

Einführung in den Compilerbau

Einleitung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

WS 2018/19

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt



Embedded Systems & Applications



- ▶ Schnittstelle zwischen
 - ▶ Programmiersprache
 - ▶ Maschine

Programmiersprache Gut für Menschen handhabbar

- ▶ Smalltalk
- ▶ Java
- ▶ C++

Maschine Getrimmt auf

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

Entscheidet über dem Benutzer **zugängliche** Rechenleistung

Beispiel: Bildkompression auf Dothan CPU, 2 GHz

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



Hohe Ebene:
Smalltalk, Java, C++

Mittlere Ebene:
Assembler

Niedrige Ebene:
Maschinensprache

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

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

```
0110000100000110
0111001001000001
1011000100010010
1001000100000110
```



- ▶ 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

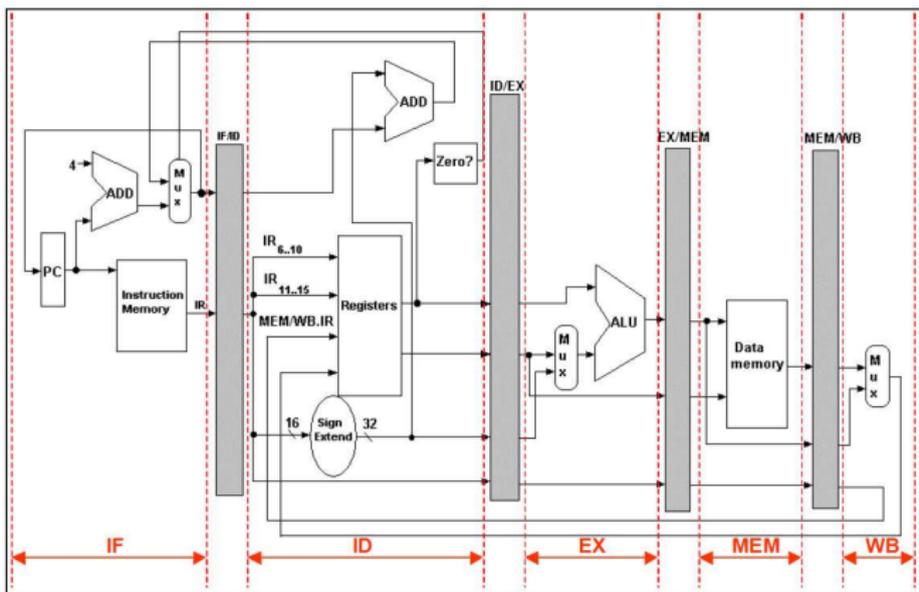


Auswirkungen der Zielmaschine

Einfach: Hennessy & Patterson DLX



