1 Kontextuelle Analyse

1.1 Einordnung und Ziele

Die kontextuelle Analyse ist die Phase zwischen Syntaxanalyse (Parsing) und Code-Generierung. Während der Parser nur die grammatikalische Korrektheit prüft (Kontextfreie Grammatik), prüft diese Phase Regeln, die vom Kontext abhängen.

- **Eingabe:** Abstrakter Syntaxbaum (AST).
- **Ausgabe:** **Dekorierter AST** (Knoten sind mit Typ- und Bindungsinformationen angereichert).
- **Aufgaben:**
  1. **Identifikation** (Identification): Zuordnung von Bezeichner-Verwendungen zu ihren Deklarationen (Geltungsbereiche prüfen).
  2. **Typprüfung** (Type Checking): Sicherstellen, dass Operatoren auf kompatible Typen angewendet werden.

1.2 Identifikation und Geltungsbereiche

1.2.1 Blockstrukturen

Programmiersprachen definieren **Geltungsbereiche** (Scopes), in denen Bezeichner sichtbar sind.

Arten von Blockstrukturen

- **Monolithisch** (z.B. BASIC): Ein einziger globaler Scope. Keine Namensdopplungen erlaubt.
- **Flach** (z.B. FORTRAN): Trennung in Global und Lokal.
- **Verschachtelt** (z.B. Triangle, Java, Pascal): Beliebig tiefe Schachtelung von Blöcken.

Regeln für verschachtelte Strukturen (Nested Scopes):

1. Ein Bezeichner darf innerhalb eines Blocks nur **einmal** deklariert werden.
2. Ein benutzter Bezeichner muss im aktuellen oder einem umschließenden (äußeren) Block deklariert sein.
3. **Verschattung (Hiding)**: Eine Deklaration in einem inneren Block verdeckt eine gleichnamige Deklaration in einem äußeren Block.

1.2.2 Die Identifikationstabelle (Symboltabelle)

Die Symboltabelle (in den Folien `IdentificationTable`) ist die zentrale Datenstruktur, um Deklarationen zu verwalten und effizient abzurufen.

Problem naiver Ansätze:

- *Liste:* Lineare Suche ist zu langsam ($O(n)$).
- *Einfache Map:* Kann keine Verschattung (gleicher Name in verschiedenen Scopes) abbilden.

Effiziente Implementierung (Triangle-Ansatz): Es wird eine Kombination aus Hash-Map und Stacks verwendet, um schnellen Zugriff ($O(1)$) und Scope-Verwaltung zu kombinieren.

Datenstrukturen der IdentificationTable

- `private Map<String, Stack<Attribute>> idents`
  Bildet Bezeichnernamen (String) auf einen Stapel von Attributen ab.
  - *Warum ein Stack?* Wenn Variable `x` global und lokal existiert, liegt die lokale (aktuelle) Definition oben auf dem Stack.
- `private Stack<List<String>> scopes`
  Verwaltet die Schachtelungsebenen. Jedes Element des Stacks ist eine Liste aller Bezeichner, die im *aktuellen* Scope deklariert wurden.
  - *Zweck:* Ermöglicht das schnelle Aufräumen (Löschen) aller Variablen eines Blocks, wenn dieser verlassen wird.

Algorithmus der Scope-Operationen:

1. **Scope öffnen** (`openScope`):
   - Lege eine neue, leere Liste auf den `scopes`-Stack.
   - Markiert den Beginn eines neuen Blocks (z.B. bei `LetCommand`).
2. **Eintrag hinzufügen** (`enter(id, attr)`):
   - Hole den Attribut-Stack für `id` aus `idents` (erstelle ihn, falls nicht existent).
   - Pushe das neue `attr` auf diesen Stack (Verschattung aktiv).
   - Füge `id` zur Liste hinzu, die oben auf `scopes` liegt (damit wir wissen, dass `id` zu diesem Scope gehört).
3. **Eintrag abrufen** (`retrieve(id)`):
   - Suche `id` in `idents`.
   - Wenn vorhanden: Gib das oberste Element des Stacks zurück (tiefste/aktuellste Verschachtelungsebene).
   - Wenn Stack leer/nicht vorhanden: Bezeichner nicht deklariert → Fehler.
4. **Scope schließen** (`closeScope`):
   - Poppe die oberste Liste von `scopes` (Liste der lokalen Variablen).
   - Durchlaufe diese Liste: Für jeden String `id` darin, poppe das oberste Element vom entsprechenden Stack in `idents`.
   - *Effekt:* Die lokalen Deklarationen sind "vergessen", vorherige (globale) Deklarationen liegen wieder oben auf den Stacks in `idents`.

