# Systemnahes Programmieren – Dokumentation

**Hochschule Karlsruhe – Technik und Wirtschaft**

Richard Gottschalk, 40365 Tobias Harms, 42894

Sommersemester 2015

# Inhaltsverzeichnis

**Abbildungsverzeichnis** **3**

**Listings** **4**

**1 Einführung** **5**

**2 Scanner** **6**
2.1 Automat . . . 6
2.1.1 Aufbau des Automaten . . . 6
2.1.2 Ablauf . . . 6
2.2 Buffer . . . 8
2.2.1 Aufbau . . . 8
2.2.2 O_DIRECT und posix_memalign . . . 8
2.2.3 Implementierung . . . 10
2.3 Symboltabelle . . . 11
2.3.1 Implementierung . . . 11
2.4 Scanner . . . 12
2.4.1 Initialisierung . . . 12
2.4.2 nextToken() . . . 12

**3 Parser** **14**
3.1 Parse-Tree . . . 14
3.1.1 Type-Check . . . 15
3.1.2 Code-Erzeugung . . . 16
3.2 Parser . . . 16

# Abbildungsverzeichnis

1.1 Die Grammatik der zu erkennenden Sprache . . . . . 5

2.1 Grafische Darstellung des Automaten . . . . . 7

3.1 Klassendiagramm des Compilers . . . . . 19

# Listings

Lst. 2.1 Buffer: Initialisierung und Lesen aus Datei . . . . . 9
Lst. 2.2 Buffer: Tauschen der Buffer-Teile . . . . . 10

Lst. 3.1 Parser: IParseTree-Interface . . . . . 14
Lst. 3.2 Parser: Klassenstruktur von Decl_I . . . . . 15
Lst. 3.3 Parser: Code-Erzeugung des Zuweisung-Statements . . . . . 16
Lst. 3.4 Parser: Methode parse() . . . . . 16
Lst. 3.5 Parser: Methode DECL() . . . . . 17

# 1 Einführung

Im Labor „Systemnahes Programmieren“ geht es darum einen Compiler zu programmieren, welcher aus einer gegebenen Grammatik einen Assembler-ähnlichen Code erzeugt. Für diesen Code gibt es einen Interpreter, sodass die Programme dann tatsächlich ausgeführt werden können.

| | | |
|---|---|---|
| *PROG* | ::= | *DECLS STATEMENTS* |
| *DECLS* | ::= | *DECL* ; *DECLS* \| $\varepsilon$ |
| *DECL* | ::= | int *ARRAY identifier* |
| *ARRAY* | ::= | [ *integer* ] \| $\varepsilon$ |

**Hinweis:**
- normaleTerminale (Schlüsselworte) sind klein, **fett** und *rot*
- kleine, **fette**, ***kursive*** und ***blaue*** Terminale stehen für Konstanten bzw. Bezeichner (3, 3.14, x,...).
- Nichtterminale sind groß und *KURSIV* gedruckt

| | | |
|---|---|---|
| *STATEMENTS* | ::= | *STATEMENT* ; *STATEMENTS* \| $\varepsilon$ |
| *STATEMENT* | ::= | *identifier INDEX* := *EXP* \| write( *EXP* ) \| read ( *identifier INDEX*) \| {*STATEMENTS*} \| if ( *EXP* ) *STATEMENT* else *STATEMENT* \| while ( *EXP* ) *STATEMENT* |
| *EXP* | ::= | *EXP2 OP_EXP* |
| *EXP2* | ::= | ( *EXP* ) \| *identifier INDEX* \| *integer* \| - *EXP2* \| ! *EXP2* |
| *INDEX* | ::= | [ *EXP* ] \| $\varepsilon$ |
| *OP_EXP* | ::= | *OP EXP* \| $\varepsilon$ |
| *OP* | ::= | + \| - \| * \| : \| < \| > \| = \| <:> \| & |

Abbildung 1.1: Die Grammatik der zu erkennenden Sprache

Für diese Aufgabe waren einige Voraussetzungen gegeben:

- Die Verwendung von Linux
- Programmieren in C++
- Die Verwendung von GNU make zum Build-Management
- Die Verwendung von O_DIRECT und posix_memalign beim Buffer (siehe Kap. 2.2.2)

Die Aufgabe wurde in zwei Aufgabenteile zerlegt, den *Scanner* (Kapitel 2) und den *Parser* (Kapitel 3).

