Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Systemnahes Programmieren

Dokumentation

Gruppe 35

Kishor Rana, 43281

Jonas Weber, 43399

Tim Bierenbreier, 43235

Lutz Bader, 42887

Wintersemester 2016/2017

20. Januar 2017

Inhaltsverzeichnis

Teil 1

Lexikalische Analyse

Ziel des Labors ist es, dass sich die Studierenden mit der Funktionsweise und der Implementierung eines Compilers vertraut machen. Diese Funktionsweise lässt sich grob wie in der Abbildung 1.1 darstellen.

Außerdem werden durch die Aufgabenstellung die Kenntnisse vieler Disziplinen vereint, da ein Compiler vielerlei Kenntnisse verlangt. Die Aufgabe des Labors teilt sich grob in zwei Teile. Zum einen soll im ersten Teil soll ein Scanner implementiert werden, der wiederum grundlegende Funktionen des Compilers beinhaltet. Dieses ersten Kapitels der Dokumentation gibt einen Überblick über die grobe Implementierung des Scanners.

1.1 Buffer

Der Buffer dient als Zwischenspeicher für die eingelesene Datei. Der Scanner bedient sich am Buffer, indem er aus ihm die Chars der eingelesenen Datei liest, wobei es die Aufgabe des Buffers ist, sich zur richtigen Zeit neu zu befüllen. Die Klasse Buffer verwaltet zwei Objekte buffer1 und buffer2 vom Typ `BufferHalf`, sowie einen Pointer `bufferAktuell`, der auf die aktuell benutzte Bufferhälfte zeigt. Eine Bufferhälfte besitzt ein Char-Array für die Speicherung der Zeichen, außerdem einen Counter, der die aktuelle Stelle im Buffer mitzählt, eine Variable `bytesread`, die die mit der Systemfunktion `read()` eingelesenen Bytes zählt und ein Flag `finished`, welches binär speichert, ob die Bufferhälfte fertig gelesen wurde. Im Konstruktor `BufferHalf()` wird die Systemfunktion `posix_memalign()` aufgerufen, um 1024 Byte für das Char-Array zu reservieren. Ein char entspricht einem Byte, wodurch das Char-Array nun Indizes von 0 bis 1023 besitzt. Im Konstruktor

Abbildung 1.1: Schematische Darstellung eines Compilers

der Klasse Buffer werden zuerst zwei Objekte der Klasse `BufferHalf` erstellt. Anschließend wird die Systemfunktion `open()` aufgerufen, der den Pfad der Datei übergeben wird, die zu öffnen ist. Es wird ein Integer file descriptor zurückgegeben, der später der Systemfunktion `read()` übergeben werden muss, um die referenzierte Datei in den Buffer einzulesen. Anschließend wird im Konstruktor noch dafür gesorgt, dass der Pointer `bufferAktuell` auf die erste Bufferhälfte buffer1 zeigt und der Spaltenzähler column und der Zeilenzähler line des Buffers auf 1 stehen. Um den Buffer das erste Mal aufzufüllen, wird die Methode `fillBuffer()` aufgerufen. Diese Methode ruft die Systemfunktion `read()` auf, welcher der oben genannte file descriptor übergeben wird, um die ersten 1024 Zeichen (Bytes) in das Char-Array der aktuellen Bufferhälfte einzulesen. Die `read()`-Funktion liest immer ab der zuletzt gelesenen Stelle weiter. Der Return-Wert der `read()`-Funktion ist die Anzahl der tatsächlich gelesenen Bytes (sind weniger als die geforderte Anzahl, falls die Datei vorher zu Ende ist) und wird in die Variable `bytesread` der aktuellen Bufferhälfte gespeichert. Die Methode `getChar()` gibt das aktuelle Zeichen im Buffer zurück und zählt den Counter hoch. Wenn das Ende der aktuellen Bufferhälfte erreicht ist, wird die Bufferhälfte ausgewechselt und die nächste aufgefüllt. Wenn das Ende der eingelesenen Datei erreicht ist, wird `#` zurückgegeben. In `getChar()` wird des Weiteren jedes Mal die `column`-Variable der aktuellen Bufferhälfte hochgezählt, sowie die `line`-Variable, falls das aktuelle Zeichen ein Zeilenumbruch ist. Wenn die aktuelle Bufferhälfte vollständig gelesen wurde, wird ihr `finished`-Flag gesetzt, das letzte Zeichen der Bufferhälfte wird zwischengespeichert, die Bufferhälften werden gewechselt, ihr Counter und ihr `finished`-Flag wird 0 gesetzt und die nun aktuelle Bufferhälfte wird mit der `read()`-Funktion neu befüllt. Wenn das zwischengespeicherte letzte Zeichen der vorherigen Bufferhälfte der Nullterminator aus der gelesenen Datei ist (`\n`) und das erste Zeichen der nächsten Bufferhälfte `\0` ist, also die Datei genau in die vorherige Bufferhälfte gepasst hat, wird `#` zurückgegeben. Wenn dies nicht der Fall ist, wird als nächstes das zwischengespeicherte Zeichen ausgegeben. Ansonsten, wenn man sich in keinem Sonderfall befindet, wird ganz normal das aktuelle Zeichen der aktuellen Bufferhälfte ausgegeben.