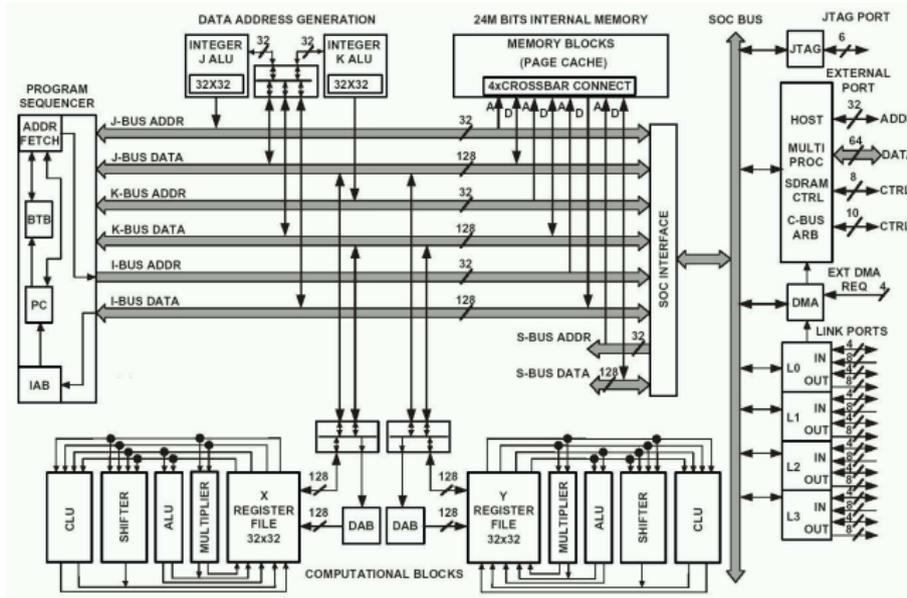
TECHNISCHE
UNIVERSITÄT
DARMSTADT



Komplizierter: Analog Devices TigerSHARC

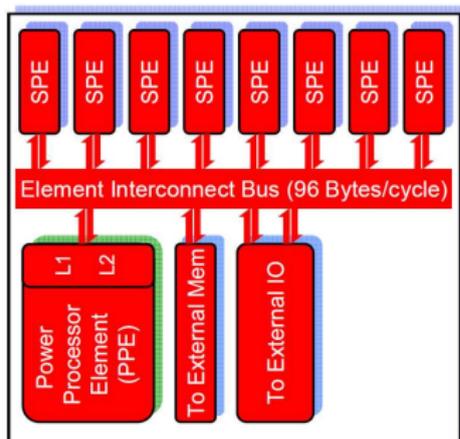


TECHNISCHE
UNIVERSITÄT
DARMSTADT

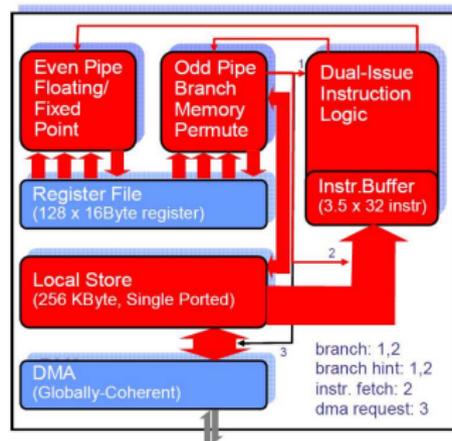


Problematisch: IBM/Sony Cell Processor

Übersicht



SPE





Je nach Anwendungsgebiet mehr oder weniger wichtig ...

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

... können häufig nur durch spezialisierte Prozessoren erfüllt werden

➡ **Benötigen passende Compiler**



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



- ▶ 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-CPU's
 - ▶ NVidia CUDA: GPU's
 - ▶ OpenCL: Heterogene Systeme (GPU's+CPU's, experimentell auch schon FPGA's)
- ▶ Aber noch wenig abstrakt
- ▶ *Keine* automatische Parallelisierung!



Kombination von verschiedenen Feldern der Informatik

- ▶ Theoretische Informatik (Parsing)
- ▶ Technische Informatik (Architektur der Zielmaschine)
- ▶ Praktische Informatik (Software-Engineering beim Aufbau des Compilers)

