Einführung in den Compilerbau

Niclas Kusenbach
LaTeX version: SCHOUTER

Table of Contents

Contents

1 Einführung in den Compilerbau

1.1 Grundlagen und Motivation

Ein **Compiler** agiert als essentielle Schnittstelle zwischen der für Menschen lesbaren Programmiersprache und der hardwarenahen Maschinensprache.

Rolle des Compilers

Der Compiler übersetzt ein in einer Hochsprache (z. B. Java, C++, Smalltalk) verfasstes Programm in eine Sprache, die von der Maschine ausgeführt werden kann.

- **Programmiersprache:** Optimiert auf menschliche Handhabung (Lesbarkeit, Abstraktion).
- **Maschine:** Optimiert auf Ausführungsgeschwindigkeit, Chip-Fläche und Energieverbrauch.

Moderne Hardware-Architekturen (z. B. VLIW, Multi-Core, Heterogene Systeme wie Cell Prozessoren oder GPUs) sind extrem komplex und schwer manuell effizient zu programmieren. Der Compiler überbrückt diese Lücke durch verschiedene Abstraktionsebenen.

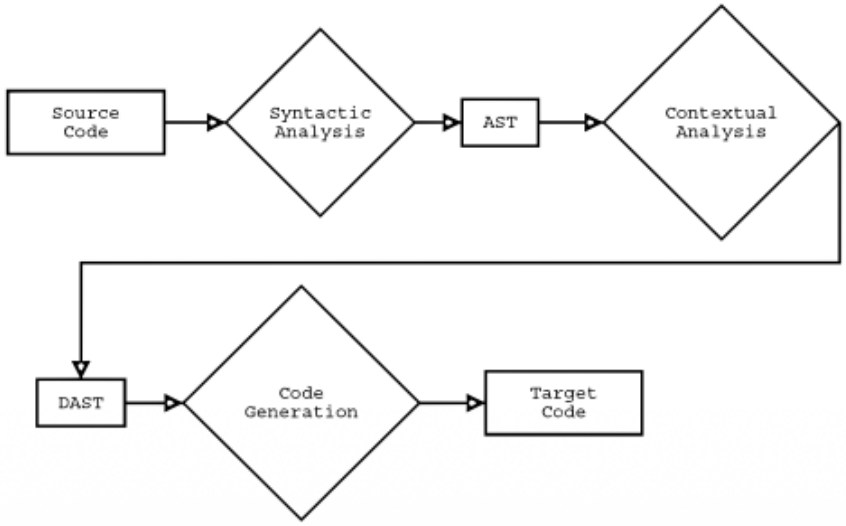
1.1.1 Abstraktionsebenen

Die Übersetzung erfolgt schrittweise über mehrere Ebenen, wobei Details (wie Registerzuweisung oder spezifische Maschineninstruktionen) durch Synthese hinzugefügt werden.

1. **Hohe Ebene:** Hochsprachen (Java, C++).
2. **Mittlere Ebene:** Assembler (symbolische Maschinenbefehle).
3. **Niedrige Ebene:** Maschinensprache (Binärcode, z. B. `0110...`).

1.2 Aufbau eines Compilers

Die Verarbeitung eines Programms erfolgt in mehreren Phasen, die den Informationsaustausch über **Zwischendarstellungen** (Intermediate Representations, IR) organisieren.



1.2.1 Das Phasen-Modell

1. **Syntactic Analysis (Syntaxanalyse):** Überprüfung der grammatikalischen Korrektheit und Erstellung eines Abstrakten Syntaxbaums (AST).

2. **Contextual Analysis (Kontextanalyse):** Überprüfung von Geltungsbereichen und Datentypen. Das Ergebnis ist ein dekorierter AST (DAST).
3. **Code Generation (Code-Erzeugung):** Übersetzung des DAST in den Zielcode (Maschinensprache, Assembler oder C).

1.2.2 Architektur-Gliederung

Ein Compiler wird typischerweise in drei Hauptkomponenten unterteilt:

Compiler-Komponenten

- **Front-End:** Zuständig für die Analyse (syntaktisch und kontextuell). Es ist abhängig von der Quellsprache, aber weitgehend unabhängig von der Zielmaschine.
- **Middle-End:** Zuständig für die Transformation und Optimierung von Zwischendarstellungen (IR). Hier finden maschinenunabhängige Optimierungen statt.
- **Back-End:** Zuständig für die Code-Erzeugung. Es ist stark abhängig von der Zielarchitektur (Instruktionssatz, Registeranzahl).

1.2.3 Optimierung

Ein **optimierender Compiler** versucht, den Code hinsichtlich bestimmter Gütekriterien (Laufzeit, Speicherbedarf, Energie) zu verbessern. Beispiele für Optimierungstechniken:

- **Constant Folding:** Berechnung konstanter Ausdrücke zur Compile-Zeit.

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

- **Common-Subexpression Elimination:** Vermeidung wiederholter Berechnungen gleicher Teilausdrücke.

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

- **Strength Reduction:** Ersetzen teurer Operationen durch günstigere (z. B. Multiplikation durch Addition in Schleifen).
- **Loop-invariant Code Motion:** Verschieben von Berechnungen, die sich innerhalb einer Schleife nicht ändern, vor die Schleife.

1.3 Sprachbeschreibung

Die Definition einer Programmiersprache ruht auf drei Säulen:

1. **Syntax:** Welche Zeichenfolgen bilden ein korrektes Programm? (Struktur, Grammatik).
2. **Kontextuelle Einschränkungen:** Welche Regeln gelten für Gültigkeitsbereiche und Typen? (z. B. „Variablen müssen vor Gebrauch deklariert werden“).
3. **Semantik:** Was bedeutet das Programm? (Verhalten zur Laufzeit).

Diese Aspekte können entweder *informal* (natürliche Sprache) oder *formal* (mathematische Notation, Grammatiken) spezifiziert werden.

1.4 Syntaxspezifikation

Eine Sprache ist formal definiert als eine Menge von Zeichenketten aus einem Alphabet. Da Programmiersprachen meist unendlich viele gültige Programme erlauben, ist eine Aufzählung unmöglich.

1.4.1 Reguläre Ausdrücke (Regular Expressions, RE)

Reguläre Ausdrücke erweitern Zeichenketten um Operatoren zur Musterbeschreibung.

- | (Alternative): $a|b$ (a oder b).
- $*$ (Kleene-Stern): Beliebige Wiederholung (0 oder mehr).
- ϵ (Epsilon): Das leere Wort.

Grenzen: Reguläre Ausdrücke sind nicht mächtig genug, um die geschachtelte Struktur (z. B. Klammerungsebenen) komplexer Programmiersprachen vollständig zu beschreiben. Sie werden jedoch häufig für den **Scanner** (lexikalische Analyse) verwendet.

1.4.2 Kontextfreie Grammatiken (CFG)

Zur Beschreibung der Syntax von Programmiersprachen werden kontextfreie Grammatiken verwendet.

Bestandteile einer CFG

- **Terminalsymbole (T):** Die eigentlichen Zeichen/Token des Alphabets.
- **Nicht-Terminalsymbole (N):** Platzhalter für syntaktische Strukturen.
- **Startsymbol (S):** Ein Element aus N.
- **Produktionen (P):** Regeln, wie Nicht-Terminale durch eine Folge von Terminalen und Nicht-Terminalen ersetzt werden können.

Notationen:

- **BNF (Backus-Naur-Form):** $N ::=$ Zeichenkette aus T und N.
- **EBNF (Extended BNF):** Erlaubt reguläre Ausdrücke auf der rechten Seite der Produktion (kompakter).

1.4.3 Ableitung und Mehrdeutigkeit

Ein Satz (gültiges Programm) wird durch wiederholtes Anwenden der Produktionen ausgehend vom Startsymbol hergeleitet.Eine Grammatik ist **mehrdeutig** (ambiguous), wenn es für eine Zeichenkette mehr als einen möglichen Ableitungsbaum gibt.

- *Beispiel:* $S ::= S + S \mid x$.
- Der Ausdruck $x + x + x$ kann unterschiedlich geklammert interpretiert werden: $(x + x) + x$ oder $x + (x + x)$.
- Für Compiler muss die Grammatik **eindeutig** sein (z. B. durch Festlegung von Vorrangregeln oder Umstrukturierung der Grammatik).

1.5 Syntaxbäume

1.5.1 Konkreter Syntaxbaum

Ein Syntaxbaum (Parse Tree) repräsentiert die grammatikalische Struktur gemäss der CFG.

- **Blätter:** Terminalsymbole.
- **Innere Knoten:** Nicht-Terminalsymbole.
- Die Kinder eines Knotens entsprechen der rechten Seite der angewendeten Produktion.

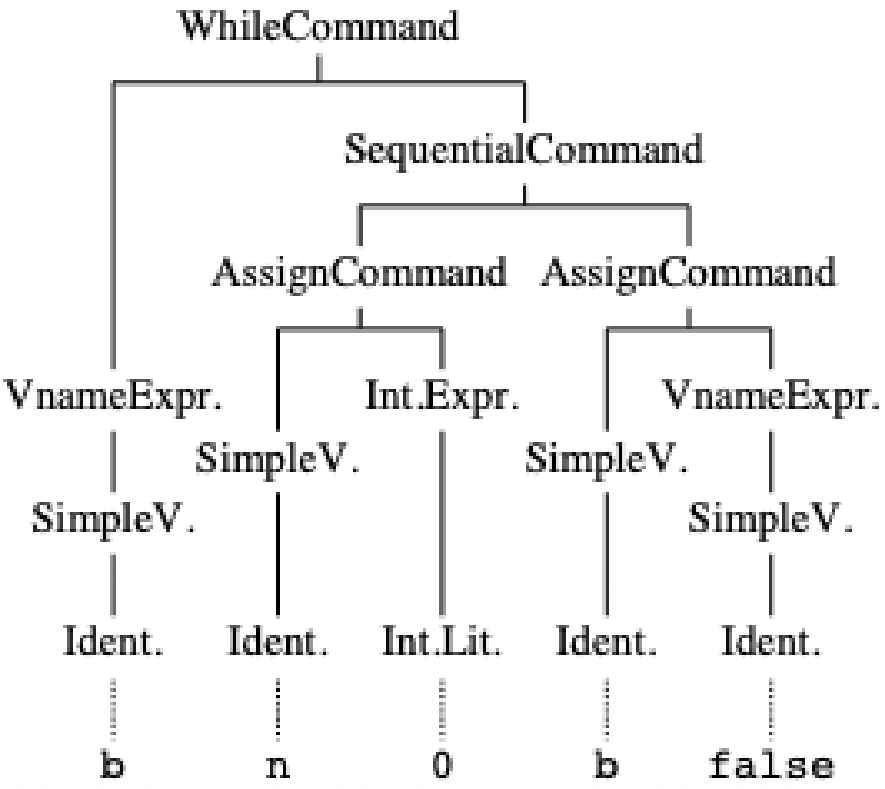
1.5.2 Abstrakte Syntax (AST)

Die konkrete Syntax enthält Details, die für das Verständnis des Programms irrelevant sind (z. B. Schlüsselwörter wie `begin`, `end`, Semikolons). Der **Abstrakte Syntaxbaum** (AST) reduziert die Darstellung auf die essentielle Information.