# 2 Scanner

Der Programmteil *Scanner* zerlegt den Quellcode in Tokens, welche dann vom Parser weiter verarbeitet werden. Der Scanner muss sich also um das Einlesen aus der Quelldatei (Buffer, siehe Kapitel 2.2), die Erkennung von Tokens (Automat, siehe Kapitel 2.1) und die Speicherung von Identifiern (Symboltabelle, siehe Kapitel 2.3) kümmern. Die Aufgabe des Scanners ist also die *lexikalische Analyse* des Quellcodes.

## 2.1 Automat

Der Automat hat die Aufgabe aus einzeln eingehenden Zeichen und mit Hilfe interner Zustandsübergänge die richtige Bedeutung von einzelnen Zeichenketten zu erkennen.

### 2.1.1 Aufbau des Automaten

Die Zustandsübergänge sind in Form eines 2D Arrays vorhanden, wobei die 1. Dimension den Ursprungszustand und die 2. Dimension den Zielzustand repräsentiert. Jedes Zeichen hat in dieser Übergangstabelle einen stellvertretenden Eintrag. Buchstaben sowie Zahlen werden vom Automaten jeweils gruppiert. Der Startzustand zählt ebenfalls als Rücksetzpunkt, wenn auf einen Zustand mit gegebenen Zeichen kein Folgezustand gefunden wird. Leerzeichen, Zeilenumbrüche führen in allen Zuständen außer innerhalb von Kommentaren zu einem Übergang zum Rücksetzpunkt. Unbekannte Zeichen werden innerhalb von Kommentaren akzeptiert, außerhalb von Kommentaren aber als unbekanntes Zeichen repräsentiert.

### 2.1.2 Ablauf

Für ein eingehendes Zeichen in `int readChar(char c)` wird zuerst der neue Zustand ermittelt. Dies geschieht intern mit der Methode `nt getNextState(char c)`. Diese Methode sucht zuerst mittels `int getTransitionColumn(char c)` den vom aktuellen Zustand abhängigen nächsten Zustand.

Nun wird in `int readChar(char c)` geprüft, ob dieser neue Zustand nicht der Startzustand ist.

Handelt es sich bei dem neuen Zustand nicht um den Startzustand, dann handelt es sich um einen „validen“ Übergang im Sinne unserer Grammatik.

