Einführung in den Compilerbau

Vorlesungsnotizen

von Patrick Elsen, Viola und Michael Matthé

Andreas Koch

Wintersemester 2018-2019
Technische Universität Darmstadt

Inhaltsverzeichnis

1 Organisatorisches

1.1 Literatur

Die Vorlesung basiert fast vollständig auf *Programming Language Processors in Java*[1]. Auszugsweise noch weiteres Material, z. B. zum ANTLR-Parsergenerator.

Einen guten allgemeinen Überblick, aber im Detail mit anderen Schwerpunkten als diese Vorlesung, bietet *Compilers, 2. Auflage*[2].

1.2 Aufbau der Veranstaltung

Diese Veranstaltung ist logisch in mehrere Teile gegliedert.

Front-End[3] Behandelt das Lexing und Parsing von Sprachen und Zwischendarstellungen, die in diesen Stages benutzt werden.

Middle-End Behandelt die semantische Analyse und die Kontextanalyse.

Back-End Behandelt die Laufzeitorganisation und Code-Erzeugung.

Front-End-Generatoren Behandelt deren Verwendung.

Java Virtuelle Maschine

2 Einleitung

2.1 Compiler

Ein Compiler ist eine Schnittstelle zwischen *Mensch* und *Maschine*. Er übersetzt von einer Programmiersprache (menschenlesbar) in eine Maschinensprache (maschinenlesbar).

Programmiersprache Gut für Menschen lesbar. Beispiele für Programmiersprachen sind Smalltalk, Java und C++.

Maschinensprache Getrimmt auf Ausführungsgeschwindigkeit, Preis (Fläche), Energieverbaucht, nur sehr selten auf leichte Programmierbarkeit.

2.2 Auswirkung von Compilern

Mit den Eigenschaften entscheidet ein Compiler über die dem Nutzer zugängliche Rechenleistung. Durch gewisse Abwägungen kann ein Compiler ein Programm so kompilieren, dass es am schnellsten läuft oder dass der Resultierende Code an kleinsten ist. In Tabelle 1 ist zu erkennen, welche Tradeoffs unterschiedliche Compiler machen: der von GCC generierte Code ist zwar langsamer, aber auch kleiner, während ICC schnellen Code generiert, der dafür aber auch voluminöser ist.

[1] von David Watt und Deryck Brown, Prentice-Hall 2000

[2] Von Aho, Sethi, Ullmann, Lam, Addison-Wesley 2006. Auch auf Deutsch verfügbar.

[3] Die ersten drei Teile der Veranstaltung richten sich an die Veranstaltungen *IMT3052* von Ivar Farup, Universität Grøvik, Norwegen; und *Vertalerbouw* von Theu Ruys, Universität Twente, Niederlande.

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

Tabelle 1: Bildkompression auf Dothan CPU, 2 GHz

2.3 Programmiersprachen

Hohe Ebene Smalltalk, Java, C++. Beispiel:

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

Mittlere Ebene Assembly

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

Niedrige Ebene Machinensprache

```
01100001 00000110
0111001001000001
1011000100010010
10010001 00000110
```

2.4 Abstraktionsebenen

Auf den unteren Ebenen werden die Beschreibungen immer feiner, da man näher an der Zielmaschine (Hardware) arbeitet.

Der Compiler ist dafür zuständig, Details hinzuzufügen. In den obenstehenden Codebeispielen musste der Compiler unter anderem Register wählen (hier R1 für den Wert von `i`, R2 für die Konstante 1, und R1 für das Ergebnis der Addition), in denen die Werte währen dem Programmablauf zwischengespeichert werden. Außerdem musste die Adresse der Variable `i`, hinzugefügt werden, wobei hier `00000110` verwendet wurde.

Diese Details werden mithilfe von verschiedensten Algorithmen ergänzt, welche die Programmeigeschaften Analysieren und durch die Synthese von Details die Beschreibung Verfeinern.