➡ **Programmieraufgaben betreffen alle Felder!**

Motivation: Compilerbau im Beruf



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Apple Computer

Job Title Sr Compiler Engineer
Posting Date 1/30/2006
Job Location
Description

Siemens

Job Title Principal Compiler Engineer
Posting Date
Job Location
Description

Job Title Compiler Engineer
Posting Date
Job Location
Description

Sony Computer Entertainment

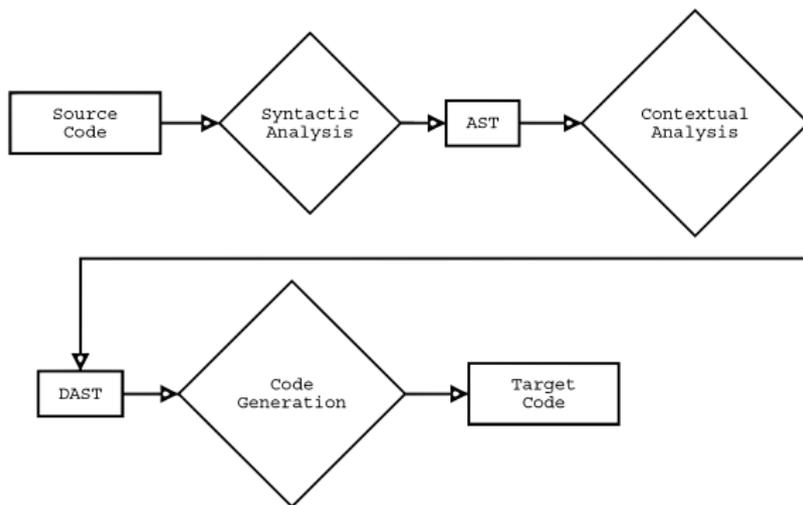
RWTH University

Job Title Research assistant/PhD candidate
Posting Date 11/15/2005
Job Location Aachen, Germany
Description The SSS division performs research and development on electronic design automation tools for embedded systems, e.g. future telecommunication and multimedia systems. Major research areas include compilers for



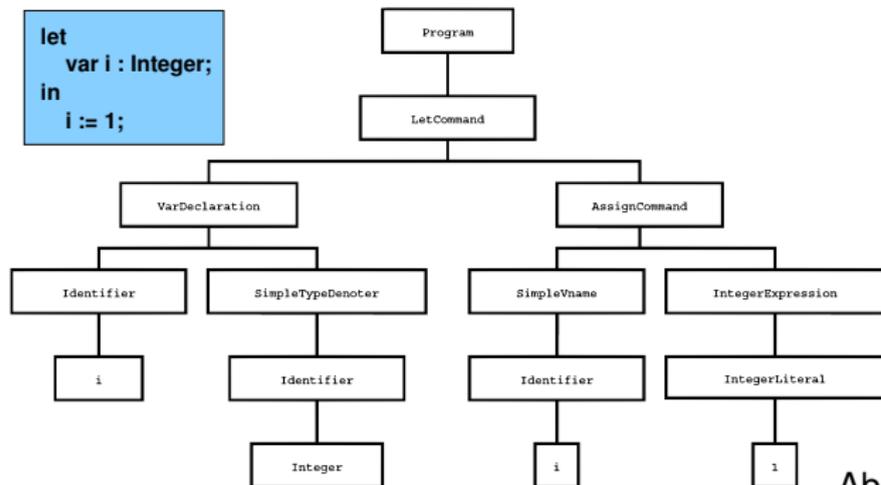
Aufbau von Compilern

Vorgehen: Bearbeitung in mehreren Phasen



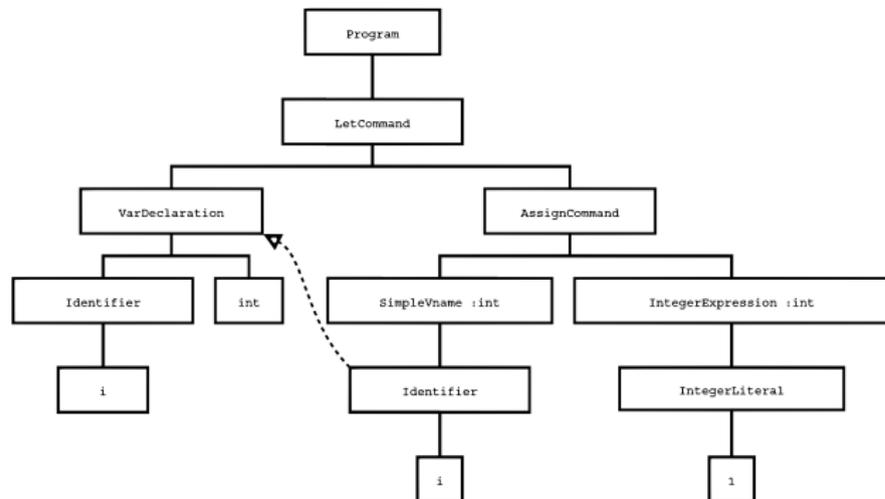
Zwischendarstellung(en) für den Informationsaustausch