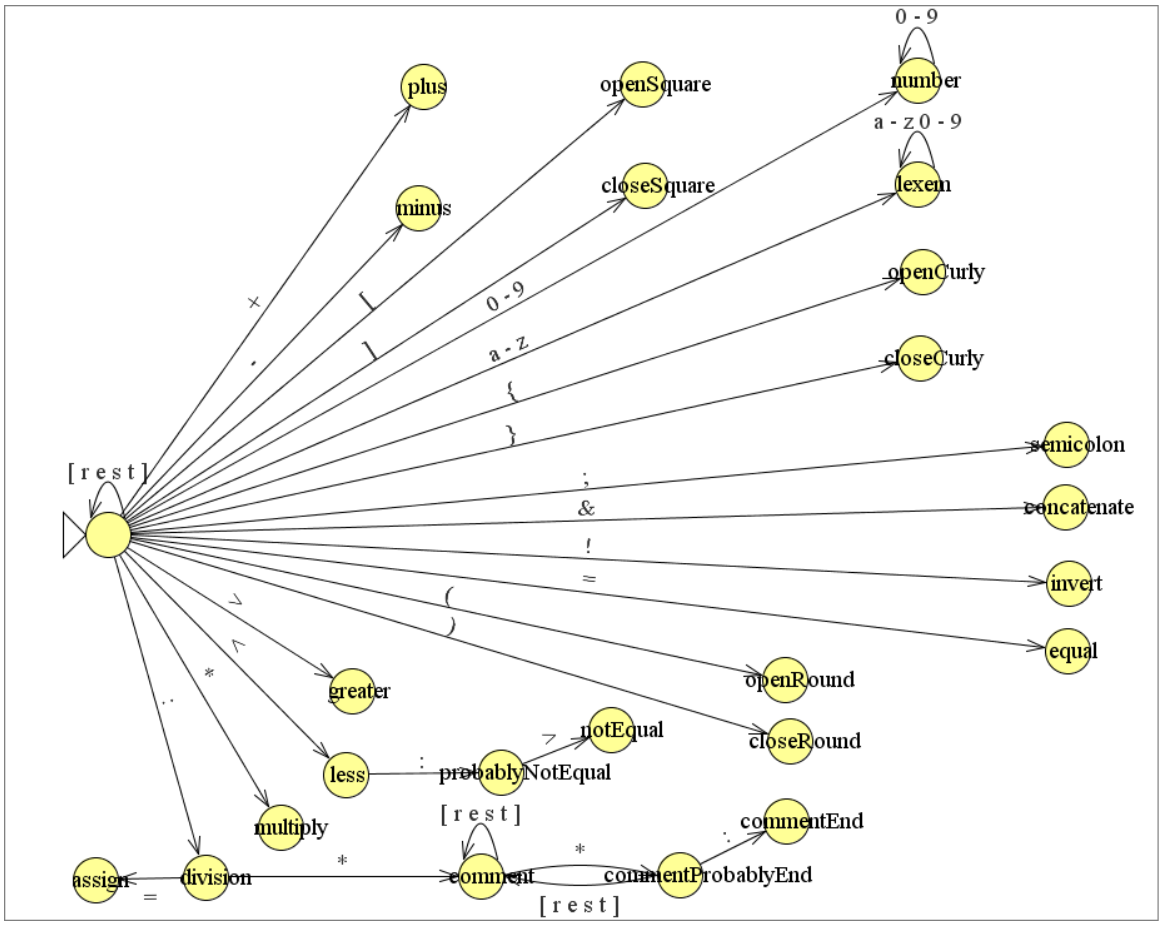


Abbildung 2.1: Grafische Darstellung des Automaten

### Vorgehen bei validem Übergang

`lastFinalStateCounter` wird um eins inkrementiert um den Offset zum letzten validen Zustand zu behalten. Zudem nimmt der Automat den neuen Zustand an. Der Automat gibt in diesem Fall den Wert 0 zurück und sagt somit, dass alles in Ordnung ist.

### Vorgehen bei nicht validem Übergang

Sollte sich der Automat noch in einem finalen Zustand befinden, dann gibt der Automat den aktuellen Zustand zurück und setzt sich wieder auf den Startzustand. Der Rückgabewert in diesem Fall ist eine positive Zahl und bedeutet somit, dass er einen validen Zustand erreicht hat, bevor er das letzte Zeichen gelesen hat.

Sollte der Automat aber nicht in einem finalen Zustand sein, dann gibt der Automat den negativen `lastFinalStateCounter` Wert zurück und setzt sich selbst auf den letzten

erreichten Finalen Zustand. Somit gibt der Automat eine negative Zahl zurück, welche die Distanz zum letzten validen Zeichen repräsentiert.

## 2.2 Buffer

Die Aufgabe des Buffers ist es die Eingabedatei einzulesen und dem Scanner Zeichen für Zeichen zu übergeben. Dabei soll die Pufferung allein durch das Programm abgewickelt werden, die automatische Pufferung durch das Betriebssystem wird also abgeschaltet. Außerdem muss der Buffer die Möglichkeit bieten Zeichen auch zurückzunehmen.

### 2.2.1 Aufbau

Der Buffer besteht intern aus zwei unabhängigen Speicherbereichen einer definierten Größe (Buffersize). Bei der Initialisierung wird der erste Speicherbereich mit dem Anfang der Quelldatei gefüllt und bei jedem Lesezugriff auf den Buffer wird der zwischengespeicherte Inhalt der Datei ausgegeben. Falls beim Lesen aus dem Buffer über das Limit des Speicherbereichs hinaus gelesen werden soll, wird der anschließende Bereich in den zweiten Speicherbereich geladen und der Puffer-Zeiger umgesetzt.

Diese Zweiteilung des Buffers hat den Vorteil, dass falls beim Zurücklesen von Zeichen der Anfang des Speicherbereichs überschritten werden würde, nichts aus der Datei geladen werden muss, da der vorherige Teil der Datei noch im zweiten Speicherbereich zwischengespeichert ist und der Zugriff damit direkt erfolgen kann.