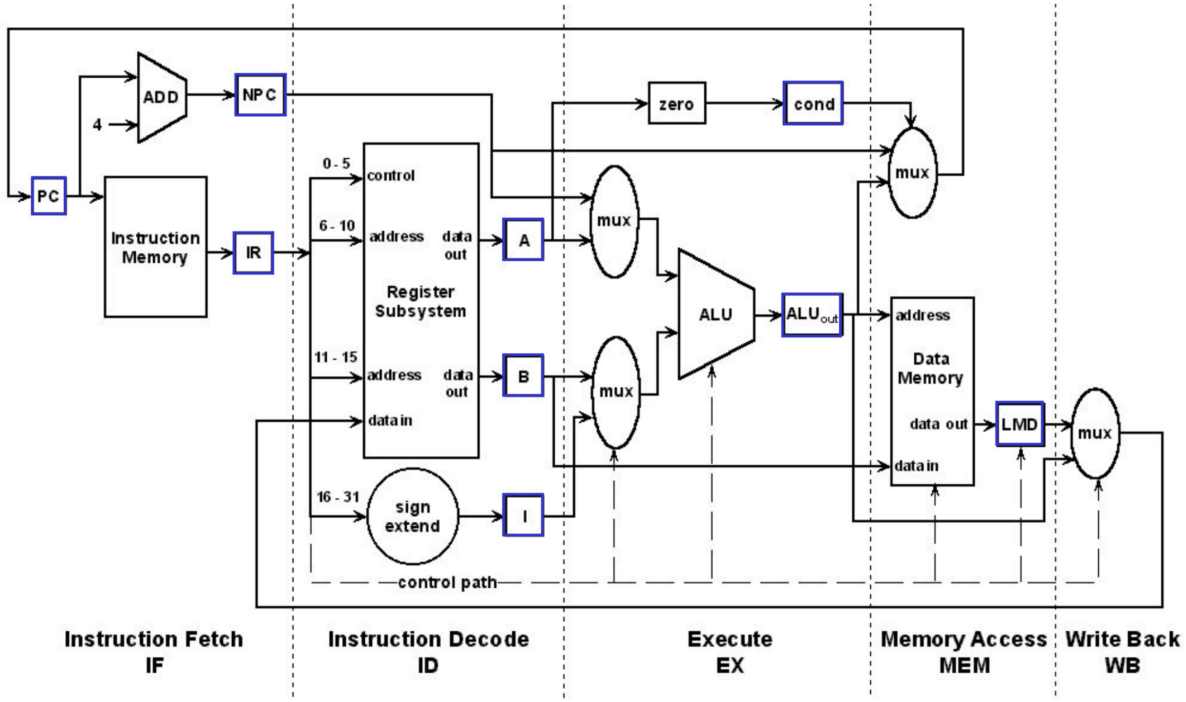

Abbildung 1: Die DLX RISC Prozessorarchitektur

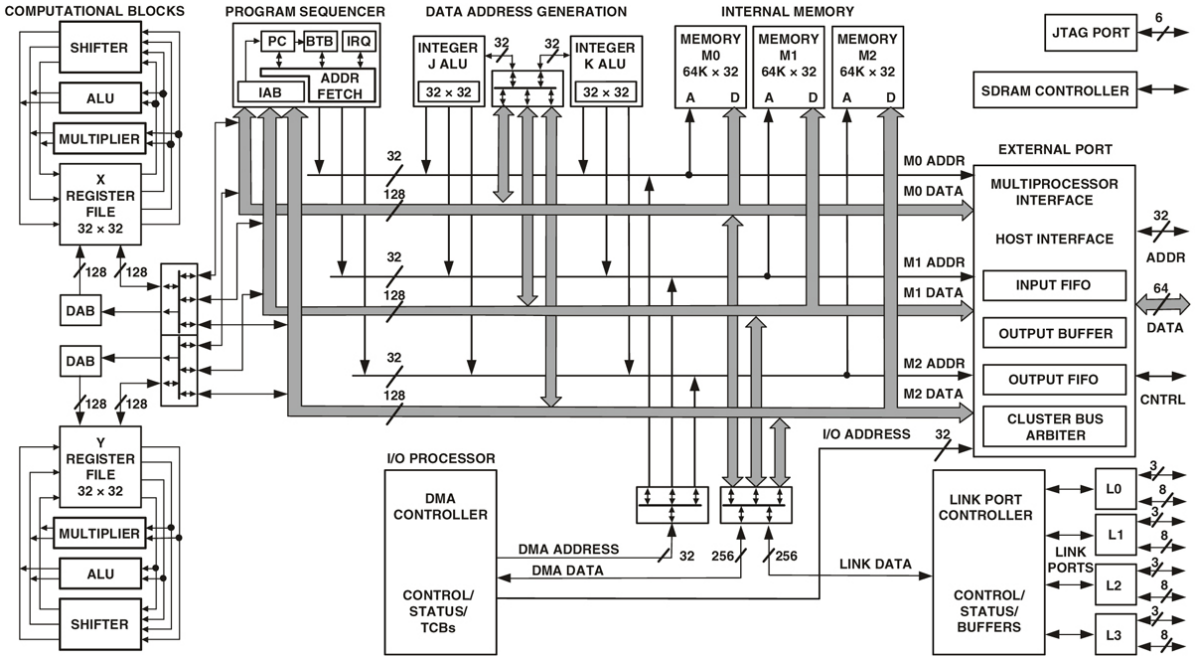

Abbildung 2: Analog Devices TigerSHARC

Abbildung 3: Ein *Synergistic Processing Element* eines Cell Prozessors.