Unterschied CST vs. AST

- **Konkrete Syntax (CST):** Wichtig für das Parsen und das Erstellen korrekter Quelltexte. Enthält „Syntaxzucker“.
- **Abstrakte Syntax (AST):** Modelliert die logische Struktur (Subphrasen). Dient als interne Zwischendarstellung (High-Level IR) für Analysen und Code-Erzeugung.

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



1.6 Kontextuelle Analyse

Nach der Syntaxanalyse folgt die Prüfung der kontextuellen Korrektheit. Dies geschieht oft auf dem AST.

1.6.1 Geltungsbereiche (Scope)

Hierbei wird die Sichtbarkeit von Bezeichnern (Identifiern) geprüft.

- **Bindendes Auftreten:** Die Deklaration einer Variable.
- **Verwendendes Auftreten:** Die Benutzung der Variable im Code.
- **Aufgabe:** Jede Verwendung muss eindeutig einer gültigen Deklaration zugeordnet werden (Identifikation).

1.6.2 Typisierung

Überprüfung der Verträglichkeit von Operationen und Datenwerten.

- **Statische Typisierung:** Typen werden zur Compile-Zeit geprüft (Thema dieser Vorlesung).
- **Typregeln:** Definieren Anforderungen (z. B. „Bedingung in `if` muss `boolean` sein“) und Ergebnistypen (z. B. „`int` + `int` ergibt `int`“).

1.7 Semantik

Die Semantik beschreibt die Bedeutung des Programms bei der Ausführung.

- **Operationelle Semantik:** Beschreibt die Schritte, die auf einer abstrakten Maschine ausgeführt werden.
- **Denotationale Semantik:** Bildet Eingaben mathematisch auf Ausgaben ab.

In der Vorlesung wird zwischen verschiedenen semantischen Aktionen unterschieden:

1. **Anweisungen (Commands):** Werden *ausgeführt* (executed). Sie haben meist Seiteneffekte (z. B. Zustandsänderung von Variablen, I/O).
2. **Ausdrücke (Expressions):** Werden *evaluiert* (evaluated), um einen Wert zu liefern.
3. **Deklarationen:** Werden *elaboriert* (elaborated), um Bindungen zu erzeugen und Speicherplatz bereitzustellen.

Beispiel Semantik einer Zuweisung $V := E$

1. Der Ausdruck E wird evaluiert, um einen Wert v zu erhalten.
2. Der Wert v wird der Variable V zugewiesen (in den Speicher geschrieben).

2 Syntaktische Analyse

Diese Sektion behandelt den Prozess der Übersetzung von Quellcode in Maschinencode, mit besonderem Fokus auf die Phasen der syntaktischen Analyse (Scanner und Parser), Grammatiktransformationen und die Implementierung von Parsern mittels rekursivem Abstieg.

2.1 Einordnung und Struktur von Compilern

Die Übersetzung erfolgt in mehreren Phasen, die logisch voneinander getrennt sind. Physikalisch können diese in einem oder mehreren Durchgängen (Passes) organisiert sein.

Terminologie

- **Phase:** Ein logischer Schritt im Übersetzungsprozess (z.B. Syntaxanalyse, Codegenerierung).
- **Pass (Durchgang):** Ein kompletter Durchlauf über den Quelltext oder eine Zwischenrepräsentation (Intermediate Representation, IR). Ein Pass kann mehrere Phasen beinhalten.

2.1.1 Vergleich: Ein-Pass vs. Multi-Pass Compiler

Kriterium	Ein-Pass Compiler	Multi-Pass Compiler
Arbeitsweise	Führt Syntaxanalyse, Kontextanalyse und Codegenerierung verschränkt aus. Keine echte IR.	Mehrere Durchläufe. Datenübergabe zwischen Passes erfolgt über IR (z.B. AST).
Laufzeit	+ Schnell (weniger I/O-Overhead).	- Langsamer durch mehrfaches Lesen/Schreiben der IR.
Speicher	+ Geringer Speicherbedarf (gut für große Programme bei wenig RAM).	+ Besser für kleine Programme; bei großen Programmen speicherintensiv durch IR.
Modularität	– Schlecht trennbar, "Spaghetti-Code".	+ Klare Trennung der Belange.
Flexibilität	- Starr.	+ Austauschbare Backends/Frontends möglich.
Optimierung	– Nur lokale Optimierungen möglich.	+ Globale Optimierungen auf IR möglich.
Kontext	Definitionen müssen oft *vor* der Verwendung stehen (z.B. Pascal).	Auflösung von Vorwärtsreferenzen (z.B. Java) problemlos möglich.

2.2 Ablauf der Syntaxanalyse

Die Syntaxanalyse ist der Kern des Frontends und transformiert den linearen Zeichenstrom in eine strukturierte Repräsentation.

1. **Scanner (Lexikalische Analyse):**
   - Eingabe: Zeichenfolge (Source Code).
   - Aufgabe: Gruppierung von Zeichen zu **Tokens** (atomare Symbole wie Schlüsselwörter, Bezeichner, Literale).
   - Filterung: Entfernt Whitespace und Kommentare.
2. **Parser (Syntaktische Analyse):**
   - Eingabe: Token-Stream.
   - Aufgabe: Überprüfung der grammatikalischen Struktur gemäß einer kontextfreien Grammatik (CFG).

- Ausgabe: Abstract Syntax Tree (AST) oder Fehlermeldungen.

Token

Ein Token ist ein Tupel bestehend aus:
- **Kind:** Die Art des Tokens (z.B. `IDENTIFIER`, `INTLITERAL`, `IF`, `PLUS`).
- **Spelling:** Der tatsächliche Textinhalt (z.B. "x", "42", "if", "+").
- **Position:** Zeile und Spalte im Quelltext (für Fehlermeldungen).

2.3 Grammatiken und Transformationen

Die Syntax von Programmiersprachen wird meist durch kontextfreie Grammatiken (CFG) spezifiziert.

2.3.1 Notation

- **BNF (Backus-Naur-Form):** Klassische Notation.
- **EBNF (Extended BNF):** Erlaubt reguläre Ausdrücke auf der rechten Seite für kompaktere Schreibweise.
  - `(...)`: Gruppierung
  - `|`: Alternative
  - `[...]` oder `?`: Optional (0 oder 1 Mal)
  - `{...}` oder `*`: Wiederholung (0 bis n Mal)

2.3.2 Notwendige Grammatik-Transformationen für Parser

Für bestimmte Parsing-Techniken (insbesondere Top-Down / Recursive Descent) muss die Grammatik angepasst werden, ohne die definierte Sprache zu ändern.

1. Linksausklammern (Left Factoring) Problem: Der Parser kann sich nicht entscheiden, welche Produktion er wählen soll, da mehrere mit demselben Symbol beginnen.
Regel: $X ::= \alpha\beta \mid \alpha\gamma \quad \Rightarrow \quad X ::= \alpha(\beta \mid \gamma)$
Beispiel:

$$\begin{aligned} Cmd ::= & \texttt{ if } E \texttt{ then } C \\ & \mid \texttt{if } E \texttt{ then } C \texttt{ else } C \end{aligned}$$

wird zu:

$$Cmd ::= \texttt{if } E \texttt{ then } C \ (\epsilon \mid \texttt{else } C)$$

2. Beseitigung von Linksrekursion Problem: Ein Top-Down-Parser gerät in eine Endlosschleife, wenn ein Nicht-Terminal sich selbst als erstes Symbol aufruft ($N \Rightarrow N\alpha$).

Direkte Linksrekursion: $N ::= N\alpha \mid \beta$

Lösung (Transformation in Rechtsrekursion oder Iteration):

$$N ::= \beta(\alpha)^*$$

bzw. formal in BNF:

$$\begin{aligned} N &::= \beta N' \\ N' &::= \alpha N' \mid \epsilon \end{aligned}$$

	Top-Down (z.B. LL)	Bottom-Up (z.B. LR)
Richtung	Wurzel → Blätter	Blätter → Wurzel
Vorgehen	Expandiert das am weitesten links stehende Nicht-Terminal.	"Shift" (Einlesen) und "Reduce" (Ersetzen der rechten Seite durch linke Seite).
Lookahead	Wählt Produktion anhand der nächsten k Token.	Entscheidungen basieren auf Stack-Zustand und Lookahead.
Eignung	Intuitive Implementierung von Hand (Rekursiver Abstieg).	Mächtiger, handhabt Linksrekursion, meist generiert (Yacc, Bison).

2.4 Parsing-Strategien

Man unterscheidet zwei grundlegende Herangehensweisen, um den Ableitungsbaum (Parse Tree) zu konstruieren.

LL(k)

Eine Grammatik ist $LL(k)$, wenn der Parser beim Lesen von links nach rechts (**L**) und beim Konstruieren einer Linksanleitung (**L**) mit k Symbolen Vorausschau (Lookahead) immer eindeutig die nächste Produktion bestimmen kann.

2.5 Rekursiver Abstieg (Recursive Descent)

Dies ist eine direkte Implementierung eines Top-Down-Parsers.

- Jedes Nicht-Terminal der Grammatik wird zu einer Methode (z.B. `parseStatement()`).
- Die rechte Seite der Produktion bestimmt den Rumpf der Methode.
- Terminale werden mit dem aktuellen Token verglichen (`accept`).
- Nicht-Terminale führen zu rekursiven Aufrufen der entsprechenden Methoden.

Struktur einer Parse-Methode:

```
void parseN() {
  if (currentToken in Starters[Alternative1]) {
    parseAlternative1();
  } else if (currentToken in Starters[Alternative2]) {
    parseAlternative2();
  } else {
    error("Syntax Error");
  }
}
```

2.6 LL(1)-Analyse und Entscheidungsmengen

Damit ein rekursiver Abstieg deterministisch funktioniert (ohne Backtracking), muss die Grammatik die LL(1)-Eigenschaft erfüllen. Dazu werden Hilfsmengen berechnet.

2.6.1 Starters (First) Menge

$Starters[[X]]$ ist die Menge aller Terminalsymbole, mit denen ein aus X ableitbarer Satz beginnen kann.

- $Starters[[t]] = \{t\}$ (für Terminal t)
- $Starters[[\epsilon]] = \emptyset$ (Achtung: ϵ wird oft separat behandelt)
- $Starters[[XY]] = Starters[[X]]$, falls X nicht zu ϵ werden kann. Sonst $Starters[[X]] \cup Starters[[Y]]$.
- $Starters[[X|Y]] = Starters[[X]] \cup Starters[[Y]]$

2.6.2 Follow Menge

$Follow[[N]]$ ist die Menge aller Terminalsymbole, die in irgendeiner Satzform direkt *nach* dem Nicht-Terminal N stehen können.

- Wird benötigt, wenn eine Produktion zu ϵ (leer) abgeleitet werden kann.
- Regel: Wenn $A ::= \alpha B \beta$, dann ist alles in $Starters[[\beta]]$ auch in $Follow[[B]]$.
- Wenn $\beta \to \epsilon$ (oder B am Ende steht: $A ::= \alpha B$), dann ist alles in $Follow[[A]]$ auch in $Follow[[B]]$.

