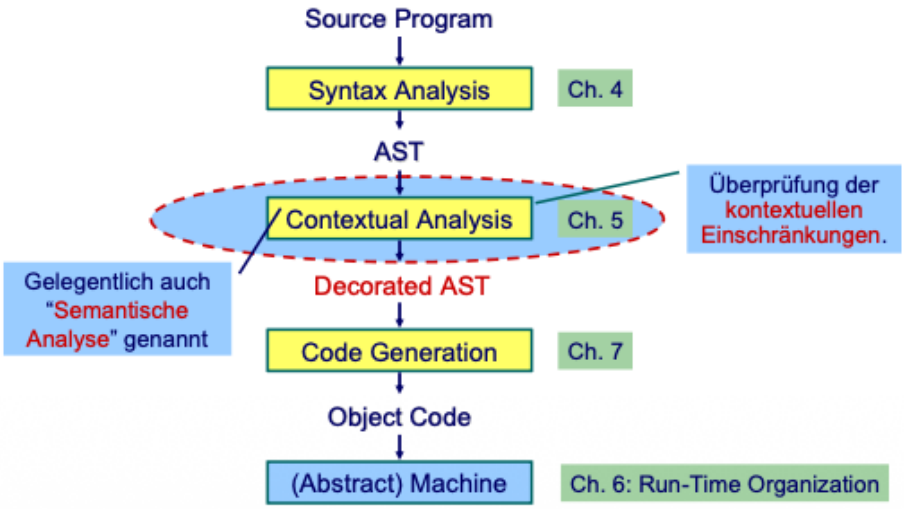
1 Kontextuelle Analyse

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).

1.1 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).

1.1.1 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.

1.1.2 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).

1.2 Symboltabellen (Identification Tables)

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

1.2.1 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.

1.2.2 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.

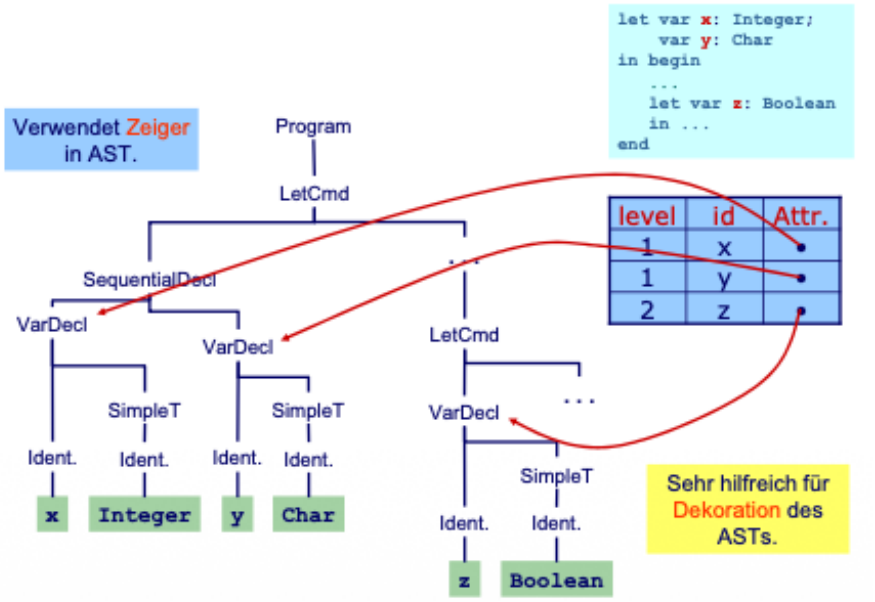
Operationen der Symboltabelle

- `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.

1.3 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.



1.4 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).

1.4.1 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).

1.4.2 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.

1.5 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).

1.5.1 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.

1.5.2 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
}
```

1.6 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`).

1.7 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`.