3 Zielmaschine

3.1 Auswirkungen der Zielmaschine

Die Zielmaschine hat einen Einfluss auf die Architektur des Compilers. So basiert zum Beispiel die DLX-Architektur von John Hennessy und David Patterson auf der MIPS Architektur, und ist nur leicht verändert um diese zu modernisieren. Damit ist es sehr einfach, einen Compiler zu bauen, der für diese Architektur Code generiert.

Etwas komplizierter wird es mit der *TigerSHARC* Architektur von Analog Devices. Dies ist ein Beispiel für ein DSP, also *Digital Signalling Processor*. Es gibt hier zwei *Computational Blocks*, damit parallel Rechnungen ausgeführt werden können. Diese können aber nicht kommunizieren, also muss der Compiler drauf achten, dass auf die Register nur von dem jeweiligen Block aus zugegriffen werden können. Außerdem besitzt diese Architektur separate Rechenblocks, um Adressen zu bestimmen. Ein Compiler für diese Architektur muss also wissen, wie die Architektur aufgebaut ist, um sie effizient zu nutzen.

Am problematischsten wird das aber erst bei extremen Architekturen wie der des IBM/Sony *Cell* Prozessors. Hierbei handelt es sich um eine sehr gewagte, und wie sich leider herausgestellt hat, zu komplexe Architektur. Es gibt ein PowerPC-basierten Hauptprozessor, der aber nicht sehr leistungsstark ist. Die eigentliche Arbeit kann von *Synergistic Processing Elements*, kurz SPE, ausgeführt werden. Diese sind so konstruiert, dass ein sorgfältig produzierter Instruktionsstream die Hardware maximal ausnutzen kann, mit parallelen Arithmetischen- und Speichereinheiten (siehe Abbildung 3). Ein Compiler für diese Architektur hat es also wesentlich schwerer.

3.2 Anforderungen an CPUs

Je nach Anwendungsgebiet sind die Anforderungen mehr oder weniger wichtig. CPUs werden mit Spezifikationen wie der gewünschten Rechenleistung, Datentype und Operationen, die effizient verarbeitet werden sollen (Gleitkomma, Vektoren, Matrizen, multiplikation, *multiply-accumulate*), Speicherbandbreite, Energieeffizienz und Platzbedarf konstruiert. Oft können bestimmte Anforderungen nur durch spezialisierte Prozessoren realisiert werden. Diese spezialisierten Prozessoren brauchen auch passende Compiler.

3.3 Paralleles Rechnen

Paralleles Rechnen ist die Königsklasse der Forschung. Warum eigentlich? Das Wettrennen der Taktfrequenzen ist im großen und ganzen vorbei, mehr als ~4 GHz ist nicht realistisch. Also ist der Trend weg von hochgetakteten Einzelprozessoren und hin zu vielen (2-8, teilweise 16) Prozessoren, die aber weniger schnell getaktet werden. Dadurch erreicht man mehr Rechenleistung. Aber wie kann man für solche parallele Rechenkapazitäten programmieren?

Erste praktische Ansätze sind OpenMP, mit dem man für parallele CPUs programmieren kann. OpenCL kann mit heterogenen Systemen (GPUs, CPUs, experimentell auch FPGAs) arbeiten, und für NVidia Grafikkarten gibt es CUDA. Diese Lösungen erfordern aber aktuell noch, dass der Programmierer explizit für ein paralleles System programmiert. Es gibt noch keine automatische Parallelisierung.