1.2 Automat

Durch eine Zustandsübergangstabelle, die die gesamte Logik des Automaten beinhaltet, ist der Code überschaubar. Er besteht insgesamt aus drei verschiedenen Teilen. Abbildung 1.2 zeigt die beschriebene Zustandsübergangstabelle im Detail.

1.2.1 Die Zustandsübergangstabelle

Der neue Zustand berechnet sich aus der Kombination aus dem aktuellen Zustand und der Eingabe. Die Eingabe befindet sich in der Zustandsübergangstabelle in der ersten Spalte (Abbildung 1.2a). Der aktuelle Zustand ist in der ersten Zeile der Tabelle zu finden. Die verschiedenen Kombinatinonsmöglichkeiten können dementsprechend zusammengesetzt werden.

Anfangs ist der Automat im Zustand 0. Kommt ein Zeichen, welches getestet werden soll, zum Beispiel ein *, würde der Automat im Zustand 9 landen. Da ein * ein fertiges Token ist, ist in der Spalte 9 des Automaten, egal bei welchem Input, der Zustand 109 eingetragen. Alle „Hunderter"-Zustände sind in der Tabelle Endzustände, bei dem der Automat weiß, dass er ein bestimmtes Token ausgeben muss.

1.2.2 Eingabeumwandlung

Hier werden die übergebenen Characters in die richtigen Integers umgewandelt, welche dann von der Zustandsübergangstabelle verarbeitet werden können. Der zugehörige Quellcode ist in Listing 1.1 in gekürzter Fassung zu finden.

```
1  int Automat::transformChar(char c) {
2    int eingabe;
3
4    if (c == '0' || // ...
5    c == '9') eingabe = 0;
6    else if (c == 'A' || // ...
7    c == 'z') eingabe = 1;
8
9    else if (c == '+') eingabe = 2;
10   else if (c == '-') eingabe = 3;
11   //...
12   else if (c == ']') eingabe = 17;
13   else if (c == ' ') eingabe = 18;
14   else eingabe = 19;
15
16   return eingabe;
17 }
```

Listing 1.1: Übergangstabelle

1.2.3 Der Check und die Ausgabe

Hier werden zuerst der Eingabewert und der aktuelle Zustand in die Zustandsübergangstabelle gegeben. Der neue Zustand wird anschließend zurückgegeben. Jetzt wird anschließend gecheckt, ob es sich um einen Endzu-