2.6.3 Director Sets (Entscheidungsmengen) und LL(1)-Bedingungen

Um zwischen Alternativen $A ::= \alpha \mid \beta$ zu wählen, nutzen wir das **Director Set** Dir.

$$Dir[[\alpha]] = \begin{cases} Starters[[\alpha]] & \text{falls } \epsilon \notin Starters[[\alpha]] \\ Starters[[\alpha]] \cup Follow[[A]] & \text{falls } \epsilon \in Starters[[\alpha]] \end{cases}$$

Bedingung für LL(1): Für alle Produktionen $A ::= \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ muss gelten:

$$Dir[[\alpha_i]] \cap Dir[[\alpha_j]] = \emptyset \quad \text{für alle } i \neq j$$

Das bedeutet:

- Die Anfangssymbole der Alternativen müssen disjunkt sein.
- Falls eine Alternative leer sein kann (ϵ), darf keines der Symbole, die *nach* dem Nicht-Terminal folgen können, in den Anfangsmengen der anderen Alternativen enthalten sein.

2.7 Abstract Syntax Tree (AST)

Der AST ist eine komprimierte Version des Parse Trees. Er enthält keine unnötigen Details der konkreten Syntax (wie Klammern, Semikolons oder Hilfs-Nicht-Terminale aus der Grammatiktransformation), sondern repräsentiert die logische Struktur des Programms.

Implementierung:

- Eine abstrakte Basisklasse `AST`.
- Unterklassen für Sprachkonstrukte (z.B. `Command`, `Expression`, `Declaration`).
- Konkrete Klassen für Varianten (z.B. `IfCmd`, `WhileCmd`, `BinaryExpr`).
- **Aufbau:** Die Parse-Methoden werden von `void` auf den Rückgabetyp `AST` geändert. Sie erzeugen Knoten und geben diese an den Aufrufer zurück ("bottom-up" Aufbau während der Rekursion).

2.8 Lexikalische Analyse (Scanner-Implementierung)

Der Scanner (Lexer) ist oft als endlicher Automat (Finite State Machine) realisiert oder wird manuell implementiert.

2.8.1 Aufgaben

1. Lesen der Eingabezeichen.
2. Überlesen von "Whitespace" (Leerzeichen, Tabs, Newlines) und Kommentaren.
3. Erkennen des längstmöglichen passenden Tokens (Longest Match).
4. Unterscheidung zwischen Bezeichnern (Identifiers) und Schlüsselwörtern (Keywords).

2.8.2 Behandlung von Schlüsselwörtern

Da Schlüsselwörter (z.B. `if`, `while`) lexikalisch oft wie Bezeichner aussehen, werden sie zunächst als `IDENTIFIER` gescannt.

Best Practice:

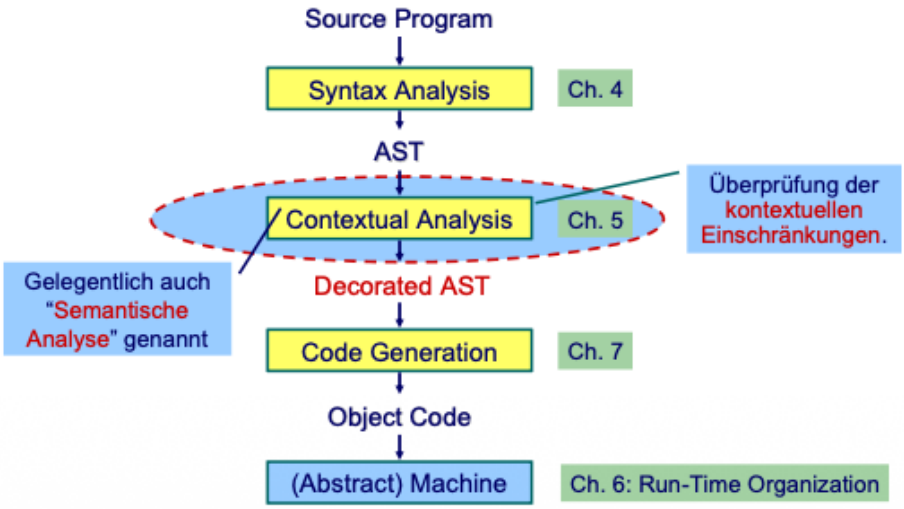
- Scanne das Wort als Identifier.
- Prüfe in einer Hash-Map/Tabelle, ob der Text ein reserviertes Wort ist.
- Wenn ja: Ändere die Token-Art (z.B. von `IDENTIFIER` zu `IF`).

2.8.3 Schnittstelle zum Parser

- `scan()`: Liefert das nächste Token.
- `currentKind`: Art des aktuellen Tokens.
- `currentSpelling`: Textinhalt des aktuellen Tokens.

3 Kontextuelle Analyse

Die kontextuelle Analyse ist die Phase im Compilerbau, die zwischen der Syntaxanalyse (Parsing) und der Code-Generierung steht. Sie dient dazu, die Gültigkeit des Programms über die reine Syntax hinaus zu überprüfen und den abstrakten Syntaxbaum (AST) mit semantischen Informationen zu angereichern.



Ziel der kontextuellen Analyse

Eingabe: Ein Abstrakter Syntaxbaum (AST).
Ausgabe: Ein **dekorierter AST** (angereichert mit Typinformationen und Bindungen) oder Fehlermeldungen.
Aufgaben: Überprüfung der **Geltungsbereiche** (Identification) und der **Typregeln** (Type Checking).

3.1 Kontextuelle Einschränkungen

Die Syntaxanalyse prüft lediglich die grammatikalische Struktur. Die kontextuelle Analyse prüft Bedingungen, die vom Kontext abhängen:

- **Geltungsbereiche (Scope):** Jeder verwendete Bezeichner (Variable, Funktion) muss korrekt deklariert (gebunden) sein.
- **Typen:** Operationen müssen auf kompatiblen Datentypen ausgeführt werden (z. B. `if`-Bedingung muss `Boolean` sein).

3.1.1 Geltungsbereiche (Identification)

Es wird zwischen der *Deklaration* (Bindung) und der *Benutzung* (Verwendung) eines Namens unterschieden.

Regel für Bezeichner

Falls im Geltungsbereich der Verwendung eines Bezeichners n keine Bindung von n existiert $\rightarrow$ Fehler.

3.1.2 Typprüfung

Wir betrachten hier die **statische Typisierung**, bei der Typen zur Compile-Zeit geprüft werden.

- Jeder Wert hat einen Typ.

- Jede Operation hat Anforderungen an die Typen ihrer Operanden und liefert einen Ergebnistyp.
- **Vorteile:** Fehlervermeidung („eckiger Kreis“) und Laufzeitoptimierung (keine Typchecks zur Laufzeit nötig).

3.2 Symboltabellen (Identification Tables)

Um Namen (Strings) effizient ihren Attributen (Typ, Art, Adresse) zuzuordnen, wird eine Symboltabelle verwendet. Dies vermeidet langsames Suchen im AST.

3.2.1 Struktur von Geltungsbereichen

1. **Monolithische Blockstruktur:**
   - Nur ein globaler Geltungsbereich (z. B. BASIC).
   - Jeder Bezeichner darf nur einmal deklariert werden.
2. **Flache Blockstruktur:**
   - Globale und lokale Ebene (z. B. FORTRAN).
   - Lokale Deklarationen verschatten globale, werden aber nach Blockende verworfen.
3. **Verschachtelte Blockstruktur (Nested Scopes):**
   - Beliebige Schachtelungstiefe (z. B. Pascal, Java, Ada).
   - **Regel:** Kein Bezeichner darf im *selben* Block mehrfach deklariert werden. Verwendete Bezeichner müssen im lokalen oder einem umschließenden Block deklariert sein.

3.2.2 Implementierung der Symboltabelle

Eine effiziente Implementierung für verschachtelte Blöcke verwendet eine Hash-Tabelle, die Stacks enthält.

- **Datenstruktur:** `Map<String, Stack<Attribute>> idents`
- Der Schlüssel ist der Bezeichnername.
- Der Wert ist ein Stack von Attributen. Oben auf dem Stack liegt immer die Deklaration der tiefsten (aktuellsten) Verschachtelungsebene.
- Zusätzlich gibt es einen `Stack<List<String>> scopes`, der speichert, welche Bezeichner zu welchem Scope gehören, um sie beim Verlassen des Scopes (`closeScope`) wieder aus der Map zu entfernen.

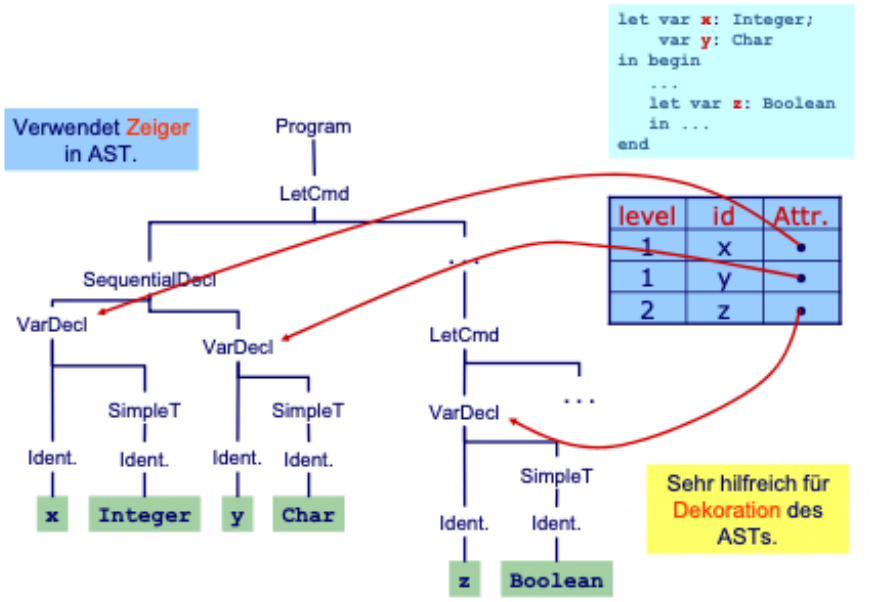
Operationen der Symboltabelle

- `enter(id, attr)`: Fügt eine neue Bindung hinzu.
- `retrieve(id)`: Liefert das Attribut der innersten sichtbaren Deklaration (oberstes Element im Stack).
- `openScope()`: Öffnet einen neuen Geltungsbereich.
- `closeScope()`: Entfernt alle Einträge des aktuellen Geltungsbereichs.

3.3 Attribute und AST-Dekoration

Anstatt alle Informationen (Art, Typ, etc.) explizit in komplexen Attribut-Objekten zu speichern, nutzt man im Compilerbau oft den AST selbst.

- **Idee:** Im AST stehen bei der Deklaration bereits alle Informationen.
- **Umsetzung:** Die Symboltabelle speichert Verweise (Zeiger) auf die Deklarations-Knoten im AST.
- **Dekoration:** Verwendungsstellen im AST (z. B. `VnameExpr`) erhalten einen Zeiger auf ihre Deklarationsstelle. Ausdrucksknoten (`Expression`) erhalten ein Feld für ihren berechneten Typ.



3.4 Implementierung: Das Visitor Pattern

Für die Durchquerung des AST zur Typprüfung und Code-Generierung eignet sich das **Visitor Pattern**. Es trennt die Datenstruktur (AST) von den Operationen (Check, Encode).