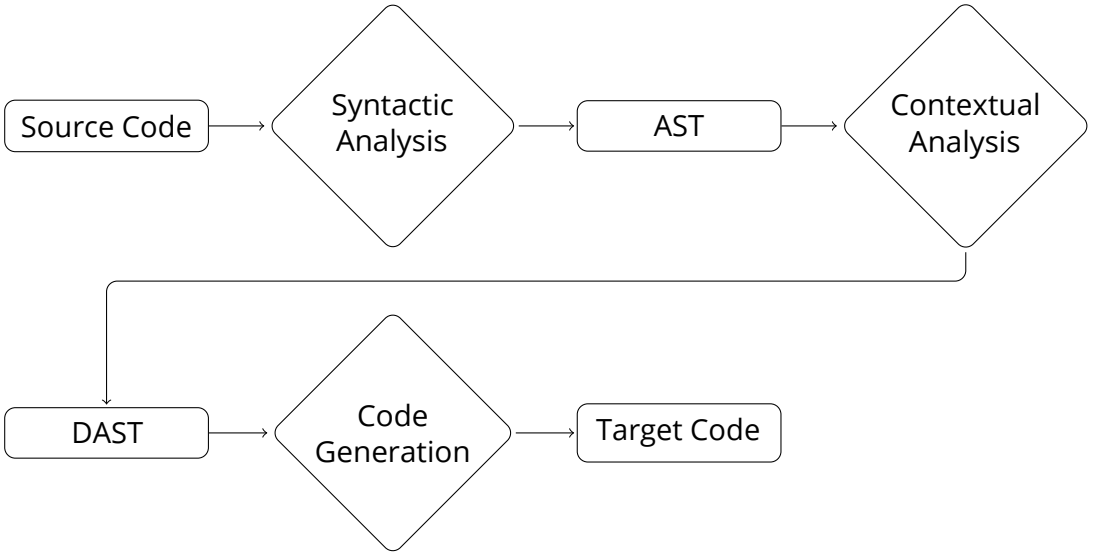

Abbildung 4: Zwischendarstellungen für den Informationsaustausch

3.4 Compilerbau in der Lehre

Warum wird der Compilerbau so früh gelehrt? Es handelt sich hierbei um eine Kombination von verschiedenen Disziplinen. Zum Compilerbau benötigt man einen *Parser*, welcher zur theoretischen Informatik gehört. Man braucht auch das Hintergrundwissen der *Architektur der Zielmaschine*, welches zur technischen Informatik gehört. Außerdem braucht man zum Compilerbau Kenntnisse von Software-Engineering, was zur praktischen Informatik gehört. Damit vereint der Compilerbau drei wichtige Disziplinen. Außerdem werden Compiler von vielen Firmen in vielen unterschiedlichen Kontexten gebraucht.

4 Aufbau

Compiler arbeiten generell in mehreren Phasen. In Abbildung 4 sind diese Phasen aufgezeigt. Außerdem sind die Zwischendarstellungen, die zum Informationsaustausch zwischen den Phasen genutzt werden, dargestellt.

4.1 Syntaxanalyse

Bei der Syntaxanalyse wird überprüft, ob das Programm Syntaxgerecht aufgebaut ist. Das Programm wird in einer geeigneten Darstellung gespeichert. Das Resultat ist dann ein *Abstract Syntax Tree*, kurz AST. Abbildung 5 zeigt anhand eines Beispiels, wie das Resultat der Syntaxanalyse aussehen könnte.

4.2 Kontextanalyse

Bei der Kontextanalyse werden Variablen ihren Deklarationen zugeordnet. Außerdem werden die Typen von Ausdrücken berechnet. Hier wird ein *Decorated Abstract Syntax Tree*,

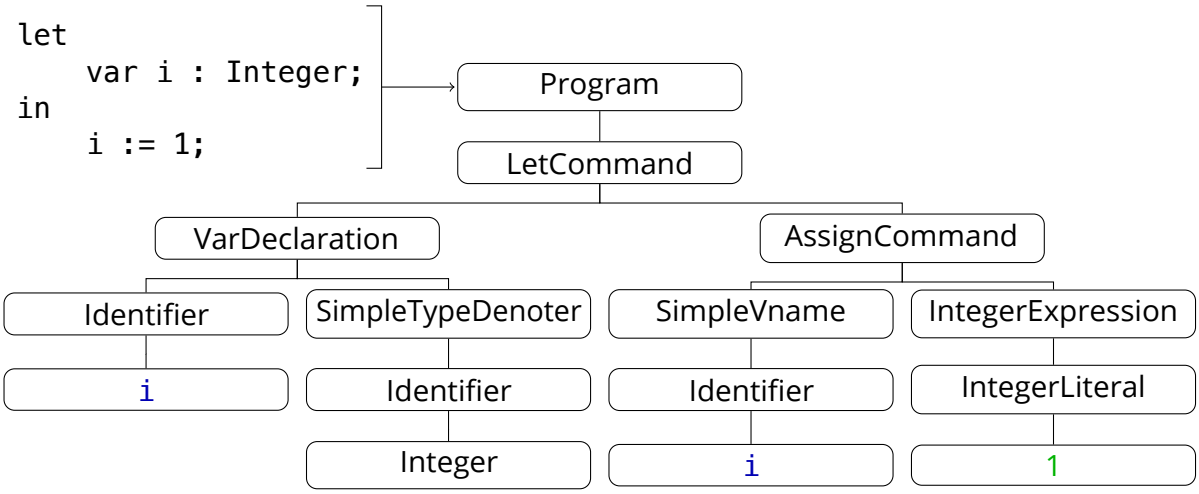

Abbildung 5: Beispiel der Syntaxanalyse

kurz DAST, generiert. Dieser ist wie ein AST, nur mit Zusatzinformationen.