### 2.2.2 O_DIRECT und posix_memalign

Um die Pufferung des Betriebssystems zu deaktivieren, muss beim Öffnen der Datei das `O_DIRECT`-Flag gesetzt werden:

```
1  this->fileDescriptor = open(path, O_RDONLY | O_DIRECT)
```

Leider führt die Verwendung von O_DIRECT häufig zu Problemen. So müssen folgende Vorraussetzungen gegeben sein, damit der `open`-Befehl ohne Fehler abläuft:

- Verwendung eines Linux-Betriebssystems
- Die zu öffnende Datei muss auf einer lokalen Festplatte liegen (keine Netzwerkfreigabe o. Ä.)
- Das Betriebssystem darf nicht virtualisiert sein

Des Weiteren muss beim Lesen aus einer mit O_DIRECT geöffneten Datei darauf geachtet werden, dass der Zielbuffer entsprechend ausgerichtet ist und stets nur ein Vielfaches von 512 Bytes gelesen wird.

Bei uns ergibt sich aus diesen Gründen folgende Implementierung für das Öffnen einer Datei, allozieren des Speichers und das Lesen aus der Datei:

```
1  #define BUFFERSIZE (1024 * (int)sizeof(char))
2
3  Buffer::Buffer(const char *path) {
4      if ((this->fileDescriptor = open(path, O_RDONLY | O_DIRECT)) < 0) {
5      perror("Oeffnen fehlgeschlagen");
6      exit(-1);
7    }
8    // Buffer 1 initialisieren (Platz fuer BUFFERSIZE + 1 chars)
9    allocMem(&memptr1, BUFFERSIZE, BUFFERSIZE + sizeof(char));
10   // ...
11 }
12
13 // Inline Methode, die Speicher alloziert, nur zur Uebersichtlichkeit
       ausgelagert.
14 inline void Buffer::allocMem(void **mempt, int align, int size) {
15   if (posix_memalign(mempt, align, size) != 0) {
16     perror("Speicher allozieren fehlgeschlagen");
17     exit(-1);
18   }
19 }
20
21 // Lese count chars in buf, gibt die Anzahl der gelesenen Bytes zurueck
22 int Buffer::readBytes(char *buf, int count) {
23   if (fileDescriptor > 0) {
24
25     int bytesRead = read(this->fileDescriptor, buf, count);
26
27     if (bytesRead < 0) {
28       perror("Fehler beim lesen");
29       exit(-1);
30     }
31
32     // Wenn weniger Bytes zurueckgeliefert wurden als angefordert wurden
33     // (Dateiende ist nahe), setze EOF-Markierer hinter letztes gelesenen
           char
34     if (bytesRead < BUFFERSIZE) buf[bytesRead] = '\0';
35     return bytesRead;
36   }
37   return -1; // Falls kein fileDescriptor, sollte nicht auftreten
38 }
```

Listing 2.1: Buffer: Initialisierung und Lesen aus Datei

Da O_DIRECT am Anfang häufiger zu Problemen führte, wurde es erst eingeschaltet, als der Rest des Buffers implementiert war. Nachdem alle obigen Punkte allerdings erfüllt waren, funktionierte alles wie vorgesehen.

Das folgende Zitat fasst unsere Erfahrungen mit O_DIRECT auf amüsante Weise erstaunlich gut zusammen:

> "The thing that has always disturbed me about O_DIRECT is that the whole interface is just stupid, and was probably designed by a deranged monkey on some serious mind-controlling substances."
>
> – Linus Torvalds, aus der `Linux Programmer's Manual` zu `open(...)`

### 2.2.3 Implementierung

Der Buffer steht, wie oben beschrieben, aus zwei getrennten Speicherbereichen (in diesem Fall: der Größe 1025 Bytes), auf welche, um die Verarbeitung zu vereinfachen, zwei Char-Pointer zeigen. Mit Hilfe dieser Pointer kann über den Arrayzugriffsoperator auf die einzelnen Zeichen zugegriffen werden.

```
1 return (*this->currentBuffer)[this->currentBufferIndex];
```

