

1. Syntax
2. kontextuelle Einschränkungen
3. Semantik

...gibt es jeweils zwei Spezifikationsarten

- formal
- informal

Definition

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?

1. mathematische Mengennotation
2. reguläre Ausdrücke
3. kontextfreie Grammatik

Syntax durch Mengenbeschreibung

Beispiele für die beschriebenen Zeichenketten:

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

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

Reguläre Ausdrücke (REs)

- mächtiges Werkzeug zur Beschreibung von Mustern in Zeichenketten
- erweitere Zeichenkette aus dem Alphabet um Operatoren
 - $|$ zeigt Alternative an
 - $*$ zeigt Null oder mehr Vorkommen des vorangehenden Zeichens an
 - ε ist die leere Zeichenkette
 - $(...)$ erlauben die Gruppierung von Teilausdrücken durch Klammerung

Kontextfreie Grammatiken (CFGs)

Eine kontextfreie Grammatik besteht aus

- einer Menge von Terminalsymbolen T aus Alphabet
- einer Menge von Nicht-Terminalsymbolen N
- einem Startsymbol S
- einem Startsymbol $S \in N$
- einer Menge von Produktionen P
 - beschreiben, wie Nicht-Terminalsymbole aus Terminalsymbolen zusammengesetzt sind

Backus-Naur-Form (BNF)

- Produktionen werden als Regeln geschrieben, bei denen ein **Nicht-Terminalsymbol** auf eine Zeichenkette aus **Terminal- und Nicht-Terminalsymbolen abgebildet** wird
- Nicht-Terminal ::= **Zeichenkette**

erweiterte Backus-Naur-Form (EBNF)

- erweitert BNF durch zusätzliche Operatoren, die die Grammatik **kompakter** und **ausdrucksstärker** machen
- ermöglicht die Verwendung von regulären Ausdrücken zur Beschreibung der Produktionen
- aus Terminal und Nicht-Terminalsymbolen
- Nicht-Terminal ::= **RE**

(Mini-)Triangle

- pascal-artige Sprache als Anschauungsobjekt
- Compiler-Quellcode auf Webpage

```

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
end

```

Abbildung 8: Mini-Triangle

Terminologie

Phrase: von einem gegebenen Nicht-Terminalsymbol herleitbare Kette von Terminalsymbolen

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

Abbildung 9: Beispiel

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 X_1, \dots, X_n hat, entsprechend der Produktion $N := X_1, \dots, X_n$
 - N ist ein Nicht-Terminalsymbol, welches durch die Symbole X_1, \dots, X_n ersetzt werden kann

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

Expression-Baum für $d + 10 * n$

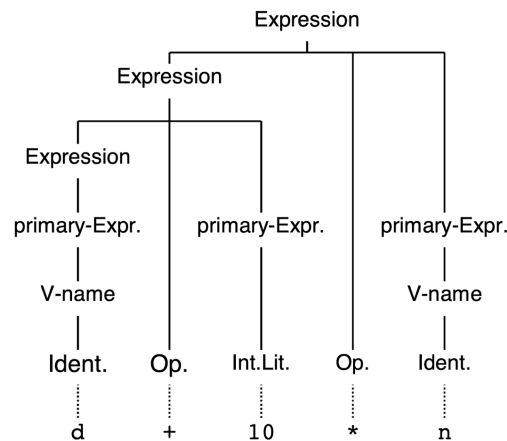


Abbildung 10: Syntaxbaum

single-Command-Baum

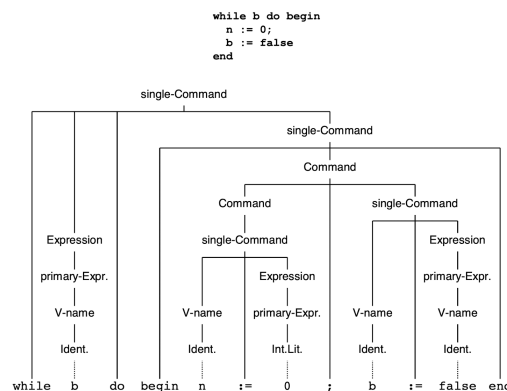


Abbildung 11: Syntaxbaum

Konkrete und abstrakte Syntax

Konkrete Syntax

- beschreibt die **genaue, textuelle Form** eines Programms
- legt fest, wie Programme geschrieben werden müssen, um von einem Compiler oder Interpreter als korrekt erkannt zu werden
- konkrete Syntax hat **keinen Einfluss** auf **Semantik** der Programme

Abstrakte Syntax

- ist eine vereinfachte Darstellung der Struktur eines Programms
- entfernt unnötige syntaktische Details und konzentriert sich auf die **wesentlichen semantischen Strukturen** eines Programms
- Idee: Orientierung an der Subphrasen-Struktur der Produktionen
- Schlüsselworte wie **do, :=** sind irrelevant

- Unterscheidungen zwischen
 - Command und Single-Command
 - Declaration und Single-Declaration
 - Expression und primary-Expression
- ...sind nur für das Erkennen des Programms relevant, nicht zur Darstellung seiner Semantik

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

Abbildung 12: Auszug aus der abstrakten Syntax

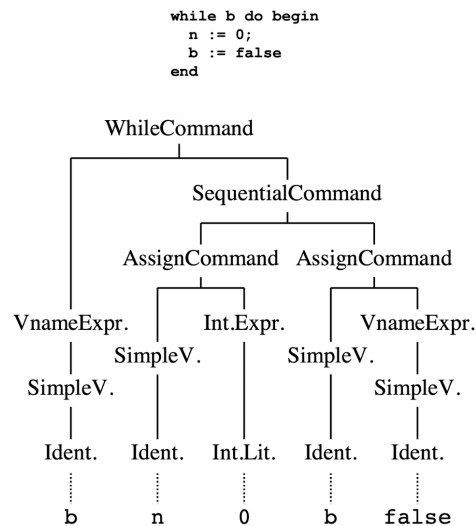


Abbildung 13: Beispiel abstrakte Syntax

AST als Zwischendarstellung

- ein **abstrakter Syntaxbaum** ist eine weit verbreitete Form der Zwischendarstellung (IR)
- **höhere Abstraktionsebene**, 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
- schlechter geeignet für **maschinennahe Analyse und Transformationen**
 - Ausnutzung von Maschinenregistern

- Ausnutzung von speziellen Maschinenbefehlsfolgen
- Hier Konzentration auf maschinenunabhängige Ebene
 - (D)AST ist Hauptrepräsentation
 - für einzelne Bearbeitungsschritte: Andere IRs