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	Digt 101	{	1	1	2	103	104	105	106	107	108	109	110	111	112	113	114
3	Lexem 102	{	2	101	2	103	104	105	106	107	108	109	110	111	112	113	114
4	_+103	{	3	101	102	103	104	105	106	107	108	109	110	111	112	113	114
5	_-104	{	4	101	102	103	104	105	106	107	108	109	110	111	112	113	114
6	/ 105	{	5	101	102	103	104	105	106	107	108	109	110	111	112	113	114
7	_* 106	{	6	101	102	103	104	24	106	107	108	109	110	111	112	113	114
8	< 107	{	7	101	102	103	104	105	106	107	108	109	110	111	112	113	114
9	> 108	{	8	101	102	103	104	105	106	107	108	109	110	111	112	113	114
10	_= 109	{	9	101	102	103	104	105	106	107	108	109	110	111	112	113	114
11	& 110	{	10	101	102	103	104	105	106	107	108	109	110	111	112	113	114
12	;111	{	11	101	102	103	104	105	106	107	108	109	110	111	112	113	114
13	(112	{	12	101	102	103	104	105	106	107	108	109	110	111	112	113	114
14	)113	{	13	101	102	103	104	105	106	107	108	109	110	111	112	113	114
15	{114	{	14	101	102	103	104	105	106	107	108	109	110	111	112	113	114
16	}115	{	15	101	102	103	104	105	106	107	108	109	110	111	112	113	114
17	[116	{	16	101	102	103	104	105	106	107	108	109	110	111	112	113	114
18	]117	{	17	101	102	103	104	105	106	107	108	109	110	111	112	113	114
19	leer	{	0	101	102	103	104	105	106	107	108	109	110	111	112	113	114
20	:	{	18	101	102	103	104	105	106	20	108	109	110	111	112	113	114
21	sonst	{	27	101	102	103	104	105	106	107	108	109	110	111	112	113	114
22	! 121	{	23	101	102	103	104	105	106	107	108	109	110	111	112	113	114
23	:= 118																
24	<> 119																
25	Error 120																
26																	

(a) Linke Hälfte der Übergangstabelle

R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE
15	16	17	18	19	20	21	22	23	24	25	26	27	
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	26	122	120	},
115	116	117	120	118	21	120	119	121	25	25	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	22	120	119	121	24	24	122	120	},
115	116	117	19	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	},
115	116	117	120	118	21	120	119	121	24	24	122	120	}

(b) Rechte Hälfte der Übergangstabelle

Abbildung 1.2: Übergangstabelle

stand handelt oder nicht. Ein Endzustand wäre, wie zuvor beschrieben, ein Zustand größer als Hundert. Falls es ein Endzustand ist, wird der jeweilige Typ, also `TType`, des Tokens zurückgegeben und der aktuelle Zustand wieder auf 0 gesetzt. Falls es sich nicht um einen Endzustand handelt, wird der `TType null` zurückgegeben. Die einzige Ausnahme ist Zustand 21. Dieser ist kein Endzustand, hat aber eine Ausgabe. Dies wird benötigt, falls ein „:“ auf ein „<“ folgt, dann aber das Zeichen „>“ ausbleibt. Es muss auf das dritte Zeichen gewartet werden, damit eine Aussage zum `TType` getätigt werden kann. Falls dieses ausbleibt, muss zuerst ein `TType` und dann ein `ErrorTType` ausgegeben werden, bevor weiter verfahren werden kann. Ein Doppelpunkt darf somit nicht alleine stehen. Was muss ich nun weiter beachten, wenn ich den Automat benutzen will? Da der Automat nicht weiß, welches Zeichen als nächstes kommt, gibt er erst einen richtigen `TType` zurück, wenn ein Zeichen folgt, das nicht zu diesem Typ gehört. Dieses Zeichen muss dann nochmal in den Automaten gegeben werden. Falls man also einen `TType` ungleich Null zurück bekommt, muss man das Zeichen zuvor noch einmal testen lassen. Falls Null zurück gegeben wird, kann der Automat mit dem nächsten Zeichen gespeist werden. Dies ist zwar nicht sehr komfortabel, doch der Scanner weiß damit umzugehen. Deshalb wäre hier eine Optimierung doppelt so viel Arbeit und wurde ausgespart.

```
1  TType Automat::checkChar(char c) {
2    Automat::aktuellerZustand =
      zustandUebergangTabelle[Automat::transformChar(c)
     ][aktuellerZustand];
3    switch (Automat::aktuellerZustand) {
4      case (eDigit): aktuellerZustand = anfang;
5        return integer;
6      case (eLexem): aktuellerZustand = anfang;
7        return lexem;
8      case (ePlus):  aktuellerZustand = anfang;
9        return plusToken;
10     // ...
11     case (zUngleichFailed):
12       return gleich;
13     case (eError): aktuellerZustand = anfang;
14       return error;
15     case (eKommentar): aktuellerZustand = anfang;
16       return kommentar;
17     default:
18       return null;
19   }
20 }
```