Das Tauschen der einzelnen Buffer-Teile stellte sich am Ende als keine große Schwierigkeit mehr heraus.

```
1  // Pruefung, ob ein neuer Buffer geladen werden muss
2  if (this->currentBufferIndex < 0) { //Buffer davor laden, BufferIndex
       wuerde sonst negativ werden
3    // Buffer tauschen
4    if (this->currentBuffer == &this->buffer1) {
5      this->currentBuffer = &this->buffer2;
6    } else if (this->currentBuffer == &this->buffer2) {
7      this->currentBuffer = &this->buffer1;
8    }
9    // Index ans Ende des Buffers setzen
10   this->currentBufferIndex = BUFFERSIZE-1;
11 } else if (this->currentBufferIndex >= BUFFERSIZE) { //Naechsten Buffer
       laden, BufferIndex wuerde sonst ueberlaufen
12   if (this->currentBuffer == &this->buffer1) {
13     if (!this->memptr2) { // Buffer 2 noch nicht initialisiert
14       allocMem(&this->memptr2, BUFFERSIZE, BUFFERSIZE + sizeof(char));
15       this->buffer2 = (char*)this->memptr2;
16       this->buffer2[BUFFERSIZE] = '\0';
17     }
18     // Buffer 2 vollladen
19     this->readBytes(this->buffer2, BUFFERSIZE);
20     // Aktueller Buffer ist Buffer 2
```

```
21        this->currentBuffer = &this->buffer2;
22      } else if (this->currentBuffer == &this->buffer2) {
23        // Buffer 1 volladen
24        this->readBytes(this->buffer1, BUFFERSIZE);
25        // Aktueller Buffer ist Buffer 1
26        this->currentBuffer = &this->buffer1;
27      }
28      // BufferIndex an den Anfang
29      this->currentBufferIndex = 0;
30  }
```

Listing 2.2: Buffer: Tauschen der Buffer-Teile

## 2.3 Symboltabelle

Die Symboltabelle ist eine Sammlung von bekannten Lexemen inklusive ihrer Token. Dabei entspricht die Symboltabelle einer Key-Value-Map mit dem Lexem als Key und dem Token als Value.

### 2.3.1 Implementierung

Die Symboltabelle ist ein Array mit verketteten Listen. Bei der Initialisierung wird das Array sowie die einzelnen verketteten Listen erstellt. Die Listen sind zu diesem Zeitpunkt komplett leer.

#### Auslesen

Zum Auslesen wird ein Lexem benötigt. Dieses dient mittels einer Hash-Funktion zur Identifizierung der Konkreten verketteten Liste im Array. Daraufhin wird innerhalb der verketteten Liste der Eintrag gesucht, der das gleiche Lexem besitzt.

#### Einfügen

Möchte man ein Lexem einfügen, dann wird zusätzlich noch ein TokenType benötigt. Es wird zuerst geprüft, ob dieses Lexem nicht bereits vorhanden ist und gegebenenfalls die dort hinterlegten Daten zurück gegeben. In diesem Fall wird der mitgegebene TokenType ignoriert. Ist ein Lexem noch nicht vorhanden, dann wird die Kombination aus Lexem und TokenType in der verketteten Liste gespeichert.

## 2.4 Scanner

Der Scanner ist für die Generierung der Tokens anhand einer Datei verantwortlich, von der nur der Pfad bekannt ist. Dazu muss nach der Initialisierung die Methode `nextToken()` aufgerufen werden.

### 2.4.1 Initialisierung

Der Scanner benötigt zur Initialisierung den Pfad zu der Quelldatei sowie eine Symboltabelle. Als erstes wird ein neuer Buffer mit dem Pfad zur Quelldatei erstellt sowie ein neuer Automat erstellt.

In die Symboltabelle werden dann die bekannten Schlüsselwörter als entsprechende Token eingetragen.

### 2.4.2 nextToken()

Diese Methode ist dafür verantwortlich das nächste Token aus der Datei zu generieren.

Dazu wird so lange aus dem Buffer der nächste Buchstabe ausgelesen, wie ein Zeichen ausgelesen wird, welcher zu keinem Token gehört. Diese Zeichen sind die Zeilenumbrüche sowie Whitespaces, wie Tabs oder Leerzeichen. Somit werden diese Zeichen vor einem neuen Token ignoriert.

Als nächstes wird gespeichert, in welcher Zeile und Spalte man sich gerade befindet. Dies ist für Debug Informationen wichtig.