3.4.1 Standard Visitor

Das Interface definiert für jeden Knotentyp eine Methode:

```
public interface Visitor<RetTy, ArgTy> {
    RetTy visitProgram(Program p, ArgTy arg);
    RetTy visitAssignCommand(AssignCommand c, ArgTy arg);
    // ... für alle AST-Knoten
}
```

Jede AST-Klasse implementiert eine `visit`-Methode, die den Visitor aufruft (Double Dispatch).

3.4.2 Herausforderung und Lösung in Triangle

Da verschiedene AST-Knoten unterschiedliche Rückgabetypen bei der Analyse benötigen (z. B. liefert ein `Command` nichts zurück, eine `Expression` aber einen Typ), ist ein einziger generischer Visitor oft unhandlich.

Lösung: Spezialisierte Visitors (Checkers), die von einer Basisklasse (`VisitorBase`) erben.

- `CommandChecker`: Überprüft Anweisungen (Return: `Void`).
- `ExpressionChecker`: Überprüft Ausdrücke (Return: `TypeDenoter`).
- `DeclarationChecker`: Trägt Deklarationen in die Symboltabelle ein.

3.5 Algorithmus der Kontextanalyse

Die Analyse erfolgt meist als **Tiefensuche** (Depth-First Traversal) von links nach rechts durch den AST. Identifikation und Typprüfung werden oft in einem Pass kombiniert (möglich in Sprachen wie Triangle, wo „Definition vor Verwendung“ gilt).

3.5.1 Vorgehen für spezifische Knoten

1. **Variablendeklaration (`VarDecl`):**
   1. Prüfe den Typ-Teilbaum (validiere Typnamen).
   2. Prüfe, ob der Bezeichner im aktuellen Scope bereits existiert (Duplikat-Check).
   3. Trage Bezeichner und Verweis auf `VarDecl`-Knoten in Symboltabelle ein.
2. **Zuweisung (`AssignCmd` $V := E$):**
   1. Besuche V (Variable): Ermittle Typ T_V und prüfe, ob es eine Variable (keine Konstante) ist.
   2. Besuche E (Expression): Ermittle Typ T_E.
   3. Prüfe: $T_V \equiv T_E$ (Typkompatibilität).
3. **Bedingung (`IfCmd` if E then C_1 else C_2):**
   1. Besuche E: Typ muss `Boolean` sein.
   2. Besuche C_1 und C_2 rekursiv.
4. **Binärer Ausdruck (`BinaryExpr` E_1 *op* E_2):**
   1. Bestimme Typen T_1 von E_1 und T_2 von E_2.
   2. Suche Operator *op* in Symboltabelle.
   3. Prüfe, ob *op* für (T_1, T_2) definiert ist.
   4. Ergebnis ist der Ergebnistyp des Operators.

3.5.2 Let-Command (Scope Management)

Beim Knoten `LetCommand` (lokale Deklarationen) muss das Scope-Management erfolgen:

```
visitLetCommand(ast) {
  idTable.openScope();
  visit(ast.Declarations); // Trägt lokale Variablen ein
  visit(ast.Command);      // Rumpf mit Sichtbarkeit
  idTable.closeScope();    // Entfernt lokale Variablen
}
```

3.6 Standardumgebung

Die Standardumgebung (Standard Environment) enthält vordefinierte Typen (`Integer`, `Boolean`) und Funktionen (`put`, `get`).

- Diese werden nicht geparst, sondern müssen vor der Analyse in die Symboltabelle geladen werden.
- **Implementierung:** Man erzeugt manuell kleine AST-Teilbäume für diese Definitionen (z. B. eine `ConstDeclaration` für `true`) und trägt sie in den initialen Scope ein.
- `Integer`, `Boolean`, etc. werden oft als Singleton-Objekte implementiert (z. B. `Type.intT`).

3.7 Typäquivalenz

Wann gelten zwei Typen als „gleich“? Dies ist besonders bei Arrays und Records wichtig.

- **Strukturelle Typäquivalenz (Triangle):** Zwei Typen sind äquivalent, wenn ihre Struktur identisch ist.
  - Beispiel: `array 8 of Char` ist äquivalent zu `array 8 of Char`, auch wenn sie an verschiedenen Stellen stehen.
- **Namensäquivalenz (Name Equivalence):** Jede Typdefinition erzeugt einen einzigartigen Typ.
  - Beispiel:

```
type T1 = array 8 of Char;
type T2 = array 8 of Char;
var a : T1;
var b : T2;
```

Bei Namensäquivalenz sind `a` und `b` **nicht** kompatibel. Bei struktureller Äquivalenz sind sie kompatibel.

Handhabung komplexer Typen: In der Analyse werden Typnamen (z. B. „Word“) durch Verweise auf ihre tatsächliche Definition (den Sub-AST, z. B. `ArrayTypeDenoter`) aufgelöst. Der Vergleich erfolgt dann rekursiv über die Struktur der `TypeDenoter`.

4 Laufzeitorganisationen

4.1 Einführung und Überblick

Die Laufzeitorganisation beschäftigt sich mit der Abbildung von abstrakten Strukturen einer Hochsprache (Variablen, Prozeduren, Objekte) auf die konkreten Ressourcen der Zielmaschine (Register, Speicher, Instruktionen).

Es existiert eine **Semantic Gap** zwischen den komplexen Konstrukten der Hochsprache (Arrays, Objekte, Methoden) und den primitiven Möglichkeiten der Hardware.

Aufgaben der Laufzeitorganisation

- Datendarstellung (Primitive Typen, Records, Arrays).
- Auswertung von Ausdrücken (Stack vs. Register).
- Speicherverwaltung (Global, Lokal/Stack, Heap).
- Routinen und Aufrufkonventionen (Parameterübergabe).

4.2 Triangle Abstract Machine (TAM)

Die TAM ist eine abstrakte Zielmaschine für Lehrzwecke. Sie basiert auf einer **Harvard-Architektur**, was bedeutet, dass Befehls- und Datenspeicher getrennt sind.

4.2.1 Speicherbereiche und Register

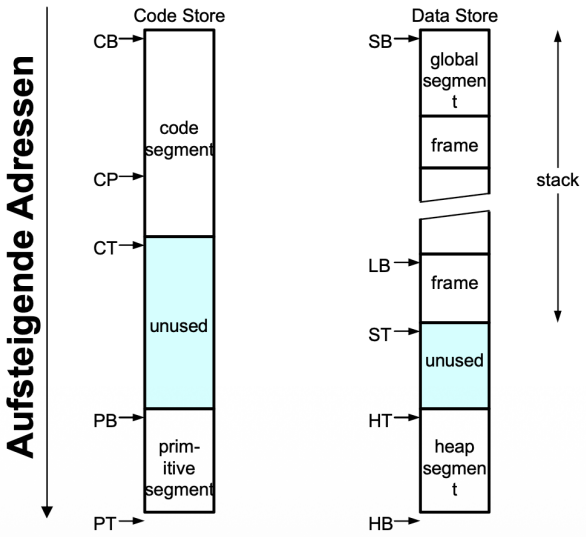
Die TAM nutzt verschiedene Register zur Adressierung der Speichersegmente.

Instruktionsspeicher (Code Store) Der Code-Speicher enthält das ausführbare Programm.

- **CB** (Code Base): Startadresse des Code-Segments (konstant).
- **CT** (Code Top): Endadresse des Code-Segments (konstant).
- **CP** (Code Pointer): Aktueller Befehlszähler (Instruction Pointer), zeigt auf den nächsten auszuführenden Befehl.
- **PB** (Primitive Base): Startadresse der Intrinsics (eingebaute Funktionen).
- **PT** (Primitive Top): Endadresse der Intrinsics.

Datenspeicher (Data Store) Der Datenspeicher ist in Stack und Heap unterteilt. In der TAM wachsen diese Bereiche aufeinander zu (siehe Speicherverwaltung).

- **SB** (Stack Base): Boden des Stacks (Start der globalen Variablen).
- **ST** (Stack Top): Aktuelles oberes Ende des Stacks.
- **HB** (Heap Base): Startadresse des Heaps (oberes Ende des Speichers).
- **HT** (Heap Top): Aktuelle Grenze des belegten Heaps.



- **LB** (Local Base): Zeiger auf den aktuellen *Stack Frame* (Beginn der lokalen Variablen der aktuellen Prozedur).

4.2.2 Instruktionen

TAM-Instruktionen sind 32-bit breit und haben folgendes Format:

$$\text{Instruktion} = \underbrace{\text{op (4 Bit)}}_{\text{Opcode}} \mid \underbrace{\text{r (4 Bit)}}_{\text{Register}} \mid \underbrace{\text{n (8 Bit)}}_{\text{Größe}} \mid \underbrace{\text{d (16 Bit)}}_{\text{Displacement}}$$

Beispiel: `LOAD (1) 3[ST]` lädt ein Wort von der Adresse $ST + 3$.

4.3 Datendarstellung (Repräsentation)

Daten müssen im Speicher so abgelegt werden, dass sie effizient zugreifbar sind.

4.3.1 Prinzipien

1. **Unverwechselbarkeit**: Unterschiedliche Werte sollten unterschiedliche Bitmuster haben.
2. **Einzigartigkeit**: Ein Wert wird immer gleich dargestellt.
3. **Konstante Größe**: Alle Werte eines Typs belegen gleich viel Platz.

Invariante der Datengröße

Es muss gelten: $size[T] \geq \log_2(\#[T])$, wobei $\#[T]$ die Anzahl der unterschiedlichen Elemente in T ist.

4.3.2 Primitive Typen

- **Boolean**: 1 Wort (16b in TAM). Werte: 00..00 (false), 00..01 (true). *Hinweis: In C/x86 oft nur 8 Bit.*
- **Char**: 1 Wort (16b), Unicode/ASCII.
- **Integer**: 1 Wort (16b), Zweierkomplement.

4.3.3 Zusammengesetzte Typen

Records (Verbundtypen) Die Felder eines Records werden im Speicher nacheinander (sequenziell) abgelegt.

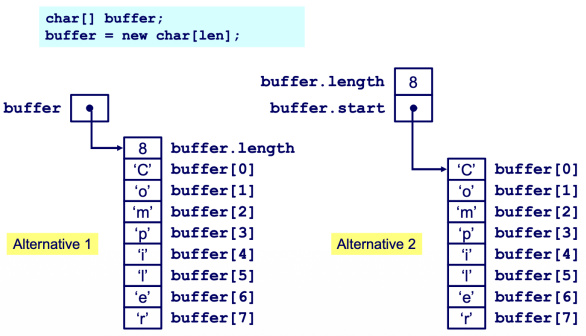
- **Adressierung**: Adresse des Records + Offset des Feldes.
- **Padding**: Viele Prozessoren verlangen eine Ausrichtung (Alignment) auf Wortgrenzen (z.B. 32-bit), was zu ungenutzten Lücken (Padding) führen kann. TAM adressiert wortweise, daher weniger Padding-Probleme, aber Platzverschwendung bei Booleans.

Arrays (Felder)

- **Statische Arrays**: Größe zur Compile-Zeit bekannt. Elemente liegen direkt hintereinander.

$$address[me[i]] = address[me] + i \times size[Element]$$

- **Dynamische Arrays**: Größe erst zur Laufzeit bekannt.
- **Repräsentation**: Indirekt über einen **Deskriptor** (Dope Vector). Dieser enthält einen Zeiger auf die Daten (im Heap) und die aktuelle Größe.