4.3 Code-Erzeugung

Wenn das Programm syntaktisch und kontextuell korrekt ist, kann es in die Zielsprache übersetzt werden. Eine Zielsprache kann eine Low-Level Sprache wie die Maschinensprache oder Assembly, aber auch C oder eine andere Hochsprache.

Dazu wird DAST-Teilen Instruktionen der Zielsprache zugeordnet. Variablen werden gehandhabt, indem zum Beispiel bei einer Deklaration Speicherplatz reserviert wird, und bei der Verwendung der zugeordnete Speicherplatz referenziert wird.

5 Optimierung

5.1 Optimierender Compiler

Bei einem optimierenden Compiler findet die Kompilierung in drei Phasen statt.

Front-End Macht die syntaktische und kontextuelle Analyse und gibt den AST in einem IR-Format (*Intermediate Representation*) aus.

Back-End Liest das IR, und erzeugt den Code.

Middle-End Transformation von Zwischendarstellungen. Dieses arbeitet auf der IR. Hier wird kein Code ausgegeben, sondern nur der IR optimiert. Intern werden hier auch oft zusätzliche Darstellungen verwendet werden.

Anmerkung
LLVM ist ein sehr bekanntes Projekt, welches eine Art Plugin-basierte Architektur aufweist. Es können Front-Ends geschrieben werden, die unterschiedliche Programmiersprachen unterstützen (zum Beispiel `clang` für C und C++, `rubinius` für Ruby). Optimierungen können ebenso als Plugin geschrieben werden, und arbeiten auf dem LLVM IR. Es können außerdem Back-Ends geschrieben werden, die für unterschiedlichste Architekturen Code ausgeben. Somit funktioniert zum Beispiel das *Emscripten* Projekt, mit welchem man C und C++ Code in JavaScript kompilieren kann.

5.2 Beispiele für Optimierung

Constant Folding Bei dieser Optimierung werden Ausdrücke, die ein konstantes Ergebnis haben, von Optimierer ausgewertet. Hat man zum Beispiel den Code

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

Bei dem Constant Folding erkennt der Compiler, dass der ausdruck `2 + 3` konstant ist. Er kann den Ausdruck dann bestimmen, und den Code in eine minimierte Version transformieren.

```
int x = 5 * y;
```

Anmerkung
Sowohl C++ (seit C++11) als auch Rust haben mittlerweile Support für Constant Folding in die Sprache eingebaut. Mit dem `constexpr` Keyword kann man dem Compiler sagen, dass er einen Funktion oder Variable als Konstante ansehen und diese während der Kompilierung auswerten soll.

Common Subexpression Elimination Hierbei werden Ausdrücke (*Subexpressions*), die öfters im Code vorkomment (*Common*) eliminiert. Hat man zum Beispiel folgenden Code:

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

So kann der Compiler erkennen, dass der Ausdruck `5 * a` mehrmals vorkommt. Wenn er garantieren kann, dass der Wert von `a` während den Instruktionen konstant bleibt, kann er diese vereinfachen, indem er eine temporäre Variable einführt.

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

6 Syntax

6.1 Syntax oder Grammatik

Der Syntax beschreibt die Satzstruktur von korrekten Programmen. Jede Sprache hat ihren eigenen Syntax. So ist zum Beispiel der folgende Ausdruck Syntaktisch korrekt in C.

```
int n = m + 1;
```

Der Satz „Ein Kreis hat zwei Ecken“ ist wiederum eine syntaktisch korrekte Aussage im Deutschen (was aber nicht bedeutet, dass diese auch *semantisch* korrekt ist).

6.2 Kontextuelle Einschränkungen

Nicht jede Aussage, die syntaktisch korrekt ist, ist in dem Kontext auch erlaubt. Es gibt gewisse Regeln, die vom Geltungsbereich (*scope*) und dem Typ von Aussagen abhängen. So ist der folgende Code zum Beispiel nur dann korrekt, wenn die Variable x deklariert ist.

```
int y = 2 * x;
```

Bei dem Satz „Ein Kreis hat zwei Ecken“ ist der Fehler, dass ein Kreis keine Ecken besitzt. Hier passen die Typen nicht, ein Objekt der Klasse *eckige geometrische Formen*, wie zum Beispiel das Dreieck, würde hier kontextuell passen (aber der Satz wäre dann semantisch immer noch falsch).

