1 Kontextuelle Analyse

1.1 Einordnung und Ziele

Kontextuelle Analyse

Die Phase zwischen Syntaxanalyse (AST-Erstellung) und Code-Generierung. Sie prüft Regeln, die nicht durch die kontextfreie Grammatik abgedeckt sind.

- **Eingabe:** Abstrakter Syntaxbaum (AST).
- **Ausgabe:** **Dekorierter AST** (mit Typ- und Bindungsinformationen).

Hauptaufgaben:

1. **Identifikation** (Identification): Zuordnung von Bezeichner-Verwendungen zu ihren Deklarationen (Scope-Regeln).
2. **Typprüfung** (Type Checking): Sicherstellen der Typkompatibilität von Operatoren und Operanden.

1.2 Identifikation und Geltungsbereiche (Scopes)

1.2.1 Blockstrukturen

- **Monolithisch** (z. B. BASIC): Ein globaler Scope. Bezeichner müssen programmweit eindeutig sein.
- **Flach** (z. B. FORTRAN): Zwei Ebenen (Global und Lokal).
- **Verschachtelt** (z. B. Pascal, Triangle, Java): Beliebige Schachtelungstiefe.

Regeln für verschachtelte Scopes

- **Deklaration:** Ein Bezeichner darf im selben Block nur einmal deklariert werden.
- **Sichtbarkeit:** Ein Bezeichner ist sichtbar, wenn er im aktuellen oder einem umschließenden Block deklariert wurde.
- **Verschattung** (Hiding): Eine lokale Deklaration verdeckt eine gleichnamige Deklaration in einem äußeren Block.

1.2.2 Symboltabelle (Identification Table)

Datenstruktur zur effizienten Zuordnung von Namen zu Attributen.

Implementierung (Triangle-Ansatz): Um lineare Suche zu vermeiden, wird eine Kombination aus Hash-Map und Stacks verwendet:

1. `Map<String, Stack<Attribute>>`: Der Bezeichner ist der Key. Der Value ist ein Stack aller aktuellen Deklarationen dieses Namens (oberstes Element = aktuell sichtbare Bindung).
2. `Stack<List<String>> scopes`: Verwaltet die Scope-Ebenen. Beim Öffnen eines Scopes (`openScope`) wird eine neue Liste aufgelegt. Alle darin deklarierten Variablen werden vermerkt, um sie beim Schließen (`closeScope`) effizient aus der Map zu entfernen.

1.3 Attribute und AST-Dekoration

Attribute

Gespeicherte Eigenschaften eines Bezeichners:
- **Art**: Variable, Konstante, Typ, Prozedur, Funktion.
- **Typ**: Verweis auf die Typstruktur.
- Laufzeitinfos: Adressen, Offsets (für Code-Gen).

Speicherstrategie: Statt komplexe Attribut-Klassen zu bauen, nutzt man den AST selbst. Die Symboltabelle speichert **Referenzen auf die Deklarations-Knoten** im AST.

Ablauf der Dekoration:

- Bei der Deklaration (`Binding Occurrence`): Eintrag in Symboltabelle.
- Bei der Verwendung (`Applied Occurrence`): Suche in Symboltabelle → AST-Knoten der Verwendung erhält Zeiger auf AST-Knoten der Deklaration.

1.4 Typprüfung (Type Checking)

Statische vs. Dynamische Typisierung

- **Statisch** (Triangle, Java): Prüfung zur Compile-Zeit. Jeder Ausdruck hat einen fixen Typ. Sicherer und performanter.
- **Dynamisch** (Python, Lisp): Prüfung zur Laufzeit. Flexibler, aber fehleranfälliger.

Algorithmus (Bottom-Up): Die Prüfung erfolgt rekursiv von den Blättern zur Wurzel:

1. **Blätter:** Typen von Literalen (z. B. `1` → Integer) sind bekannt. Variablen-Typen kommen aus der Identifikation (Symboltabelle).
2. **Knoten:** Typ eines Ausdrucks E_1 *op* E_2 wird aus den Typen der Kinder E_1, E_2 und der Signatur von *op* abgeleitet.
3. **Kontext-Check:** Z. B. muss die Bedingung in `if E ...` vom Typ `Boolean` sein.

1.5 Implementierung mit dem Visitor-Pattern

Um die Logik (Typprüfung) von der Datenstruktur (AST) zu trennen, wird das **Visitor-Pattern** verwendet. Dies verhindert das Aufblähen der AST-Klassen.

Funktionsweise Visitor

- **Double Dispatch:** Der AST-Knoten ruft `v.visit(this)` auf, der Visitor führt dann die spezifische Methode (z. B. `visitAssignCommand`) aus.
- **Generics:** `Visitor<RetTy, ArgTy>` ermöglicht flexible Rückgabe- und Argumenttypen.

Spezialisierte Checker (Best Practice): Statt eines riesigen Visitors gibt es spezialisierte Unterklassen von `VisitorBase`:

- `ExpressionChecker`: Liefert `TypeDenoter` zurück (da Ausdrücke einen Typ haben).
- `CommandChecker`: Liefert `Void` zurück (da Befehle keinen Typ haben).

Beispiele für Visitor-Methoden:

- `visitAssignCommand`: Prüft, ob LHS eine Variable ist und ob $Type(LHS) == Type(RHS)$.
- `visitLetCommand`: Ruft `idTable.openScope()` auf, besucht Deklarationen, besucht Body, ruft `idTable.closeScope()` auf.

1.6 Standardumgebung (Standard Environment)

Vordefinierte Bezeichner (z. B. `Integer`, `true`, `put`) sind nicht Teil der Grammatik, sondern werden **vor** der Analyse in die Symboltabelle geladen.

Realisierung: Es werden künstliche AST-Fragmente erstellt (z. B. eine `ConstDeclaration` für `true`), die dann in den globalsten Scope eingefügt werden.

1.7 Typäquivalenz

Wann sind zwei Typen T_1 und T_2 gleich?

Äquivalenzarten

- **Strukturelle Äquivalenz** (Triangle): Typen sind gleich, wenn ihre Struktur identisch ist (z. B. Array gleicher Länge und gleicher Elementtyp).
- **Namensäquivalenz** (Pascal, Ada): Typen sind nur gleich, wenn sie denselben Typnamen haben. Jede `type`-Definition erzeugt einen neuen, inkompatiblen Typ.

Beispiel Triangle (Strukturell):

- `array 8 of Char` == `array 8 of Char` (Äquivalent)
- `record a:Int end` ≠ `record b:Int end` (Nicht äquivalent, da Bezeichner zur Struktur gehören).

1.8 Zusammenfassung des Gesamtablaufs

1. Aufbau der **Standardumgebung**.
2. **Tiefensuche** über den AST (Visitor).
3. Bei Eintritt in Block (`Let`): `openScope`.
4. Bei Deklaration: Eintrag in Symboltabelle + Prüfung auf Dopplung.
5. Bei Verwendung: Lookup in Symboltabelle + **Identifikation** (Setzen des Links).
6. Berechnung der Typen von unten nach oben (**Typprüfung**).
7. Bei Austritt aus Block: `closeScope`.