Listing 1.2: Check und Ausgabe

1.3 Symboltabelle

Die Symboltabelle ist als `HashMap` umgesetzt. Die wichtigsten Tokens werden mit der der Methode `initSymbols()` in die Symboltabelle eingetragen. Die Symboltabelle dient dazu, die Tokens zu verwalten, zu denen Informationen wie das zugehörige Lexem und der zugehörige Wert gespeichert werden sollen. Insbesondere trifft dies auf die Bezeichner zu. Andererseits können auch einige in der Sprache vorhandenen Schlüsselwörter, wie beispielsweise `read` oder `print` in der Symboltabelle statt im Automaten verwaltet werden, was zu einer einfacheren Wartung des Automaten führt. Bei der Umsetzung der Symboltabelle war ein wichtiger Eckpfeiler, dass sich Informationen schnell abrufen und einspeichern lassen. Insofern haben wir uns dazu entschieden die Symboltabelle als `HashMap` zu implementieren. Diese greift intern wiederum auf verkettete Listen zurück.

1.3.1 HashMap

Die `HashMap` für die Symboltabelle besteht aus `LinkedLists`, die Objekte vom Typ Token enthalten. Die Methode `getTokenType(String lexem)` der `HashMap` nimmt als Argument ein Lexem (String) als Key entgegen, durchsucht die `HashMap` nach einem Token, bei dem dieser String übereinstimmt und gibt dann den `TType` des Tokens zurück. `TType` ist als Enum verwirklicht, das alle möglichen `TokenTypes` definiert. Durch einen Aufruf der Methode `insertToken(Token* t)` setzt ein Objekt vom Typ Token in die `HashMap` ein. Für die Nutzung der `HashMap` wurden in die beteiligten Klassen Token und String passende `equals()` und `hashCode()` Methoden implementiert.

1.3.2 LinkedList

Die `HashMap` der Symboltabelle greift auf eine simple LinkedList-Implementierung zurück. Es handelt sich in diesem Fall um eine Klasse, die Token-Objekte speichern kann, die in Token-Node-Objekte verpackt sind. Die Token-Node-Objekte speichern jeweils eine Referenz auf das verpackte Token-Objekt und beinhalten zusätzlich Informationen zu dem vorherigen und folgenden Token-Node-Objekt in der Liste. Somit wird die Funktionalität einer doppelt verketteten Liste umgesetzt, und das eigentliche Linked-List-Objekt speichert somit einfach nur auf das erste und letzte Token-Node-Element in der Liste.

1.4 Scanner

Der Scanner stellt sich als das Ziel der bis zu diesem Zeitpunkt entwickelten Unterkomponenten Buffer, Symboltabelle und Automat dar. Insofern

greift er auf diese zurück um aus einer gegebenen Datei Token für Token zu extrahieren und als Token-Objekt auszugeben.