Daraufhin wird innerhalb einer Schleife immer wieder dasselbe getan:

- Aus dem Buffer wird der nächste Buchstabe ausgelesen. Handelt es sich hierbei um ein `\0`, dann ist die Datei zu Ende.
- Im Automaten wird geprüft, in was für einem Zustand man sich befindet und das Ergebnis wird als Token zurück gegeben.
- Gibt der Buffer kein `\0`, dann wird dieser Buchstabe in den Automaten gegeben. Handelt es sich um einen validen Übergang wird der Automat gefragt, ob das Zeichen gespeichert werden soll. Falls ja, dann wird das Zeichen in einem char Array zwischengespeichert. Dies ist nötig um Lexeme zu speichern.
- Gibt der Automat keine 0 zurück, dann handelt es sich um einen invaliden Überang. In diesem Fall wird geprüft, ob der Wert positiv oder negativ ist: ist der Wert negativ, dann muss der Buffer die invertierte Anzahl Schritte zurück gehen. Falls der Wert positiv ist, dann muss der Buffer nur einen Schritt zurück gehen.
- Der Automat wird nach dem letzten validen Zustand gefragt.

- Die zwischengespeicherten Buchstaben werden mit einem `\0` terminiert und die bekannten Informationen nun zu einem Token zusammen gefasst.

# 3 Parser

Der Aufgabenteil *Parser* implementiert die Aufgaben der *syntaktischen und semantischen Analyse* unseres Compilers. Das bedeutet, dass der Parser aus den Tokens, die er vom Scanner bekommt einen Strukturbaum aufbauen muss und auf diese Weise überprüft, ob die Ausdrücke in der Eingabedatei entsprechend der Grammatik gültig sind (syntaktische Analyse). Im nächsten Schritt wird überprüft, ob die Ausdrücke auch grundlegend logisch sind (z. B. Identifier definiert sind, Rechnungen nur mit gleichen Datentypen durchgeführt werden, etc.). Dieser Schritt wird semantische Analyse genannt.

Um diese Aufgaben erfüllen zu können, besteht der Parser aus zwei Teilen:

- dem Parse-Tree, welcher die Regeln unserer Grammatik in Programmstrukturen (Interfaces und Klassen) darstellt, und
- dem Parser selbst, welcher die Tokens entgegennimmt und prüft, welche Tokens wann gültig sind und welche Knoten am Baum angehängt werden. Der Parser baut auch den Strukturbaum selbst auf.

## 3.1 Parse-Tree

In unserer Implementierung ist die Struktur des Baumes sehr umfangreich, da wir die zweite in den Folien genannte Alternative implementiert haben, welche daraus besteht für jede einzelne Regel der Grammatik eine eigene Klasse zu erstellen und für jedes Nichtterminal ein Interface, welches die einzelnen Regeln implementieren müssen. Diese Variante hat den großen Vorteil, dass man den Strukturbaum nicht invalide aufbauen kann, da nur vorher festgelegte Klassen, entsprechend der Grammatik, angehängt werden können. So können z. B. an einen `STATEMENTS`-Knoten nur ein `STATEMENT` sowie ein `STATEMENTS`-Knoten angehängt werden; andere Konstellationen sind nicht möglich.

Alle Knoten erben außerdem vom `IParseTree`-Interface, welches die grundlegenden Methoden, die jede Klasse implementieren muss vorschreibt:

```
1  class IParseTree {
2  private:
3      int line, col;
4  public:
5      virtual void typeCheck(Symboltable *symtable)=0;
```

```
6       virtual void makeCode(std::ofstream *code)=0;
7       Type type;
8       void setPos(int line, int col){this->line = line; this->col = col;};
9       int getLine(){return this->line;};
10      int getCol(){return this->col;};
11  };
```

Listing 3.1: Parser: IParseTree-Interface

Die Klassenstruktur einer einzelnen Regeln sieht, hier am Beispiel der Regel Decl_I (`int ARRAY identifier`, wobei `int` ein Terminal (Schlüsselwort) ist), wie folgt aus:

```
1   class Decl_I : public IDecl { // Decl_I implmentiert das Interface IDecl
2   public:
3       Decl_I(const char *identifier); // Im Konstruktor wird der Identifier
            uebergeben
4       virtual ~Decl_I();
5       void typeCheck(Symboltable *symtable);
6       void makeCode(std::ofstream *code);
7       void addNodes(IArray *array); // Es darf nur ein Knoten vom Typ IArray
            angehaengt werden
8   private:
9       IArray *array;
10      const char *identifier;
11  };
```

Listing 3.2: Parser: Klassenstruktur von Decl_I

### 3.1.1 Type-Check

Der Type-Check dient der semantischen Analyse des aufgebauten Strukturbaumes. Er wird am Ursprungsknoten gestartet und läuft dann rekursiv jeden einzelnen Knoten des Baumes ab, bis alle Knoten geprüft sind. Falls bei der Überprüfung ein Fehler gefunden wird (z. B. die Verwendung eines unbekannten Identifiers), wird eine Fehlermeldung ausgegeben, der Vorgang aber nicht komplett abgebrochen, da ein Fehler hier zwar ein logischer Fehler ist, der Erzeugung des Codes aber nicht im Wege steht (ein Fehler würde erst bei der Ausführung des Codes auftreten). Ein Abbruch des Programms ist aber möglich und auch theoretisch vorgesehen, es muss nur eine Zeile einkommentiert werden.