Variante Records (Disjoint Unions) Ähnlich wie Records, aber die Komponenten überlagern sich im Speicher (Union in C). Ein *Type Tag* entscheidet, welche Interpretation gerade gültig ist. Die Größe richtet sich nach der größten Komponente.

4.4 Auswertung von Ausdrücken

Wie werden mathematische Ausdrücke wie $a \times a + 2 \times a \times b$ berechnet?

4.4.1 Stack-Maschine (z.B. TAM)

Arbeitet nach dem **Post-Fix-Prinzip**. Operanden werden auf den Stack gelegt (`LOAD`), Operationen (`ADD`, `MUL`) nehmen die obersten Elemente, verrechnen sie und legen das Ergebnis zurück.

- *Vorteil*: Einfache Code-Generierung, keine Registerverwaltung nötig.
- *Nachteil*: Viele Speicherzugriffe, langsamer als Registermaschinen.

4.4.2 Register-Maschine

Berechnungen finden in schnellen CPU-Registern statt.

- *Vorteil*: Sehr schnell.
- *Nachteil*: Begrenzte Anzahl Register erfordert komplexe Zuteilungsstrategien (Register Allocation), wenn Zwischenergebnisse die Anzahl der Register übersteigen ("Spilling").

4.5 Speicherverwaltung (Stack)

Die Verwaltung des Speichers für Variablen hängt von ihrer Lebensdauer ab.

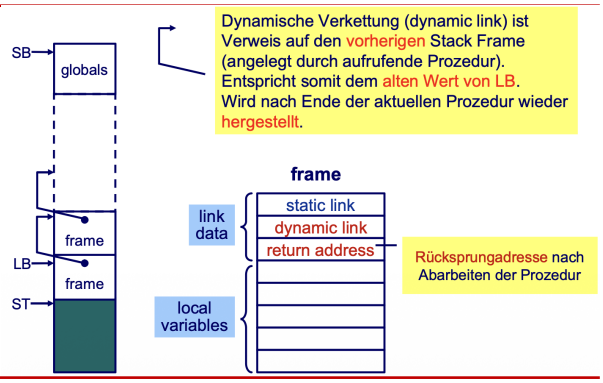
4.5.1 Arten von Variablen

1. **Globale Variablen**: Existieren über die gesamte Laufzeit. Adresse ist fest relativ zu SB.
2. **Lokale Variablen**: Existieren nur, solange der Block (Prozedur/Funktion) aktiv ist. Verwaltung über den **Stack**.
3. **Heap-Variablen**: Lebensdauer unabhängig vom Scope (siehe Abschnitt Heap).

4.5.2 Stack Frame (Activation Record)

Jeder Prozeduraufruf erzeugt einen neuen Stack Frame. Dieser enthält:

- **Parameter**: Vom Aufrufer abgelegt.
- **Verwaltungsdaten (Link Data)**: Static Link, Dynamic Link, Rücksprungadresse.
- **Lokale Variablen**: Innerhalb der Prozedur angelegt.
- **Zwischenergebnisse**: Für die Expression-Evaluation.



4.5.3 Verkettung (Linking)

Um auf Variablen zuzugreifen, werden zwei Arten von Links im Stack Frame gespeichert:

Dynamic Link (Dynamische Verkettung)

Zeigt auf den Stack Frame des **Aufrufers** (Caller). Entspricht dem alten Wert des LB-Registers. Dient dazu, beim Rücksprung (Return) den Stack-Kontext des Aufrufers wiederherzustellen.

Static Link (Statische Verkettung)

Zeigt auf den Stack Frame der Prozedur, die die aktuelle Prozedur im Quelltext **umschließt** (textuelle/lexikalische Hierarchie). Dient dem Zugriff auf **nicht-lokale Variablen** in verschachtelten Prozeduren.

Bestimmung des Static Link (SL) Wenn Prozedur P (auf Ebene L_P) eine Prozedur Q (auf Ebene L_Q) aufruft:

- **Aufruf einer globalen Prozedur** ($L_Q = 0$): $SL = SB$.
- **Aufruf einer eingebetteten Prozedur** ($L_Q > 0$):
  - *Q ist direkt in P definiert* ($L_Q = L_P + 1$): $SL = LB$ (aktueller Frame von P).
  - *Q ist auf gleicher Ebene oder weiter außen* ($L_Q \leq L_P$): Man muss der statischen Kette von P folgen ($k = L_P - L_Q + 1$ Schritte), um den korrekten Kontext zu finden.

Display-Register: Eine Alternative zur statischen Verkettung, bei der ein Array von Zeigern (Display) gepflegt wird, das für jede Schachtelungstiefe direkt auf den aktuellen gültigen Frame zeigt. Schnellerer Zugriff, aber aufwendigerer Prozeduraufruf.

4.6 Routinen und Protokolle

Das Zusammenspiel von Aufrufer (Caller) und Aufgerufenem (Callee) wird durch ein Protokoll (Calling Convention) geregelt.

4.6.1 Ablauf eines Aufrufs (TAM)

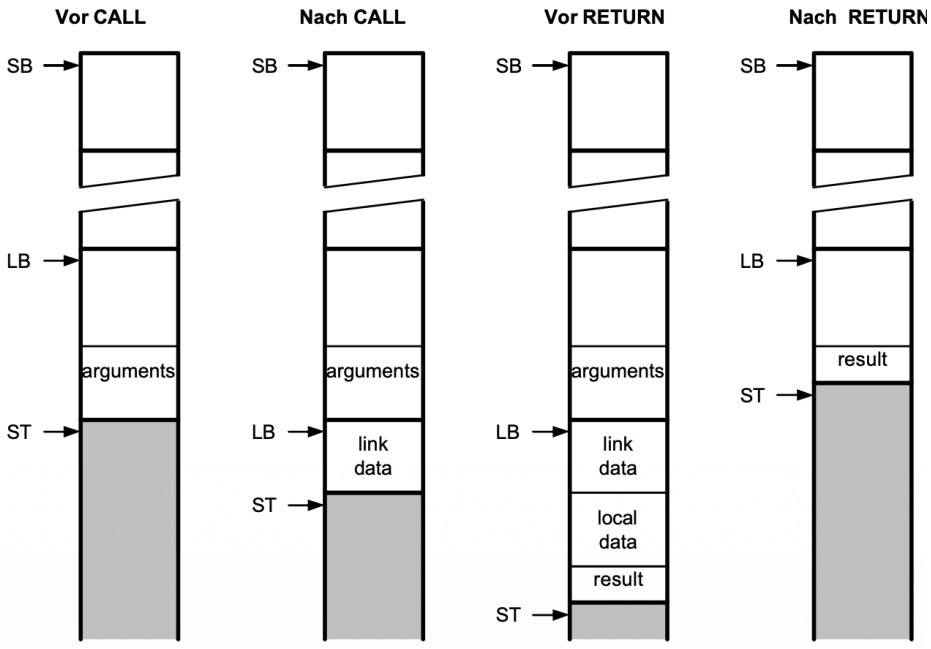


Figure 1: *Vor/Nach Call/Return im Stack*

1. **Vor CALL (Caller)**:
   - Argumente (Parameter) werden auf den Stack gepusht (in TAM: in umgekehrter Reihenfolge, damit das erste Argument oben liegt oder direkt über LB adressierbar ist).
2. **CALL (Instruktion)**:
   - Sichert *Static Link* (wird berechnet/übergeben).

- Sichert *Dynamic Link* (aktueller LB).
- Sichert *Return Address* (PC + 1).
- Setzt neuen *LB* auf den Beginn des neuen Frames.
- Sprung zur Code-Adresse der Routine.

3. **In der Routine**:
   - Reserviert Platz für lokale Variablen (Inkrementiert *ST*).
4. **RETURN (Callee)**:
   - Entfernt lokalen Speicher und Verwaltungsdaten.
   - Entfernt Argumente vom Stack.
   - Legt Rückgabewert (Result) auf den Stack.
   - Stellt alten *LB* und *ST* wieder her.
   - Springt zurück.

4.6.2 Parameterübergabe

Parameter werden relativ zu *LB* mit **negativen Offsets** adressiert (da sie vor dem Frame-Start auf den Stack gelegt wurden). Lokale Variablen haben positive Offsets.

- **Call-by-Value**: Der Wert der Variable wird kopiert. Änderungen in der Prozedur haben keinen Effekt auf den Aufrufer.
- **Call-by-Reference** (`var`): Die *Adresse* der Variable wird übergeben. Die Prozedur arbeitet via Indirektion direkt auf dem Speicherplatz des Aufrufers. Änderungen sind global sichtbar.

4.6.3 Funktionen als Parameter (Closures)

Wenn eine Funktion *F* als Parameter übergeben wird, reicht die Startadresse nicht aus, da *F* Zugriff auf ihren statischen Kontext benötigt.

- Lösung: **Closure** (Funktionsabschluss).
- Repräsentation: Paar aus `(Code-Adresse, Static Link)`.
- Aufruf: `CALLI` (Call Indirect) nutzt dieses Paar.

4.7 Heap-Speicherverwaltung

Der Heap dient für Daten, deren Lebensdauer nicht an den Block-Scope gebunden ist (z.B. verkettete Listen, Bäume).

4.7.1 Organisation

In der TAM (und vielen Systemen) wachsen Stack und Heap aufeinander zu. Wenn sie sich treffen → *Out of Memory*.

- **Allokation**: Suchen eines freien Blocks geeigneter Größe.
- **Deallokation**: Freigabe von Speicher.

4.7.2 Probleme und Strategien

- **Fragmentierung**: Durch unregelmäßiges Anlegen und Freigeben entstehen Lücken ("Löcher"), die zu klein für neue Objekte sind, obwohl in Summe genug Speicher frei wäre.
- **Freispeicherliste (Free List)**: Liste (z.B. `HF` in TAM) verkettet alle freien Blöcke.
- **Kompaktierung**: Verschieben von belegten Blöcken, um Lücken zu schließen. Erfordert Aktualisierung aller Zeiger (schwierig!) oder Nutzung von *Handles* (Zeiger auf Zeiger).

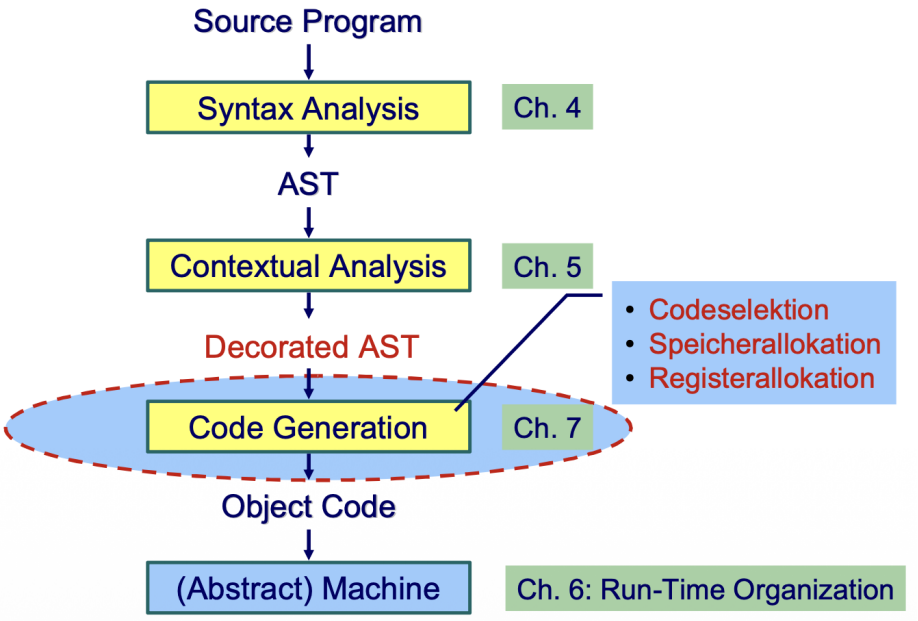
4.7.3 Garbage Collection (Automatische Speicherbereinigung)

Verfahren, um nicht mehr erreichbaren Speicher automatisch freizugeben (z.B. in Java).