1.3 Attribute und AST-Dekoration

Was genau wird in der Symboltabelle gespeichert?

- **Klassischer Ansatz:** Eigene Klasse `Attribute` mit Feldern für `Kind` (Var, Const, Proc) und `Type` (Int, Bool). Wird bei komplexen Typen (Arrays, Records) schnell unhandlich.
- **AST-Ansatz (Triangle):** Da im AST (genauer: im Deklarations-Teilbaum) bereits alle Infos stehen, speichert man in der Symboltabelle einfach **Referenzen auf die AST-Knoten**.

Dekoration

Der Prozess der Kontextanalyse reichert den AST an:
- **Bei der Deklaration:** Der AST-Knoten der Deklaration wird in die Symboltabelle eingetragen.
- **Bei der Verwendung (Applied Occurrence):** Der AST-Knoten der Verwendung (z.B. `Identifier`) erhält einen Zeiger (`public Declaration decl`) auf den AST-Knoten seiner Deklaration.

1.4 Typprüfung (Type Checking)

- **Ziel:** Sicherstellen, dass Operationen mit validen Typen ausgeführt werden (z.B. `if` benötigt `Boolean`).
- **Vorgehen:** Bottom-Up Verfahren (von den Blättern zur Wurzel).
- **Statische Typisierung:** Findet zur Compile-Zeit statt. Jeder Ausdruck hat einen festen Typ.

Ablauf am Beispiel `n + 1`:

1. `IntLit (1)`: Typ ist direkt bekannt (`Integer`).
2. `Ident (n)`: Typ wird aus der Symboltabelle geholt (via Link zur Deklaration).
3. `BinaryExpr (+)`: Prüft, ob Operator `+` für `Integer` × `Integer` definiert ist und was der Rückgabetyp ist.

1.5 Implementierung: Das Visitor Pattern

Um die Logik der Kontextanalyse nicht in den AST-Klassen zu verstreuen, wird das **Visitor Pattern** verwendet. Dies trennt Datenstruktur (AST) von Algorithmus (Checker).

- **Prinzip:** `astNode.visit(visitor, arg)`.
- **Double Dispatch:** Der Knoten ruft zurück auf `visitor.visitSpecificNode(this, arg)`.
- **Vorteil:** Neue Analysen (z.B. CodeGen) können hinzugefügt werden, ohne AST-Klassen zu ändern.

Spezialisierte Visitors in Triangle: Statt eines einzigen Visitors werden spezialisierte Unterklassen verwendet, um Typsicherheit bei Rückgabewerten zu erhöhen:

- `ExpressionChecker`: Liefert `TypeDenoter` (da Ausdrücke einen Typ haben).
- `CommandChecker`: Liefert `Void` (Befehle haben keinen Typ).
- `DeclarationChecker`: Trägt Bezeichner in die Symboltabelle ein.

1.6 Standardumgebung (Standard Environment)

Sprachen haben vordefinierte Typen (`Integer`, `Boolean`) und Funktionen (`put`, `get`).

- Diese sind nicht Teil der Grammatik.
- **Lösung:** Vor dem Start der eigentlichen Analyse wird die Symboltabelle mit "künstlichen" Deklarationen befüllt (z.B. wird ein AST-Fragment für eine Konstante `true` erzeugt und eingetragen).

1.7 Typäquivalenz

Wann gelten zwei Typen als gleich?

Äquivalenz-Arten

- **Strukturelle Äquivalenz** (Triangle): Typen sind gleich, wenn ihre Struktur identisch ist.
  Beispiel: `array 8 of Char` ist kompatibel mit `array 8 of Char`.
- **Namensäquivalenz** (Pascal, Ada): Typen sind nur gleich, wenn sie denselben Typnamen haben. Jede Typ-Definition erzeugt einen neuen, inkompatiblen Typ.