Für den Type-Check musste auch die Symboltabelle angepasst werden. Die Möglichkeit zu einem Identifier den Typ abzuspeichern, welchen dieser Identifier beschreibt (z. B. Integer oder Array) war vorher nicht vorgesehen und wurde eingefügt. Dazu wurden die Methoden `storeIdentifierType()` und `getIdentifierType()` implementiert, welche einfach noch ein weiteres Feld zum Identifier speichern.

### 3.1.2 Code-Erzeugung

Die Aufgabe der Code-Erzeugung wird in den einzelnen Klassen durch die Methode `makeCode()` implementiert. Auch hier wird der Baum rekursiv durchlaufen, jeder Knoten muss also auch die Code-Erzeugung bei seinen Kind-Knoten aufrufen. Als Beispiel hier die Code-Erzeugung bei `Statement_I`, also der Regel `identifier INDEX := EXP`, welche die Zuweisung einer Variablen darstellt:

```
1 void Statement_I::makeCode(std::ofstream *code) {
2     this->exp->makeCode(code); // Rufe Code-Erzeugung beim Child-Node exp
          auf
3     *code << "LA " << "$" << this->identifier << ENDL; // Fuege "Lade-
          Adresse" des Identifiers ein
4     this->index->makeCode(code); // Falls ein Array-Zugriff erfolgen soll,
           fuege diesen Zugriff ein
5     *code << "STR" << ENDL; // Fuege den "Store"-Befehl ein
6 }
```

Listing 3.3: Parser: Code-Erzeugung des Zuweisung-Statements

#### Besonderheiten bei '>' und '<:>'

Da es in der Assembler-Sprache die Konstrukte „größer als“ und „ungleich“ nicht gibt, diese aber in unserer Grammatik vorgesehen sind müssen diese in valide, gleichbedeutende Ausdrücke umgewandelt werden.

So wird, nach folgender Formel, im Falle von „größer als“ einfach ein „kleiner als“ verwendet und die beiden Operanden getauscht:

$a > b \iff b < a$

Im Falle von „ungleich“ ist es ähnlich einfach: hier wird das „ungleich“ durch ein „gleich“ ersetzt und das Ergebnis am Ende negiert:

$a \neq b \iff \neg(a = b)$

## 3.2 Parser

Der Parser selbst hat neben der `parse()`-Methode, welche den Parse-Vorgang startet und einer Methode, die Labels generiert, für jedes Nichtterminal eine Methode, welche den Baum unterhalb dieses Nichtterminals rekursiv aufbaut. Der Aufruf muss hier also auch nur einmal, an der Wurzel, erfolgen und der komplette Strukturbaum entsteht automatisch.

Die Methode `parse()` ist wie folgt implementiert:

```
1 void Parser::parse(const char *path) {
2   // Ausgabedatei oeffnen
3     std::ofstream code(path, std::ios::out | std::ios::trunc);
```

```
4
5       // Erstes Token anfordern
6       nextToken();
7
8       printf("parsing...\n");
9       // Baum aufbauen, Wurzel ist immer ein PROG
10      IParseTree *parseTree = PROG();
11
12      printf("type checking...\n");
13      // Type-Check an der Wurzel starten
14      parseTree->typeCheck(this->symtable);
15
16      printf("generating code...\n");
17      // Code-Erzeugung an der Wurzel starten
18      parseTree->makeCode(&code);
19      printf("stop.\n");
20
21      code.close();
22  }
```

Listing 3.4: Parser: Methode parse()

Als Beispiel für das Aufbauen des Baumes hier die Methode `DECL()`:

```
1   IDecl* Parser::DECL() {
2       IDecl *tree;
3       if (currentToken->getTokenType() == tokenInt) {
4           nextToken();
5           IArray *array = ARRAY();
6           if (currentToken->getTokenType() == tokenIdentifier) {
7               tree = new Decl_I(currentToken->getInformation()->getLexem());
8               tree->setPos();
9               static_cast<Decl_I*>(tree)->addNodes(array);
10              nextToken();
11          } else {
12              error("expected identifier", currentToken);
13          }
14      } else {
15          error("unidentified token", currentToken);
16      }
17      return tree;
18  }
```