Mark-and-Sweep Algorithmus

1. **Mark (Markieren)**: Gehe von allen Wurzeln (Register, Stack-Variablen) aus und verfolge alle Zeiger. Markiere jedes erreichte Objekt im Heap als "lebendig".
2. **Sweep (Fegen)**: Durchlaufe den gesamten Heap. Alle nicht markierten Objekte sind "Müll" und werden zur Freispeicherliste hinzugefügt. Markierungen werden für den nächsten Lauf zurückgesetzt.

5 Code-Generierung



5.1 Einführung und Herausforderungen

Die Code-Generierung ist der letzte Schritt im Compiler-Backend. Sie übersetzt den dekorierten Abstrakten Syntaxbaum (AST) in den Zielcode (hier: TAM-Maschinencode).

- **Abhängigkeit:** Dieser Schritt hängt sowohl von der **Semantik der Eingabesprache** als auch von der **Zielmaschine** ab. Das macht eine allgemeingültige Formulierung schwierig.
- **Semantik:** Erst in diesem Schritt erhält eine Zeichenkette (z.B. "4.2") ihre tatsächliche Bedeutung als Zahl (Integer 42) auf der Zielmaschine.
- **Ziel:** Erzeugung einer Instruktionsfolge, die semantisch äquivalent zum Quellprogramm ist.

Das Gesamtproblem wird in drei Teilprobleme zerlegt:

1. **Code-Selektion:** Abbildung von Phrasen des Quellprogramms auf Folgen von Maschineninstruktionen.
2. **Speicherallokation:** Zuweisung von Speicheradressen für Variablen und Verwaltung der Adressen (Buchführung).
3. **Registerallokation:** Verwaltung der Register für Variablen und Zwischenergebnisse (bei der TAM als Stackmaschine weitestgehend irrelevant, da alles auf dem Stack passiert).

5.2 Code-Selektion und Code-Funktionen

Die Übersetzung erfolgt **induktiv**: Die Übersetzung des Gesamtprogramms wird aus den Übersetzungen der Einzelphrasen hergeleitet. Hierfür werden **Code-Funktionen** definiert, die durch **Code-Schablonen** (Templates) konkretisiert werden.

Code-Funktionen

Eine Code-Funktion bildet eine Phrase (Teil des AST) auf eine Folge von Maschineninstruktionen ab.

Wichtige Code-Funktionen für Triangle/TAM:

- $run[P]$: Führt das Programm P aus und hält dann an. Startet und endet mit leerem Stack.
- $execute[C]$: Führt das Kommando C aus. Der Stack darf sich während der Ausführung ändern, muss aber am Ende wieder die gleiche Höhe haben (keine Netto-Stack-Änderung).
- $evaluate[E]$: Wertet den Ausdruck E aus und legt das Ergebnis oben auf den Stack (Push result).
- $fetch[V]$: Legt den Wert der Variablen/Konstanten V auf den Stack.
- $assign[V]$: Nimmt einen Wert vom Stack und speichert ihn in die Variable V.
- $elaborate[D]$: Verarbeitet eine Deklaration D. Reserviert Speicherplatz auf dem Stack für die deklarierten Variablen.

5.2.1 Beispiele für Code-Schablonen

1. Sequentielle Ausführung ($C_1; C_2$)

$$execute[[C_1; C_2]] = \begin{cases} execute[[C_1]] \\ execute[[C_2]] \end{cases}$$

2. Zuweisung ($V := E$)

$$execute[[V := E]] = \begin{cases} evaluate[[E]] & \text{(Berechnet Wert, legt auf Stack)} \\ assign[[V]] & \text{(Speichert Wert vom Stack in V)} \end{cases}$$

3. If-Anweisung ($if\ E\ then\ C_1\ else\ C_2$) Hier werden Labels und Sprungbefehle benötigt.

$$execute[[if\ E\ then\ C_1\ else\ C_2]] = \begin{cases} evaluate[[E]] & \\ JUMPIF(0)\ L_{else} & \text{(Springe zu Else, wenn false/0)} \\ execute[[C_1]] & \\ JUMP\ L_{fi} & \text{(Überspringe Else-Zweig)} \\ L_{else}: & \\ execute[[C_2]] & \\ L_{fi}: & \end{cases}$$

4. While-Schleife (Optimierung) Eine naive Implementierung prüft die Bedingung am Anfang und springt am Ende zurück. Eine effizientere Variante (weniger Sprünge im "Steady State") nutzt folgenden Aufbau:

$$execute[[while\ E\ do\ C]] = \begin{cases} JUMP\ L_{check} & \\ L_{loop}: & \\ execute[[C]] & \\ L_{check}: & \\ evaluate[[E]] & \\ JUMPIF(1)\ L_{loop} & \text{(Springe zurück, wenn true)} \end{cases}$$

Anmerkung: In der Vorlesung wurde auch die Standard-Variante mit L_{while} am Anfang und $JUMPIF(0)$ zum Ende besprochen.

Sonderfallbehandlung (Optimierung): Anstatt generische Schablonen zu nutzen, können spezielle Muster erkannt werden:

- $i + 1$: Statt 'LOAD i', 'LOADL 1', 'CALL add' → 'LOAD i', 'CALL succ'.
- **Constant Inlining:** Wenn der Wert einer Konstanten zur Kompilierzeit bekannt ist, muss kein Speicherzugriff ('LOAD') erfolgen. Stattdessen wird der Wert direkt mit 'LOADL' in den Code eingebettet.

5.3 Implementierung mittels Visitor-Pattern

Der Code-Generator wird systematisch mit dem Visitor-Pattern implementiert. Verschiedene Encoder-Klassen (z.B. ‘CommandEncoder‘, ‘ExpressionEncoder‘) besuchen die entsprechenden AST-Knoten.

5.3.1 Backpatching (Rückwärts-Einsetzen von Adressen)

Ein Problem bei der Generierung von Kontrollstrukturen (wie ‘if‘ oder ‘while‘) sind Vorwärtssprünge, da die Zieladresse des Sprungs zum Zeitpunkt der Generierung des Sprungbefehls noch nicht bekannt ist (der Code dazwischen wurde noch nicht erzeugt).

Lösung: Backpatching

1. Erzeuge Sprunginstruktion mit einer “leeren” Zieladresse (z.B. 0).
2. Merke die Adresse dieser unvollständigen Instruktion.
3. Generiere den Code für den Rumpf/Block.
4. Sobald die Zieladresse erreicht ist (aktuelle Code-Adresse), **patche** die gemerkte Instruktion nachträglich mit der korrekten Adresse.

```
int jumpAddr = nextInstrAddr;
emit(Machine.JUMPIFop, 0, Machine.CBr, 0); // Dummy Ziel 0
// ... Generiere Code fuer Block ...
int targetAddr = nextInstrAddr;
patch(jumpAddr, targetAddr); // Trage echtes Ziel nach
```

Pseudocode Backpatching

5.4 Speicherverwaltung und Adressierung

5.4.1 Verwaltung im AST (Runtime Entities)

Der Code-Generator muss wissen, wo Variablen liegen oder welche Werte Konstanten haben. Diese Information wird in **Entitätsdeskriptoren** (‘RuntimeEntity‘) gespeichert und an die AST-Knoten gehängt.

- **KnownValue:** Für Konstanten mit bekanntem Wert. Speichert den Wert direkt.
- **KnownAddress:** Für Variablen/Konstanten mit bekannter Adresse. Speichert Größe und Adresse (Level, Displacement).
- **UnknownValue:** Wert wird erst zur Laufzeit berechnet (z.B. ‘const c x + 5‘).
- **UnknownAddress:** Für Referenzparameter (‘var‘-Parameter), deren Adresse erst zur Laufzeit bekannt ist.

5.4.2 Adressvergabe

Der Compiler führt Buch über den belegten Speicherplatz. Dies geschieht oft über einen Parameter in den Visitor-Methoden (z.B. ‘gs‘ für Global Space oder ‘frame‘).

- **Eingabe-Parameter:** Aktueller Offset / erste freie Adresse.
- **Rückgabewert:** Größe des durch die Deklaration zusätzlich belegten Speichers.

Vorteil der Rückgabe des Deltas: Beim Verlassen eines Blocks (Scope) muss keine globale Variable zurückgesetzt werden; die lokalen Änderungen “verfallen” automatisch.

Globale Variablen

```
let
   var a: Integer;
   var b: Boolean;
   var c: Integer
in begin
   ...
end
```

var	size	address
a	1	[0]SB
b	1	[1]SB
c	1	[2]SB

In TAM, echte Maschinen haben hier wahrscheinlich unterschiedliche Grössen

Verschachtelte Blöcke

```
let var a: Integer
in  begin
    ...
    let var b: Boolean;
        var c: Integer
    in ...

    let var d: Integer
    in ...
end
```

var	size	address
a	1	[0]SB
b	1	[1]SB
c	1	[2]SB
d	1	[1]SB

d verwendet Platz von **b** wieder (anderer Geltungsbereich)

5.4.3 Adressierung in verschachtelten Blöcken (Triangle)

In Sprachen mit verschachtelten Prozeduren (wie Triangle, im Gegensatz zu Mini-Triangle) reicht eine einfache Adresse relativ zu 'SB' (Stack Base) nicht aus.

Adress-Tupel

Jede Variable wird durch ein Paar identifiziert: **(Schachtelungstiefe, Displacement)**.

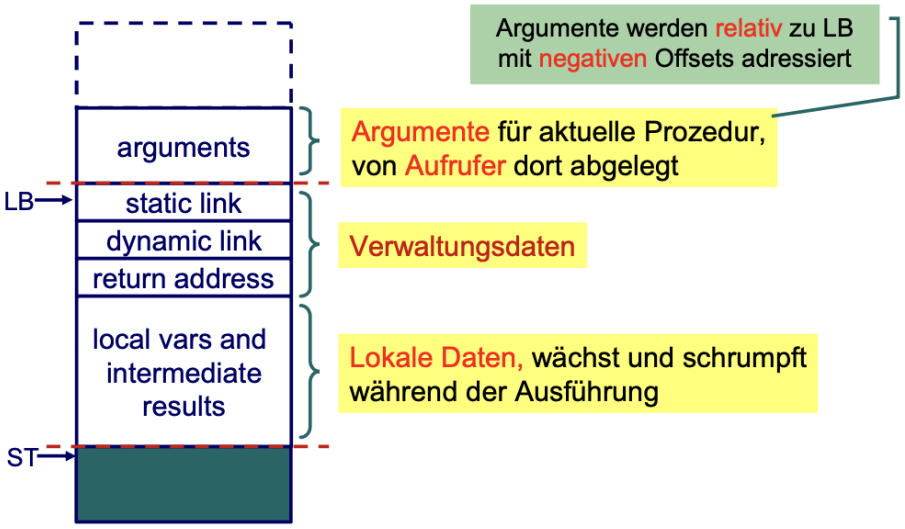
Der Zugriff erfolgt über **Display-Register** oder 'LB' (Local Base) / 'SB' (Stack Base):

Algorithmus zur Wahl des Basisregisters: Sei $level_{decl}$ die Ebene, auf der die Variable deklariert wurde, und $level_{current}$ die aktuelle Ausführungsebene.

