1 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.

1.1 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.

1.1.1 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.

1.2 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).

1.3 Grammatiken und Transformationen

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

1.3.1 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)

1.3.2 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).

1.4 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.

1.5 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");
  }
}
```

1.6 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.

1.6.1 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 $Starters[[X]] \cup Starters[[Y]]$.
- $Starters[[X|Y]] = Starters[[X]] \cup Starters[[Y]]$

1.6.2 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 $Follow[[B]]$.

1.6.3 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.

1.7 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).

1.8 Lexikalische Analyse (Scanner-Implementierung)

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

1.8.1 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).

1.8.2 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`).

1.8.3 Schnittstelle zum Parser

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