6.3 Semantik

Bei der Semantik geht es um die *Bedeutung* einer Anweisung oder Aussage einer Sprache. Bei Programmiersprachen wird die Semantik häufig auf zwei Arten beschrieben.

Operationell Welche Schritte laufen ab, wenn das Programm gestartet wird? Wird unter Anderem in Form von *Pseudocode* angegeben.

Denotational Abbildung von Eingaben auf Ausgaben. Hier gibt man funktional an, was geändert wird, aber nicht, wie dies geschieht. Als Beispiel kann man die Anweisung $x := 1$ nehmen. Diese könnte man denotational als $v := i$ so spezifizieren, dass man eine Funktion f auf den Speicher angewendet wird, so dass nach der Anweisung an der Stelle v im Speicher der Wert i steht.

$$f(\mathrm{MEM}) = \mathrm{MEM}' \doteq \mathrm{MEM}'[v] = i$$

6.4 Art der Spezifikation

Für alle drei Teile (Syntax, kontextuelle Eingschränkungen und Semantik) gibt es jeweils zwei Spezifikationsarten: *Formal* und *Informal*.

Für die Sprache Triangle, die in dieser Vorlesung verwendet wird, gibt es eine formale Syntaxbeschreibung (als regulären Ausdrücke in der EBNF[4], sowie informale kontextuelle Einschränkungen und Semantik.

[4] Erweiterte Backus-Naur-Form, kurz EBNF, ist eine Erweiterung der Backus-Naur-Form (BNF), die ursprünglich von Niklaus Wirth zur Darstellung der Syntax der Programmiersprache Pascal eingeführt wurde.

Der Unterschied zwischen einer formalen und informalen Spezifikation ist, dass formale Beschreibungen automatisiert gelesen werden können und mit bestimmten Tools (Parser-Generatoren, wie zum Beispiel ANTLR[5]) automatisch verarbeitet werden können.

Mit einer formal beschriebenen Semantik kann man auch wieder automatisierte Tools verwenden, die zum Beispiel automatisiert Codegeneratoren schreiben können oder gewisse Eigenschaften von Programmen beweisen können. Leider werden in der Praxis Semantiken fast nie formal beschrieben, weil das zu kompliziert ist. Deswegen hat man meistens eine informelle Spezifizierung (einen Sprachstandard). Diese sind menschenlesbar. Der Nachteil hierbei ist, dass unter Umständen unterschiedliche Menschen den Standard unterschiedlich interpretieren, was zu Inkompatibilitäten führen kann.

6.5 Formale Syntax

Eine Sprache ist eine Menge von Zeichenketten aus einem Alphabet. Wie ist diese Menge angegeben? Bei endlichen Sprachen kann man einfach alle Elemente aufzählen, zum Beispiel {`true`, `false`} für die Sprache der booleschen Literale. Das geht aber nicht bei unendlichen Sprachen. In der Praxis sind aber alle Programmiersprachen unendlich.

Es gibt mehrere mögliche Vorgehensweisen, um Sprachen zu notieren. Man könnte die mathematische Mengennotation benutzen, reguläre Ausdrücke, oder eine kontextfreie Grammatik.

6.6 Syntax durch Mengenbeschreibung

Man kann eine Sprache durch mathematische Mengennotation beschreiben. Das kann zum Beispiel folgendermaßen aussehen.

$L = \{a, b, c\}$	beschreibt **a**, **b**, **c**.
$L = \{x^n \mid n > 0\}$	beschreibt **x**, **xx**, **xxx**.
$L = \{x^n y^m \mid n > 0, m > 0\}$	beschreibt **xxy**, **xyy**, **xxxyy**.
$L = \{x^n y^n \mid n > 0\}$	beschreibt **xy**, **xxyy**, **xxxyyy**.

Diese Art der Spezifikation ist leider nicht sonderlich nützlich bei der Spezifikation komplexerer Sprachen.

6.7 Reguläre Ausdrücke

Reguläre Ausdrücke bestehen aus Zeichen und Operatoren. Tabelle 2 zeigt einige der Operatoren von regulären Ausdrücken. Wie mächtig sind regulären Ausdrücke eigentlich? Kann man die Menge aller regulären Ausdrücke als regulären Ausdruck beschreiben? Leider ist das nicht möglich. Reguläre Ausdrücke sind also nicht mächtig genug und deswegen ungeeignet zur Beschreibung der Syntax komplexer Programmiersprachen. Dennoch sind diese innerhalb von Compilern nützlich.