- Wenn $level_{decl} == 0$: Benutze **SB** (Globale Variable).
- Wenn $level_{decl} == level_{current}$: Benutze **LB** (Lokale Variable im aktuellen Frame).
- Sonst: Benutze Display-Register oder Statische Verkettung, um den Frame zu finden.

$$r = L(level_{current} - level_{decl})$$

Oder über Display-Register direkt, sofern die Architektur diese unterstützt.



5.5 Prozeduren und Funktionen

5.5.1 Deklaration

Da der Code linear im Speicher liegt, aber Prozeduren nur bei Aufruf ausgeführt werden sollen, muss der Kontrollfluss um den Prozedurrumpf herumgeleitet werden.

Code-Schema für Deklaration ‘proc I() C‘:

1. **JUMP g**: Überspringe den Prozedurcode.
2. **Label e**: Einstiegspunkt (Entry) der Prozedur.
3. **execute[C]**: Code des Rumpfes.
4. **RETURN**: Rückkehr zum Aufrufer.
5. **Label g**: Ziel des Sprungs von Schritt 1 (weiter im Hauptprogramm).

5.5.2 Aufruf (Call)

Beim Aufruf (‘CALL‘) muss die **Statische Verkettung** (Static Link) korrekt gesetzt werden, damit die aufgerufene Prozedur auf Variablen der umgebenden Scopes zugreifen kann.

- Die Adresse der Routine ist im Deskriptor gespeichert (‘KnownRoutine‘).
- Der ‘CALL‘-Befehl erhält das korrekte Basisregister (siehe Adressierung), welches als Static Link in den neuen Stack Frame geschrieben wird.

5.5.3 Parameterübergabe

- **Aufrufer:** Legt die aktuellen Parameter (Argumente) auf den Stack.
- **Aufgerufener:** Greift auf Parameter relativ zu ‘LB‘ mit **negativen Offsets** zu (da sie vor dem Link-Data-Bereich auf dem Stack liegen).
- **Var-Parameter:** Werden als ‘UnknownAddress‘ behandelt. Es wird nicht der Wert, sondern die Adresse übergeben und bei ‘fetch‘/‘assign‘ genutzt.

6 Lexer/Parser-Generierung mit ANTLR

6.1 Einführung und Grundlagen

ANTLR (Another Tool for Language Recognition) ist ein Generator für Lexer und Parser. In der aktuellen Version 4 setzt es auf einen adaptiven $LL(*)$-Algorithmus (auch $ALL(*)$ genannt).

ANTLR v4 Eigenschaften

- **Adaptive $LL(*)$:** Die Grammatik wird zur Laufzeit analysiert. Der Parser nutzt Heuristiken, um Konflikte und Mehrdeutigkeiten automatisch aufzulösen.
- **Direkte Linksrekursion:** Wird im Gegensatz zu klassischen LL-Parsern unterstützt (wird intern transformiert). Indirekte Linksrekursion ist weiterhin nicht erlaubt.
- **Trennung von Grammatik und Code:** Semantische Aktionen (eingebetteter Java-Code) werden vermieden. Stattdessen werden Parse-Trees automatisch erstellt und über *Listener* oder *Visitor* traversiert.

6.2 Struktur einer Grammatik

Eine ANTLR-Grammatikdatei (Endung '.g4') definiert Lexer- und Parser-Regeln.

- **Parser-Regeln:** Beginnen mit einem **Kleinbuchstaben** (z.B. `stat`, `expr`). Sie definieren die syntaktische Struktur.
- **Lexer-Regeln (Token):** Beginnen mit einem **Großbuchstaben** (z.B. `ID`, `INT`). Sie definieren die lexikalischen Einheiten (Zeichenfolgen).

6.2.1 Operatoren und Syntax

Die Notation orientiert sich an EBNF (Extended Backus-Naur Form).

Operator	Bedeutung
`x y`	Konkatenation (x gefolgt von y)
`x | y`	Alternative (x oder y)
`x*`	0 oder mehr Wiederholungen (Kleene-Stern)
`x+`	1 oder mehr Wiederholungen
`x?`	0 oder 1 Mal (optional)
`~[a-z]`	Negation (alle Zeichen außer a-z)
`.*?`	**Non-Greedy Match**: Matcht so wenig Zeichen wie möglich (wichtig für Kommentare!)

Non-Greedy Beispiel

Um beispielsweise '/* ... */' Kommentare korrekt zu parsen, darf der '*'-Operator nicht gierig (greedy) sein, da er sonst bis zum allerletzten '*/' der Datei lesen würde.
Lösung: `'/*' .*? '*/'` (stoppt beim ersten Vorkommen von '*/').

6.3 Generierte Artefakte und Parse-Trees

ANTLR erzeugt aus der Grammatik Java-Klassen (oder andere Zielsprachen), die den Input in einen abstrakten Parse-Tree (AST) verwandeln.

Wichtige Klassen:

- `Parser`: Überprüft die syntaktische Struktur.

- `Lexer`: Zerlegt den Input-Stream in Tokens.
- `ParserRuleContext`: Repräsentiert innere Knoten im Baum (Regeln). Speichert Start-/End-Tokens und Referenzen auf Kinder.
- `TerminalNode`: Repräsentiert Blätter im Baum (Tokens/Literale).

6.4 Traversierung: Listener vs. Visitor

ANTLR v4 bietet zwei Mechanismen, um den Parse-Tree zu verarbeiten.

6.4.1 1. Listener Pattern

Der Listener ist der Standardmechanismus. ANTLR generiert einen `ParseTreeWalker`, der den Baum mittels Tiefensuche (DFS) automatisch traversiert.

- **Funktionsweise:** Der Listener reagiert auf Ereignisse. Beim Betreten eines Knotens wird `enterX()` aufgerufen, beim Verlassen `exitX()`.
- **Vorteil:** Vollautomatisch, kein Boilerplate-Code für die Traversierung nötig.
- **Nachteil:** Keine Kontrolle über den Ablauf (Reihenfolge ist fix), keine Rückgabewerte aus Methoden, Kommunikation zwischen Knoten schwierig (oft über Instanzvariablen oder einen Stack).

6.4.2 2. Visitor Pattern

Muss explizit mit `-visitor` generiert werden. Hier steuert der Entwickler die Traversierung selbst.

- **Funktionsweise:** Man implementiert `visitX(Context ctx)`. Um Kinder zu besuchen, muss man explizit `visit(ctx.child)` aufrufen.
- **Vorteil:** Volle Kontrolle (Überspringen von Zweigen, geänderte Reihenfolge), Methoden können Werte zurückgeben (z.B. `Integer` für einen Taschenrechner).
- **Nachteil:** Man muss die Traversierung (den Aufruf von `visit` für Kinder) selbst schreiben.

Vergleich für die Prüfung

Nutzen Sie **Listener**, wenn Sie den gesamten Baum standardmäßig abarbeiten wollen (z.B. Code-Formatierung, einfache Übersetzung).
Nutzen Sie **Visitor**, wenn Sie Ergebnisse berechnen (Interpreter), den Kontrollfluss steuern oder kontextabhängig traversieren müssen.

6.4.3 Labeling von Alternativen

Damit der Visitor/Listener spezifische Methoden für Alternativen generiert (statt einer riesigen Methode mit ‘if/else‘), können Alternativen mit ‘#‘ benannt werden.

Beispiel:

```
expr : expr '*' expr  # MulDiv
     | expr '+' expr  # AddSub
     | INT            # Int
     ;
```

Erzeugt Methoden: `visitMulDiv`, `visitAddSub`, `visitInt`.

6.5 Fortgeschrittene Themen

6.5.1 Assoziativität und Präzedenz

In ANTLR v4 wird Operator-Präzedenz implizit durch die Reihenfolge der Alternativen bestimmt.

- **Präzedenz:** Regeln, die weiter oben stehen, binden stärker (haben höhere Priorität). Beispiel: '*' vor '+' definieren.
- **Assoziativität:** Standard ist links-assoziativ $(1 + 2 + 3 \rightarrow (1 + 2) + 3)$. Rechts-Assoziativität (z.B. für Exponenten 2^{3^4}) wird mit `<assoc=right>` annotiert.

6.5.2 Dangling Else Problem

Das Problem der Mehrdeutigkeit bei 'if expr then if expr then stat else stat': Gehört das 'else' zum ersten oder zweiten 'if'?
Lösung in ANTLR: ANTLR parst **greedy** (gierig). Es bindet das 'else' an das nächstmögliche (innerste) offene 'if'. Dies entspricht dem Standardverhalten der meisten Programmiersprachen.

6.5.3 Semantische Prädikate

Manchmal reicht die Grammatik allein nicht aus (z.B. wenn Parsing von Laufzeitdaten abhängt).
Syntax: `{Bedingung}?`
Dies ist ein boolescher Ausdruck in der Zielsprache (Java).

- Ist die Bedingung `true`, wird die Alternative/Regel aktiviert.
- Ist sie `false`, wird die Alternative ignoriert (als ob sie nicht in der Grammatik stünde) und der Parser versucht eine andere Möglichkeit.

Beispiel (Datenabhängiges Parsen): Eine Zahl n gibt an, wie viele folgende Zahlen gelesen werden sollen.

```
sequence[int n] locals [int i = 1;]
   : ( {$i <= $n}? INT {$i++;} )* ;
```

Hier deaktiviert das Prädikat die Schleife, sobald $i > n$ ist.

6.6 Fehlerbehandlung

ANTLR generiert Parser mit eingebauter Fehlerbehandlung.

- **Token ignorieren:** Wenn ein Token unerwartet ist, aber das darauf folgende passt.
- **Token einfügen:** Wenn ein Token fehlt, fügt der Parser ein fiktives Token ein, um weiterzumachen (Single Token Insertion).
- Ziel ist es, möglichst viele Fehler in einem Durchlauf zu finden (Error Recovery), statt beim ersten Fehler abzubrechen.

7 Java Virtual Machine

Die **Java Virtual Machine (JVM)** ist das Herzstück der Java-Plattform. Sie fungiert als abstrakte Rechenmaschine, die eine Laufzeitumgebung zur Ausführung von Bytecode bereitstellt, unabhängig von der zugrunde liegenden Hardware oder dem Betriebssystem.

7.1 Architektur und Konzept

Die Java-Plattform besteht aus zwei Hauptkomponenten:

- **Java API:** Standardbibliothek für Zugriff auf Ressourcen (IO, Netzwerk, etc.).
- **Java Virtual Machine (JVM):** Lädt Klassen, führt Programme aus und verwaltet den Speicher (Garbage Collection).

