

# Dokumentation der Laborarbeit: Systemnahes Programmieren Teil 2

Boris Bopp (Matr. Nr: 53690)

Maksim Sunjajkin (Matr. Nr: 49946)

Henrik Erhart (Matr. Nr: 54323)

## Vorwort

Im ersten Teil dieses Projekts wurde bereits ein Scanner erstellt, der eine Sequenz von Tokens erzeugt. In diesem Teil soll nun aus dieser Sequenz von Tokens ein ParseTree erstellt werden und anschließend damit ein TypeCheck gemacht werden. Läuft dieser ohne erkannte Fehler durch soll danach der ausführbare Maschinencode erzeugt werden.

## Inhalt

Vorwort .....	2
Architektur.....	4
Parser.....	4
ParseTree.....	4
Visitor .....	5
TypeCheck .....	5
MakeCode .....	5
PrintTreeVisitor .....	6
Verzeichnisse und Dateien .....	7
Aufgetretene Probleme und Lösungen .....	8

# Architektur

## Parser

Der Parser verarbeitet die von Scanner gelieferte Sequenz von Token und erstellt daraus den ParseTree. Syntaktische Fehler werden während diesem Prozess erkannt und auf der Konsole ausgegeben. Dazu wird die Methode des Rekursiven Abstiegs benutzt. Bei dieser wird der ParseTree rekursiv nach den Regeln der gegebenen Grammatik aufgebaut. Dabei wird auf jeder Rekursionsebene die möglichen Folgeübergänge anhand des nächsten gelieferten Tokens überprüft werden. Da die Grammatik der Programmiersprache Linkseindeutig ist kann anhand des nächsten Tokens der richtige Übergang gewählt werden. Ist der richtige Übergang gefunden, werden dem aktuellen Knoten die entsprechenden Kinderknoten rekursiv berechnet und hinzugefügt. Falls laut Grammatik ein bestimmtes Token gefordert wird, dieses aber nicht das aktuelle ist, so wird ein Fehler ausgegeben und der Parse-Vorgang wird abgebrochen. Epsilon-Übergänge werden dadurch erkannt, dass keine anderen Übergänge ausgeführt werden können. Bei Epsilon-Übergängen wird der aktuelle Knoten mit einem „isEpsilon“ – Flag versehen.

## ParseTree

Der ParseTree setzt sich aus Knoten und Blättern zusammen. Die Blätter bilden dabei immer die tiefste Ebene des Baumes und können keine weiteren Kinder haben. Dafür bekommen Sie jeweils ein Token zugeordnet. Knoten haben in der Regel weitere Kinder. Die einzige Ausnahme sind Knoten, beim Parsen zu einem Epsilon-Übergang führen.

## Visitor

Die Funktionalität der Codegenerierung und des Typechecks wurden durch das „Besucher“-Pattern ausgekoppelt. Dies führt dazu, dass weitere Funktionalität wie z.B. eine Optimierung integriert werden kann, ohne den ParseTree zu verändern. Somit wird das „Open-Closed principle“ des SOLID Regelwerks umgesetzt. In Zukunft würden wir das „Visitor“-Pattern in Kombination mit spezialisierten Node-Klassen verwenden, da der Double-Dispatch die Switch-Verzweigungen der Besucher ersetzt. Ebenso könnte man die Navigationslogik der Besucher in eine eigene Klasse auslagern.

## TypeCheck

Beim TypeCheck wird der zuvor aufgestellte ParseTree rekursiv abgearbeitet, dabei ist es die Aufgabe des TypeChecks sowohl die Knoten des Baumes auf ihre entsprechenden Typen zu überprüfen als auch die Informationen von Identifiern in der Symboltabelle zu evaluieren. Der TypeCheck wird mithilfe des Visitor Pattern realisiert. Sollte ein Fehler festgestellt werden wird eine Ausnahme („Exception“) geworfen und somit die Überprüfung abgebrochen. Im Hauptprogramm wird diese Ausnahme gefangen und die Art des Fehlers, Zeile und Reihe wo dieser auftrat ausgegeben.

Aufgabe des Typechecks ist:

- das Verhindern von mehrfacher Variablendeklaration
- Verhindern eines Arrayzugriffs auf primitive Variablen
- Verhindern eines Arrays als primitiven Datentyp (ohne Angabe des Index)
- Bedingte Typüberprüfung. Entfällt, da die Grammatik lediglich Ganzzahlige Variablen als Datentyp kennt.