[5] ANTLR unterstützt die Erzeugung von Parsern, Lexern und TreeParsern für LL(k)-Grammatiken mit beliebigen k. Die verwendete Eingabe-Sprache ist eine Mischung aus formaler Grammatik und Elementen aus objektorientierten Sprachen.

OPERATOR	BEDEUTUNG	BEISPIEL	
ε	Leere Zeichenkette	ε	ε
$\|$	Alternativen	$a\|b\|c$	**a**, **b**, **c**
$+$	Ein oder mehr Vorkommen	$a+$	**a**, **aa**, **aaa**, ...
$*$	Null oder mehr Vorkommen	$ab*$	**a**, **ab**, **abb**, ...
$(...)$	Gruppierung von Teilausdrücken	$(ab)*$	ε, **ab**, **abab**, ...

Tabelle 2: Liste von Operatoren von regulären Ausdrücken

6.8 Kontextfreie Grammatiken

Eine kontextfreie Grammatik besteht aus vier Teilen: eine Menge von Terminalsybolen T aus dem Alphabet, eine Menge von Nicht-Terminalsymbolen N, eine Menge von Produktionen P sowie einem Startsybol $S \in N$. Die Menge von Produktionen beschreibt, wie die Nicht-Terminalsymbole aus den Terminalsymbolen zusammengesetzt sind.

Eine Möglichkeit, diese Produktionen anzugeben, ist die Backus-Naur-Form. Dabei wird jede Produktion als Tupel angegeben, bei dem auf der linken Seite ein Nicht-Terminalsymbol steht, und auf der rechten Seite eine Zeichenkette aus Terminal und Nicht-Terminalsybolen.

$$\langle \text{Nicht-Terminal} \rangle ::= \text{Zeichenkette aus } \langle \text{Terminal} \rangle \text{ und } \langle \text{Nicht-Terminal} \rangle$$

Kompakter und übersichtlicher ist es aber, wenn man statt der BNF die *Erweiterte Backus-Naur-Form*, kurz EBNF, verwendet. Diese ist fast genauso wie die BNF aufgebaut, hier nimmt man aber auf der rechten Seite einen regulären Ausdruck von Terminal- und Nicht-Terminalsymbolen.

$$\langle \text{Nicht-Terminal} \rangle ::= \text{Regulärer Ausdruck aus } \langle \text{Terminal} \rangle \text{ und } \langle \text{Nicht-Terminal} \rangle$$

Der Unterschied zu einer *kontextbehafteten Grammatik* ist, dass t

6.9 BNF Beispiel

Eine mit BNF spezifizierte Grammatik könnte also folgendermaßen aussehen.

$$\begin{aligned} T &= \{x, y\} \\ N &= \{S, B\} \\ S &= S \\ P &= \{\} \end{aligned}$$

6.10 Mehrdeutigkeit in BNF

7 Triangle

In der Vorlesung beschäftigen wir uns mit der Sprache *Triangle*. Triangle ist eine Pascal-artige Sprache als Anschauungsobjekt. Die Version von Triangle, die wir in der Vorlesung

verwenden, ist im Vergleich zu der regulären Version sehr abgespeckt. Der Compiler-Quellcode ist auf der Kurswebseite verfügbar.

```
let ──────────────────────────────── Lokale Deklarationen
   const MAX ~ 10; ──────────────── Konstante (hässliches ~)
   var n: Integer
in begin
   getint(var n); ──────────────── Prozedur
   if (n > 0) /\ (n <= MAX) then
      while n > 0 do begin
         putint(n); puteol();  ┐─── Anweisungen
         n := n - 1            ┘
      end
   else ─────────────────────────── Else
end
```

Was man bei diesem Codebeispiel sieht, sind einige Designfeatures der Triangle-Sprache. So kann man innerhalb eines `let`-Blockes lokale Deklarationen angeben, die in dem darauffolgenden `begin`-Block gültig sind. Man kann sowohl Konstanten als auch Variablen deklarieren. Interessant beim Prozedurenaufruf ist, dass Variablen, die von der Prozedur verändert werden können, mit dem Keyword `var` übergeben werden müssen. Die etwas kurios wirkende Konstruktion aus zwei Schrägstrichen, `/\`, soll den boole'schen UND-Operator darstellen. Eine weitere Besonderheit ist der `else`-Zweig, der *immer* mit angegeben werden muss, was den Compiler vereinfacht.

Anmerkung
Die Triangle-Sprache ist nicht darauf ausgelegt, eine nützliche Sprache darzustellen. Vielmehr geht es hierbei darum, eine stark vereinfachte Sprache anzubieten, anhand deren der Compilerbau gelernt und geübt werden kann.