Listing 3.5: Parser: Methode DECL()

Da die Methode `addNodes()` bei jeder Klasse unterschiedliche Parameter erfordert, kann diese weder bei den Nichtterminal-Interfaces, noch beim Parse-Tree-Interface vorgeschrieben sein. Damit der Aufruf erfolgen kann, muss das Objekt `tree`, vom Typ `IDecl` für diesen

Aufruf in ein Objekt vom Typ `Decl_I` gecastet werden. Dies ist problemlos möglich, da `tree` tatsächlich ein Objekt vom Typ `Decl_I` ist und erfolgt überall dort, wo die Methode `addNodes` aufgerufen wird.

Die `error`-Methode gibt eine Fehlermeldung, mit Angabe von Zeile und Spalte aus und beendet das Programm.

Für die Konstrukte `if` und `while` der Grammatik, werden Labels benötigt, welche im Programm eindeutig sein müssen. Für die Erzeugung der Labels implementiert der Parser eine Methode, welche bei jedem Aufruf einen Integer hochzählt und zurückgibt. In der Code-Erzeugung, wo die Labels benötigt werden, wird diesem Integer ein Rautensymbol (kennzeichnet ein Label), sowie ein „L“ zuvorgestellt, denn ein Label muss genau so aufgebaut sein, wie ein Identifier, darf also nicht mit einer Ziffer beginnen. Auf diese Variante ist die Erzeugung der Labels relativ einfach und die Eindeutigkeit garantiert.

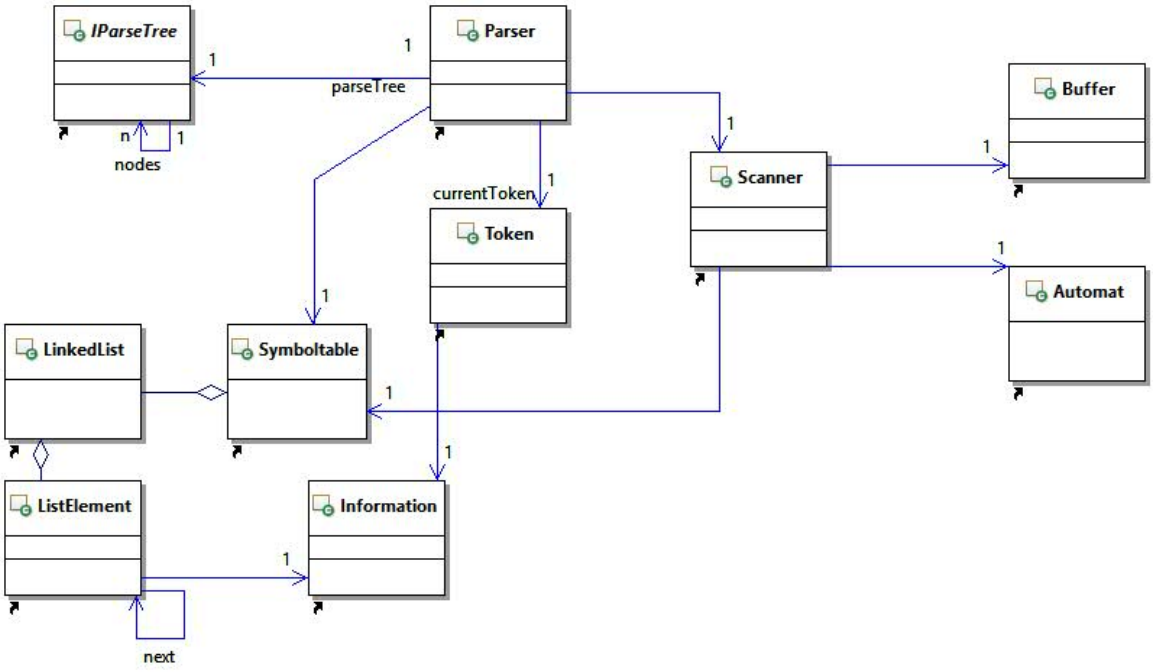


Abbildung 3.1: Klassendiagramm des Compilers