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



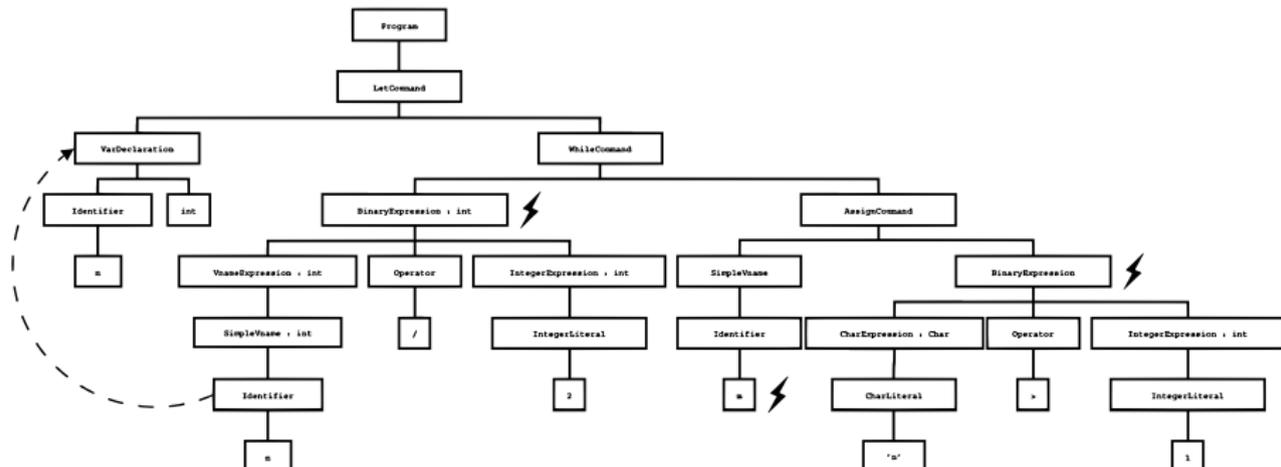
Abstrakter Syntaxbaum

- ▶ Ordne Variablen ihren Deklarationen zu
- ▶ Berechne Typen von Ausdrücken



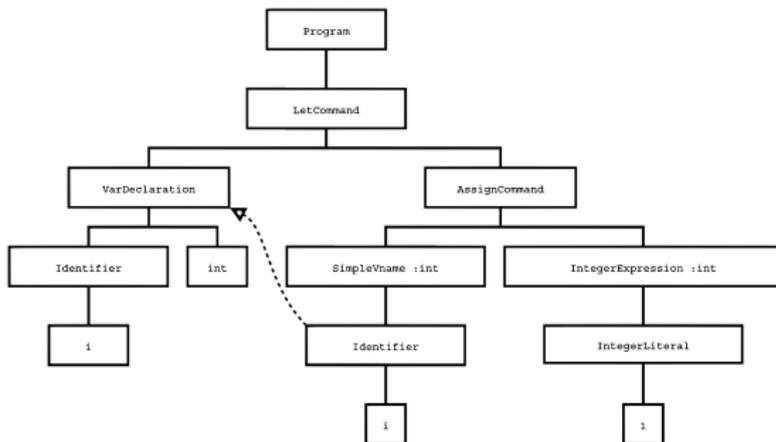
DAST: Dekorierter bzw.
annotierter AST

Erkenne Fehler in Variablen- und Typzuordnung





- ▶ 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



```
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     1      ; Ausfuehrung beenden
```



Optimierung



- ▶ 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



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



Wechsel auf anderen Foliensatz...