7.1 Syntax von Triangle

In Abbildung 6 ist der Syntax von Triangle[6] sichtbar. Man erkennt, dass Triangle im Vergleich zu gewissen anderen Programmiersprachen einen recht überschaubaren Syntax besitzt.

Es wird ein wenig Terminologie eingeführt, die beim Verständnis und zur Kommunikation wichtig ist. Eine *Phrase* ist eine von einem gegebenen Nicht-Terminalsymbol herleitbare Kette von Terminalsymbolen. Zum Beispiel eine *Expression-Phrase*, *Command-Phrase*. Ein *Satz* ist eine S*-Phrase*, wobei S das Startsymbol der kontextfreien Grammatik ist.

7.2 Syntaxbaum

Aufbauend auf die Konzepte der Phrasen und Sätze kann man den Syntax in Form eines Baumes darstellen. Diese Darstellung nennt man einen *Syntaxbaum*. Ein solcher Syntax-

[6] So, wie er in den Folien gezeigt wird.

⟨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⟩

⟨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⟩ ⟨Number⟩

⟨Integer-Literal⟩ ::= ⟨Digit⟩
| ⟨Integer-Literal⟩ ⟨Digit⟩

⟨Operator⟩ ::= `+` | `-` | `*` | `/` | `<` | `>` | `=`

⟨Declaration⟩ ::= ⟨single-Declaration⟩
| ⟨Declaration⟩ `;` ⟨single-Declaration⟩

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

⟨Type-denoter⟩ ::= ⟨Identifier⟩

⟨Comment⟩ ::= `!` ⟨CommentLine⟩ `eol`

⟨CommentLine⟩ ::= ⟨Graphic⟩ ⟨CommentLine⟩

⟨Graphic⟩ ::= any printable character or space

⟨Digit⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Abbildung 6: Syntax von Triangle in BNF

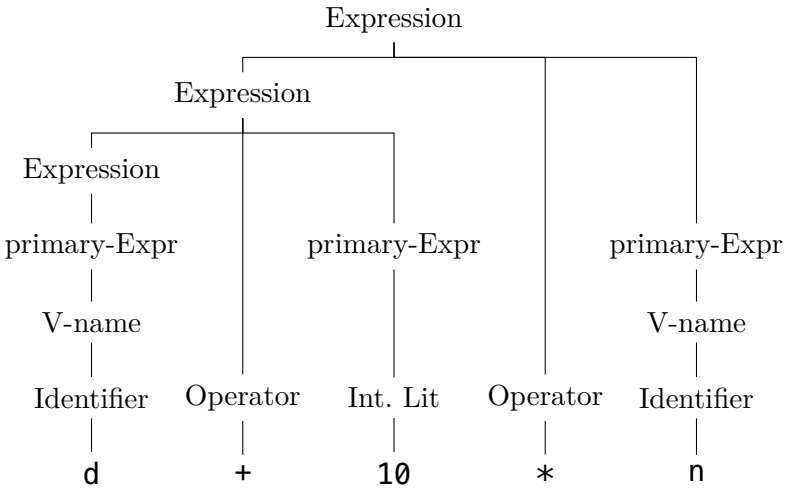

Abbildung 7: Syntaxbaum

baum ist ein geordneter, markierter Baum, bei dem die Blätter mit Terminalsymbolen markiert sind, die inneren Knoten mit Nicht-Terminalsymbolen markiert sind, und jeder Knoten Kinder hat, die mit der rechten Seite seiner Produktion übereinstimmen. Abbildung 7 zeigt ein Beispiel für einen solchen Baum.

Ein weiteres Beispiel für einen Syntaxbaum hat kann man anhand des folgenden Ausdruckes zeigen.

7.3 Konkrete und Abstrakte Syntax

Mit einer solchen Baumstruktur kann man relativ leicht und intuitiv den Syntax eines Programmes repräsentieren. Doch diese Struktur enthält immer noch viele Details, die dem Compiler nicht wirklich wichtig sind. Eine solche Syntax nennt man *konkret*.

Möchte man diese Daten weiterverarbeiten, zum Beispiel bei der Codegenerierung, so kann man vieles weglassen. Man spricht dann von einer *abstrakten Syntax*. Der Unterschied zwischen konkreter und abstrakter Syntax ist also, dass die erstgenannte das Programm fast Eins-zu-Eins wiedergibt, während letztere unwichtige Details weggelassen hat. Der konkrete Syntax hat keinen Einfluss auf die Semantik.