## MakeCode

Der aufgebaute Parse-Tree wird gemäß Vorlage im Skript in den vorgegebenen Maschinencode übersetzt. Dieser wird in die als Argument übergebene Textdatei geschrieben.

## PrintTreeVisitor

Zur Visualisierung des ParseTrees haben wir einen Visitor implementiert, der diesen auf der Konsole ausgibt. Dieser wird durch eine „#ifdef“-Präprozessoranweisung nur beim Debuggen ausgeführt und hat uns die Fehlersuche vereinfacht.

Der Parsetree wird Beispielweise folgendermaßen ausgegeben

```
int n;  
int[3] m;  
n := 3 + 4;  
n := 3*n--5;  
write(n);
```

```
Prog  
  Decls  
    Decl  
      Leaf (Keyword): Int  
      (eps)Array  
      Leaf (Identifier): Identifier  
      Leaf (Keyword): Semicolon  
    Decls  
      Decl  
        Leaf (Keyword): Int  
        Array  
          Leaf (Keyword): Bracket '['  
          Leaf (INTEGER): Integer  
          Leaf (Keyword): Bracket ']'  
          Leaf (Identifier): Identifier  
          Leaf (Keyword): Semicolon  
        (eps)Decl  
      Statements  
        StatementIdent  
          Leaf (Identifier): Identifier  
          (eps)Index  
          Leaf (Keyword): Assign  
        Exp  
          Exp2Int  
            Leaf (INTEGER): Integer  
          OpExp  
            Op  
              Leaf (Op): Plus  
            Exp  
              Exp2Int  
                Leaf (INTEGER): Integer  
              (eps)OpExp  
            Leaf (Keyword): Semicolon  
          Statements  
            StatementIdent  
              Leaf (Identifier): Identifier  
              (eps)Index  
              Leaf (Keyword): Assign  
            Exp  
              Exp2Int  
                Leaf (INTEGER): Integer  
              OpExp  
                Op  
                  Leaf (Op): Star  
                Exp  
                  Exp2Ident  
                    Leaf (Identifier): Identifier  
                    (eps)Index  
                  OpExp  
                    Op  
                      Leaf (Op): Minus  
                    Exp  
                      Exp2Minus  
                        Leaf (Keyword): Minus  
                        Exp2Int  
                          Leaf (INTEGER): Integer  
                        (eps)OpExp  
                      Leaf (Keyword): Semicolon  
                    Statements  
                      StatementWrite  
                        Leaf (Keyword): Write  
                        Leaf (Keyword): Bracket '['  
                      Exp  
                        Exp2Ident  
                          Leaf (Identifier): Identifier  
                          (eps)Index  
                        (eps)OpExp  
                      Leaf (Keyword): Bracket ']'  
                    Leaf (Keyword): Semicolon  
                  (eps)Statements
```

## Verzeichnisse und Dateien

Die bisherige Verzeichnisstruktur wurde beibehalten. Zusätzlich wird für die Parser-Komponente ein weiteres Verzeichnis erstellt. Das oberste makefile wurde entsprechend angepasst um nicht mehr den Scanner, sondern den Parser zu erstellen.

```
|--  Parser
    |--  debug/
    |--  includes/
    |--  objs/
    |--  src/
    |--  makefile
```

## Aufgetretene Probleme und Lösungen

Bei der Implementierung des Parsers sind uns folgende Fehler im ersten Aufgabenteil aufgefallen:

- Tabulatoren wurden nicht als Whitespace sondern als „Unknown“-Token erkannt. Dies ließ sich einfach beheben, indem die Erkennung der Zeichen im Automaten erweitert wurde.

Bei der Implementierung des Parsers sind folgende Probleme aufgetreten:

- Änderung des Node-Types im Typecheck gemäß Vorlage, hat zum Fehlerfall in der Codegenerierung geführt. Dies wurde behoben, indem der Typecheck nur noch lesend auf den Typ der Nodes zugreift und gegebenenfalls die Kinder der Nodes vorher auf Epsilon-Übergänge überprüft.
- Codegenerierung: Die Generierung der Labelbezeichnungen wurde auf dem Stack ausgeführt. Dies führte zum Freigeben des Speichers und war entsprechend nicht zielführend. Deshalb wurde die bereits implementierte Funktion „strdup“ verwendet, um den Speicher auf den Heap zu kopieren.