Grundlagen von Compilern: [Fast vollständig](#)

Programming Language Processors in Java

von David Watt und Deryck Brown, Prentice-Hall 2000

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



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



Überblick über Front-End¹ (ca. 3 Wochen)

- ▶ Lexing/Parsing
- ▶ Zwischendarstellungen

Überblick über Middle-End¹ (ca. 2 Wochen)

- ▶ Semantische / Kontextanalyse

Überblick über Back-End¹ (ca. 4 Wochen)

- ▶ Laufzeitorganisation
- ▶ Code-Erzeugung

¹: Diese Teile lehnen sich an an die Veranstaltungen

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



Weiterführende Themen

- ▶ Verwendung von Front-End-Generatoren (ca. 2-3 Wochen)
- ▶ Java Virtuelle Maschine (ca. 1-2 Wochen)



Syntax



Beschreibt die Satzstruktur von korrekten Programmen

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



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.



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



- ▶ 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



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
- ▶ Einige mögliche Vorgehensweisen
 1. Mathematische Mengennotation
 2. Reguläre Ausdrücke
 3. Kontextfreie Grammatik



Beispiele für die beschriebenen Zeichenketten

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

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



Erweitere Zeichenketten aus dem Alphabet um Operatoren

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

Beispiele

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



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).



Eine kontextfreie Grammatik besteht aus

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



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

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

$S ::= xS \quad (1)$

$S ::= yB \quad (2)$

$S ::= x \quad (3)$

$B ::= yB \quad (4)$

$B ::= y \quad \} \quad (5)$

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

$S \rightarrow xS \rightarrow xxS \rightarrow xxyB \rightarrow xxyyB \rightarrow xxyyy$

➡ Ja, da sie sich aus S herleiten läßt.



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

$S ::= xS \quad (6)$

$S ::= yB \quad (7)$

$S ::= x \quad (8)$

$B ::= yB \quad (9)$

$B ::= y \quad \} \quad (10)$

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

$S \rightarrow xS \rightarrow xyB \rightarrow ?$

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

Gegeben seien die Produktionen:

$$S ::= S+S \quad (11)$$

$$S ::= x \quad (12)$$

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

$$S \rightarrow S+S \rightarrow x+S \rightarrow x+S+S \rightarrow x+x+S \rightarrow x+x+x$$

Aber auch anders:

$$S \rightarrow S+S \rightarrow S+x \rightarrow S+S+x \rightarrow S+x+x \rightarrow x+x+x$$



Für sinnvolle praktische Anwendungen müssen CFGs **eindeutig** sein.

Eindeutige Produktionen für die gleiche CFG:

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

$$\mathbf{S} ::= \mathbf{x} \quad (14)$$



(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

```
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
```

Lokale Deklarationen

Konstante (häßliches "~")!

Variable kann in `getint` verändert werden

Folge von Anweisungen zwischen `begin/end`

`else` ist erforderlich (darf aber leer sein)



```
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 Digit
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
```