Konzept der Virtuellen Maschine

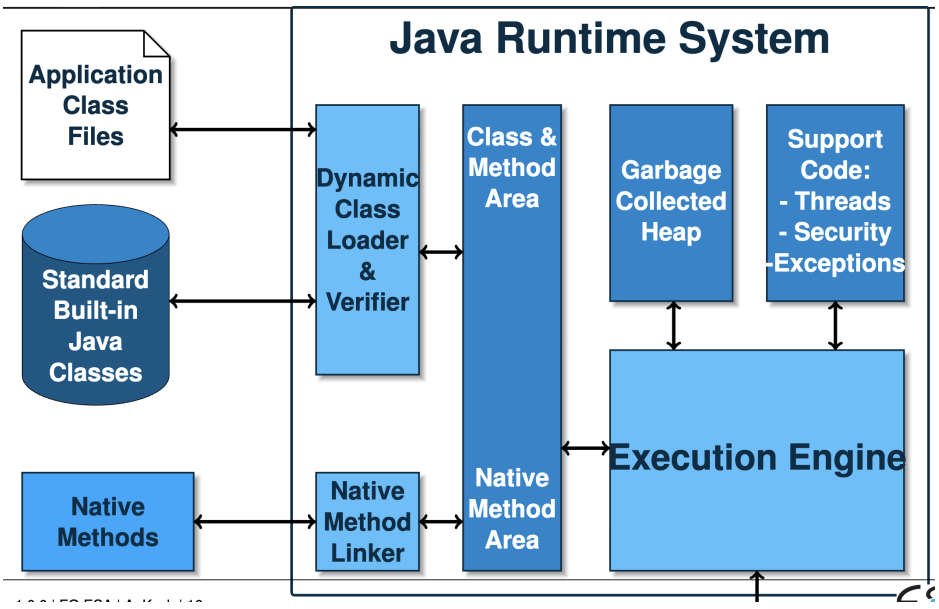
Eine **Virtuelle Maschine** ist eine "Maschine auf der Maschine". Sie definiert eine Spezifikation für die Ausführung von Programmen. Solange eine Implementierung (z.B. HotSpot) die Spezifikation erfüllt, ist das Verhalten des Programms auf jeder physischen Maschine identisch.

7.1.1 Bytecode und Kompilierung

Java-Code wird nicht direkt in Maschinencode, sondern in **Bytecode** übersetzt (.class Dateien). Dieser Ansatz ist ein Hybrid:

- **Interpretiert:** Bytecode kann direkt interpretiert werden (langsamer).
- **JIT-Kompilierung (Just-In-Time):** Häufig ausgeführte Code-Pfade ("Hotspots") werden zur Laufzeit in nativen Maschinencode übersetzt. Dies ermöglicht eine Leistung nahe an nativ kompilierten Sprachen (C++), bei gleichzeitiger Portabilität.

Portabilität: Da die JVM über das Betriebssystem abstrahiert, sind Datentypen (z.B. die Bitbreite von `int`) auf allen Plattformen identisch. Ein Programm muss nur einmal zu Bytecode kompiliert werden und läuft überall, wo eine JVM installiert ist ("Write Once, Run Anywhere").



7.2 Interner Aufbau der JVM

Das **Java Runtime System** besteht aus mehreren Subsystemen:

7.2.1 Class Loader Subsystem

Der Class Loader ist für das dynamische Laden, Linken und Initialisieren von Klassen zuständig. Der Prozess läuft in folgenden Phasen ab:

1. **Loading:** Laden des binären Class Files.
2. **Linking:** Überführung in JVM-interne Strukturen.
   - *Verification:* Prüfung der Struktur und Sicherheit (Bytecode Verifier).
   - *Preparation:* Speicherreservierung für statische Felder (Standardwerte).
   - *Resolution:* Auflösen symbolischer Referenzen im Constant Pool (kann verzögert geschehen).
3. **Initialization:** Ausführung statischer Initialisierer (z.B. `<clinit>`).

7.2.2 Speicherbereiche (Runtime Data Areas)

Die JVM verwaltet den Speicher in verschiedenen Bereichen:

- **Method Area:** Speichert Klassendefinitionen, Konstanten, statische Variablen und Code.
- **Heap:** Hier werden alle **Objekte** und **Arrays** zur Laufzeit angelegt. Der Speicher wird durch den **Garbage Collector** automatisch bereinigt.
- **Java Stack:** Speichert Stack Frames (lokale Variablen, Operandenstack) für jeden Methodenaufruf. Jeder Thread hat seinen eigenen Stack.
- **PC Register:** Zeigt auf die aktuelle Instruktion.
- **Native Method Stack:** Für Aufrufe von nativem Code (via JNI, z.B. C++ Bibliotheken).

7.3 Class File Format und Typen

Ein Class File enthält alle Informationen einer Klasse in einem binären Format (Big-Endian). Wichtige Komponenten sind:

- **Constant Pool:** Eine Tabelle, die Literale (Strings, Zahlen) und symbolische Referenzen (Klassen-, Methoden-, Feldnamen) enthält. Der Bytecode referenziert Werte oft über einen Index in diesen Pool, was das dynamische Linken ermöglicht.
- **Methods Table:** Enthält den Bytecode der Methoden.
- **Fields Table:** Beschreibt die Variablen der Klasse.

7.3.1 Type Descriptors

Die JVM verwendet kompakte Strings, um Typen zu beschreiben. Dies ist essentiell für das Verständnis von Methodensignaturen im Bytecode.

Deskriptor	Datentyp	Bemerkung
`B`	`byte`	Vorzeichenbehaftet (8 Bit)
`C`	`char`	Unicode Character (16 Bit)
`D`	`double`	64-Bit Gleitkomma
`F`	`float`	32-Bit Gleitkomma
`I`	`int`	32-Bit Integer
`J`	`long`	64-Bit Integer
`S`	`short`	Vorzeichenbehaftet (16 Bit)
`Z`	`boolean`	true/false
`V`	`void`	Nur als Rückgabetyp
`L<Klasse>;`	Object	z.B. `Ljava/lang/String;`
`[`	Array	z.B. `[I` (int[]), `[[F` (float[][])

Beispiel Methodensignatur: Java: `public int foo(char c, float f, String s)` JVM-Deskriptor: `(CFLjava/lang/Stri`

7.4 Ausführungsmodell: Stack Frames

Die JVM ist eine **stack-basierte Maschine**. Es gibt keine allgemeinen Register zur Berechnung. Operationen finden auf dem Operandenstack statt. Bei jedem Methodenaufruf wird ein neuer **Stack Frame** erzeugt, der enthält:

Bestandteile eines Stack Frames

- **Local Variables:** Ein Array von lokalen Variablen (inkl. Methodenparameter). Index 0 ist bei Instanzmethoden `this`.
- **Operand Stack:** Ein LIFO-Speicher für Zwischenergebnisse. Operationen (z.B. `iadd`) nehmen Werte vom Stack und legen das Ergebnis zurück.
- **Frame Data:** Referenzen auf den Constant Pool, Return-Adressen etc.

Wichtig: Die JVM arbeitet intern mit 32-Bit Slots ("Wörtern").

- `long` und `double` belegen **zwei** Slots (sowohl im Stack als auch in den Local Variables).
- Typen wie `short`, `byte`, `char` werden für Berechnungen implizit in `int` umgewandelt (sind sogenannte *Storage Types*).

7.5 Bytecode Instruktionen

Bytecode-Instruktionen (Opcodes) sind meist typisiert. Der erste Buchstabe (Präfix) gibt den Typ an:

- `i`: int (auch byte, char, short, boolean)
- `l`: long
- `f`: float
- `d`: double
- `a`: reference (Objekte, Arrays)

7.5.1 Daten laden und speichern

- **Konstanten laden:**
  - `iconst_<n>`: Lädt kleine Integers (-1 bis 5) effizient.
  - `bipush / sipush`: Lädt Byte/Short Konstanten.
  - `ldc`: Lädt Konstanten (Strings, große Zahlen) aus dem Constant Pool.
- **Lokale Variablen:**
  - `<p>load <index>`: Lädt Wert aus lokaler Variable auf den Stack.

– `<p>store <index>`: Speichert Wert vom Stack in lokale Variable.

7.5.2 Stack-Manipulation

Da man Register nicht direkt adressieren kann, muss der Stack oft manipuliert werden:

- `pop / pop2`: Löscht oberstes Element (1 oder 2 Wörter).
- `dup / dup2`: Dupliziert oberstes Element.
- `swap`: Vertauscht die obersten zwei Elemente.

7.5.3 Arrays

Arrays sind Objekte auf dem Heap. Zugriff erfolgt in drei Schritten:

1. Referenz auf das Array laden.
2. Index laden.
3. Operation ausführen (Wert wird geladen/gespeichert).

Instruktionen:

- `newarray`: Erstellt Array primitiver Typen.
- `anewarray`: Erstellt Array von Referenzen.
- `<p>aload`: Lädt Wert aus Array auf Stack (`ArrRef, Index -> Value`).
- `<p>astore`: Speichert Wert in Array (`ArrRef, Index, Value -> _`).

7.5.4 Arithmetik und Logik

Arithmetische Operationen konsumieren Operanden vom Stack und legen das Ergebnis ab.

- `<p>add, <sub>, <mul>, <div>, <rem>, <neg>`
- Logisch (nur int/long): `shl, shr, ushr, and, or, xor`.

Es gibt keine implizite Typumwandlung. Explizite Konversion nötig: `i2f` (int zu float), `l2i`, etc.

7.5.5 Kontrollfluss

Die JVM bietet Sprungbefehle (`goto`) und bedingte Sprünge.

1. Integer-Vergleiche:

- Unär (vergleicht Stack-Top mit 0): `ifeq` (==0), `iflt` (¡0), etc.
- Binär (vergleicht zwei oberste Werte): `if_icmpeq`, `if_icmpge`, etc.

2. Float/Double/Long Vergleiche (Wichtig!): Die JVM hat keine direkten Sprungbefehle für diese Typen (außer Vergleich gegen 0). Der Vergleich erfolgt über einen "Umweg" in zwei Schritten:

1. Eine Vergleichsoperation (`lcmp`, `fcmpl`, `dcmpg`) vergleicht zwei Werte und legt das Ergebnis als `int` (-1, 0, 1) auf den Stack.
2. Ein Standard-Integer-Branch (z.B. `ifge`) wertet dieses Ergebnis aus.

7.5.6 Methodenaufrufe

- `invokestatic`: Aufruf statischer Methoden.
- `invokevirtual`: Aufruf von Instanzmethoden (dynamischer Dispatch).

Protokoll: Argumente werden auf den Stack gelegt. Die aufgerufene Methode findet diese in ihren lokalen Variablen. Der Rückgabewert (falls vorhanden) liegt nach Rückkehr auf dem Stack des Aufrufers.

7.6 ASM Framework

Für die Code-Generierung im Compilerbau wird oft das ASM-Framework genutzt. Es abstrahiert die binäre Erzeugung des Class Files.

- Arbeitet Event-basiert (Visitor Pattern) oder Tree-basiert.
- Nimmt dem Entwickler die Berechnung von Sprung-Offsets und Constant Pool Indizes ab.
- Beispiel: `currentGenerator.push(5)` erzeugt automatisch `iconst_5`.