1 Syntaktische Analyse/Lexparse

Übersetzung und Phasen

- **Syntaxanalyse** → Struktur des Programms (AST)
- **Kontextanalyse** → Bedeutungsprüfung (Typen, Gültigkeit)
- **Codegenerierung** → Übersetzung in Zielcode

1.1 Compilerstruktur

Ein-Pass-Compiler:

- Führt alle Phasen gleichzeitig aus
- Keine echte Zwischendarstellung (IR)
- Typisch für kleine Sprachen (z. B. Pascal)

Multi-Pass-Compiler:

- Arbeitet mit mehreren Durchgängen über Quelltext/IR
- Datenweitergabe über IR (AST)
- Bessere Modularität und Optimierung

1.2 Syntaxanalyse – Überblick

- **Scanner (Lexer)**: Wandelt Zeichenfolge → Tokenfolge
- **Parser**: Wandelt Tokenfolge → Abstract Syntax Tree (AST)
- Token = atomares Symbol des Quellprogramms

Kontextfreie Grammatik (CFG)

Eine CFG ist ein 4-Tupel (N, T, P, S) mit:
- N: Nichtterminale
- T: Terminale
- P: Produktionen
- S: Startsymbol

1.3 BNF und EBNF

- **BNF:** Grundform zur Definition von Grammatiken
- **EBNF:** Erweiterung mit regulären Ausdrücken, optionalen und wiederholten Konstrukten
- Beispiel:

  Expression ::= primary-Expression (operator primary-Expression)*

1.4 Grammatiktransformationen

- **Gruppierung**: Zusammenfassen gleicher LHS
- **Linksausklammern**: Gemeinsame Präfixe auslagern
- **Linksrekursion beseitigen**: $N ::= X|NY \Rightarrow N ::= X(Y)^*$

- **Ersetzung von Nicht-Terminalen**: falls nur eine Regel existiert

1.5 Parsing-Grundlagen

Parsing

Entscheidung, ob Eingabe zur Grammatik gehört und Aufbau des Syntaxbaumes.

- **Top-Down (z. B. rekursiver Abstieg):** Von Startsymbol zu Terminalen
- **Bottom-Up (z. B. Shift/Reduce):** Von Terminalen zur Wurzel

1.6 Top-Down Parsing

- Aufbau des Syntaxbaums von oben nach unten
- Expandiere jeweils das linke Nichtterminal
- LL(k): Grammatik, bei der mit k Lookahead-Tokens eindeutig entschieden werden kann
- **LL(1)** $\rightarrow$ wichtigster Fall für rekursiven Abstieg

1.7 Rekursiver Abstieg

Rekursiver Abstieg

Jedes Nichtterminal erhält eine Prozedur `parseN()`, deren Aufrufstruktur dem Parsebaum entspricht.

- **accept(t)** prüft aktuelles Token
- **acceptIt()** akzeptiert aktuelles Token ohne Prüfung
- **currentToken**: vom Scanner geliefert

1.8 Starter- und Folgemengen

- T: Menge der Terminale (Tokens, z.B. `id`, `+`)
- N: Menge der Nichtterminale (Variablen, z.B. `Expression`)
- ε: Das leere Wort (Epsilon)
- \$: End-of-File Marker (eof)

starters$[[X]]$ (First-Menge)

Menge aller Terminale, mit denen ein aus X abgeleiteter String beginnen kann.
Berechnung:

- **Ist $X \in T$ (Terminal):**
  $starters[[X]] = \{X\}$
- **Ist $X \in N$ (Nichtterminal) mit $X \rightarrow Y_1 Y_2 \ldots$:**
  - Füge $starters[[Y_1]] \setminus \{\varepsilon\}$ hinzu.
  - Falls $\varepsilon \in starters[[Y_1]]$, füge auch $starters[[Y_2]]$ hinzu (usw.).
- **Epsilon:** Falls $X \rightarrow \varepsilon$ existiert, ist $\varepsilon \in starters[[X]]$.

follow[[A]] (Folgemenge)

Menge aller Terminale, die in einer Satzform unmittelbar rechts von einem Nichtterminal A stehen können.
Regeln:

1. **Startsymbol S:** Enthält immer das Ende-Zeichen (\$).
2. **Rechter Nachbar ($B \to \alpha A \beta$):**
   Alles aus $starters[[\beta]]$ (außer ε) kommt zu $follow[[A]]$.
3. **Eltern-Vererbung ($B \to \alpha A$ oder $\beta \Rightarrow^* \varepsilon$):**
   Wenn A am Ende steht (oder β wegfallen kann), erbt A alles aus $follow[[B]]$.

Hinweis: Folgemengen enthalten niemals ε, können aber \$ enthalten.

LL(1)-Konfliktfreiheit

Eine Grammatik ist LL(1), wenn für jede Produktion mit Alternativen $A \to \alpha \mid \beta$ gilt:

- **Disjunkte Starter (Auswahl-Konflikt):**
  Die Alternativen dürfen nicht mit demselben Terminal beginnen.
  $starters[[\alpha]] \cap starters[[\beta]] = \emptyset$
- **Disjunkte Folge bei Epsilon (Nullable-Konflikt):**
  Falls $\alpha \Rightarrow^* \varepsilon$ (d.h. α kann verschwinden), darf die Folgemenge von A keinen gemeinsamen Start mit β haben.
  $starters[[\beta]] \cap follow[[A]] = \emptyset$

1.9 AST (Abstract Syntax Tree)

- Strukturierte Repräsentation des Programms
- Parser erzeugt AST-Knoten beim rekursiven Abstieg
- Jede Grammatikregel entspricht einer AST-Unterklasse

AST-Aufbau

- **Abstrakte Basisklasse:** `AST`
- **Subklassen:** `Command`, `Expression`, `Declaration`, `TypeDenoter`
- Terminalknoten (z. B. Identifier, Operator) speichern tatsächlichen Text

1.10 Scanner (Lexikalische Analyse)

Scanner

Wandelt Zeichen → Tokens anhand regulärer Ausdrücke (REs).

Aufgaben:

- Entfernt Whitespace, Kommentare
- Liefert `Token(kind, spelling, position)`
- Nutzt endlichen Automaten oder rekursiven Abstieg

Beispiel EBNF Mini-Triangle:

$$\begin{aligned} \text{Identifier} &::= \text{Letter (Letter} \mid \text{Digit)}^* \\ \text{Integer-Literal} &::= \text{Digit Digit}^* \\ \text{Operator} &::= + \mid - \mid * \mid / \mid < \mid > \mid = \end{aligned}$$

1.11 Automatisierung

- Scanner-Generatoren: **JLex**, **JFlex**
- Parser-Generatoren: **ANTLR (LL*)**, **JavaCC (LL(k))**

1.12 Typische Fehler

- Linksrekursion nicht entfernt
- Linksausklammern vergessen
- Anfangs-/Folgemengen überschneiden sich → nicht LL(1)
- Schlüsselwörter nicht vom Identifier getrennt

1.13 Zusammenfassung – Wichtigste Punkte

- CFG-Grundlagen (BNF/EBNF)
- Transformationen: Gruppierung, Linksausklammern, Rekursionsbeseitigung
- LL(1)-Parsing und Bedingungen
- Rekursiver Abstieg: Struktur und Methoden
- AST-Struktur: Klassenhierarchie, Terminal- und Nichtterminalknoten
- Scanner: REs, endliche Automaten, Schlüsselworterkennung