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

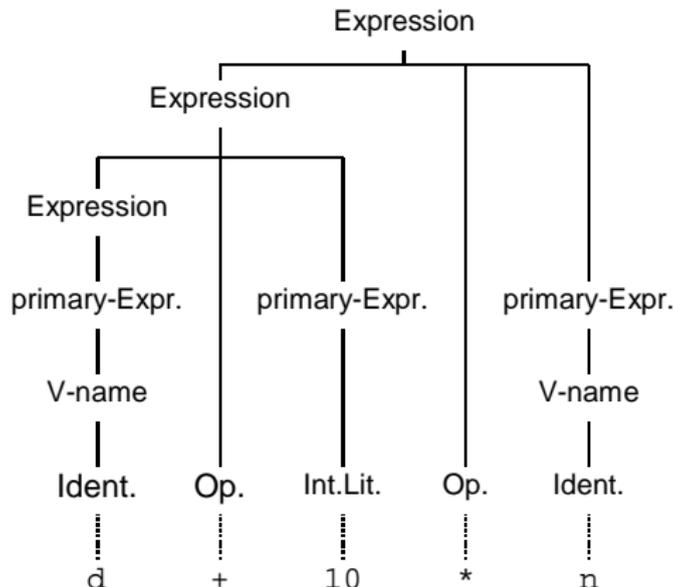


Ein Syntaxbaum ist ein geordneter, markierter Baum bei dem

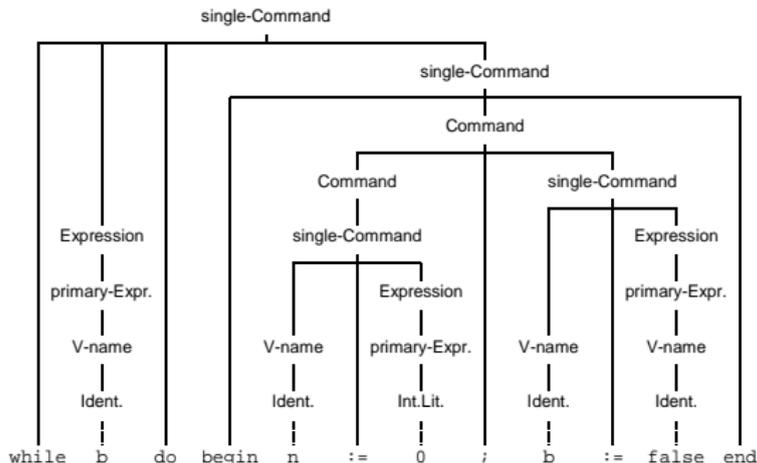
- ▶ ... die Blätter mit Terminalsymbolen markiert sind
- ▶ ... die inneren Knoten mit Nicht-Terminalsymbolen markiert sind
- ▶ ... jeder innere Knoten \mathbf{N} (von links nach rechts) die Kinder $\mathbf{X}_1, \dots, \mathbf{X}_n$ hat, entsprechend der Produktion $\mathbf{N} := \mathbf{X}_1 \dots \mathbf{X}_n$

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

Expression-Baum für $d + 10 * n$



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



- ▶ 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!



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



- ▶ 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!



Command

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



Expression

::= Integer-Literal

| V-name

| Operator Expression

| Expression Op Expression

V-name ::= Identifier

IntegerExp

VnameExp

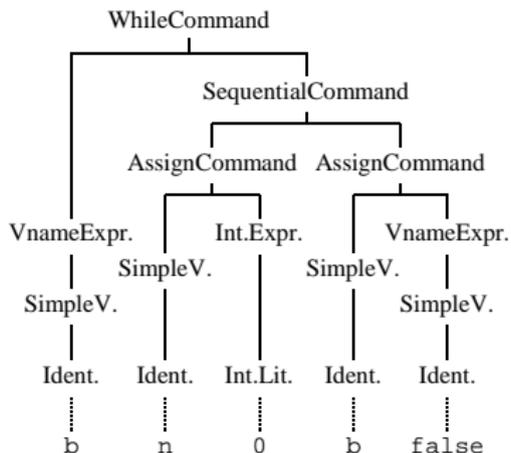
UnaryExp

BinaryExp

SimpleVName



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





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



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

Syntaktische Korrektheit reicht nicht aus für sinnvolle Übersetzung

Geltungsbereiche (Scope)

- ▶ Betreffen *Sichtbarkeit* von Bezeichnern
- ▶ 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

```
let
  const m ~ 2;
  var n: Integer
in begin
  ..
  n := m*2;
  ..
end
```

Deklaration von n:
Bindung

Benutzung von von n:
Verwendung

??

```
let
  var n: Integer
in begin
  ..
  n := m*2;
end
```

Falls im Geltungsbereich der
Verwendung vom m keine Bindung
von m existiert: Fehler!

Verwendung von m



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)

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

Anweisungen

... werden **ausgeführt**. Mögliche Seiteneffekte:

- ▶ Ändern der Werte von Variablen
- ▶ Ein-/Ausgabeoperationen

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



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 $E_1 \text{ op } E_2$

1. Der Ausdruck E_1 wird evaluiert um einen Wert v_1 zu erhalten
2. Der Ausdruck 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



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



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



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