Die Hauptmethode des Scanners heißt `nextToken()`. Die Methode ist prinzipiell so aufgebaut, dass für jedes Zeichen, dass der Scanner vom Buffer holt zwei Phasen durchlaufen werden.

1. Der Scanner cacht das aus dem Buffer entnommene Zeichen
2. Der Automat prüft das Zeichen
    - Falls noch nichts erkannt wurde, erweitert der Scanner seinen Lexem-String-Cache um das aufgenommene Zeichen und geht wieder in Phase 1
    - Wenn der Automat das Zeichen direkt erkannt hat, wird es vom Scanner abgegeben
    - Wenn ein Lexem erkannt wurde, das der Automat nicht direkt zuordnen kann, gibt der Scanner das Lexem an die Symboltabelle weiter
        - Ist ein Eintrag in der Symboltabelle vorhanden, wird das passende Token mit der für das Lexem zugehörigen `TokenInformation` vom Scanner rausgegeben.
        - Ansonsten erstellt der Scanner ein neues Token mit dem Lexem. Zudem wird für das Lexem eine `TokenInformation` erstellt und wenn zutreffend wird auch der Zahlenwert des Tokens gespeichert. Das Token mit `TokenInformation` wird in der Symboltabelle eingetragen und der Scanner gibt dieses an den Methodenaufrufer zurück

Sobald der Automat mittels des von uns verwendeten Stopzeichens `#` dem Scanner signalisiert, dass das Ende der einzulesenden Datei erreicht worden ist, erstellt der Scanner ein Stop-Token. Vom Automaten als fehlerhaft erkannte Tokens werden vom Scanner in einer Error-Message mit Zeile, Spalte und fehlerhaftem Zeichen ausgegeben. Kommentare werden vom Automaten erkannt und der Scanner tauscht diese lediglich gegen Whitespace aus.

Teil 2

Parser

Zu Beginn ist in Abbildung 1.1 der grobe Ablauf eines Compilers dargestellt worden. Die Abbildung 2.1 zeigt die einzelnen Schritte eines Compilers etwas mehr im Detail. Dabei wird auch die Aufgabe eines Parsers innerhalb eines Compilers deutlich. In der syntaktischen Analyse werden die Symbole des Quellprogramms zu einem Baum aufgebaut. In der semantischen Analyse wird das Programm auf semantische Fehler geprüft, speziell die Korrektheit der Operanden. Im letzten Schritt nach der Analyse wird die Zielsprache erzeugt. In diesem Fall erfolgt die Übersetzung in die vorgegebene, an Assemblercode angelehnte Sprache, die vom mitgelieferten Interpreter ausgeführt werden kann. Dieser Interpreter stellt darüber hinaus eine recht gute Hilfe zum Überprüfen der Ergebnisse dar.

In den folgenden Kapiteln soll nun die Implementierung des Parsers genauer erläutert werden.

2.1 Parsing

Im zweiten Teil der Aufgabe wird nun der Parser implementiert. Der Parser bildet somit das abschließende Glied in der Kompilierung, allerdings steht der Parser noch vor der Code-Generierung.

Im ersten Schritt, dem sogenannten „Parsing“, baut der Parser unter Zuhilfenahme des Scanners einen Baum aus einem gegebenen Quellcode auf. Durch Verwendung dieses Baums können darauffolgend Code-Analyse Prozesse angestoßen werden. Wir verwenden hierzu das Visitor-Pattern, das es uns erlaubt ein Visitor-Objekt durch den Baum zu schicken. Maßgeblich sind der `TypeCheckVisitor`, der die Evaluierung beziehungsweise die Typisierung durchführt und der `CodeGeneratorVisitor`, der beim Durchlaufen des Quellcodes Assembler-Anweisungen generiert. Ein oder mehrere Tokens finden sich nach dem Parsing als Knotenpunkte beziehungsweise Nodes in einer Baumhierarchie wieder. Abbildung 2.2 zeigt das zugehörige Klassendiagramm.

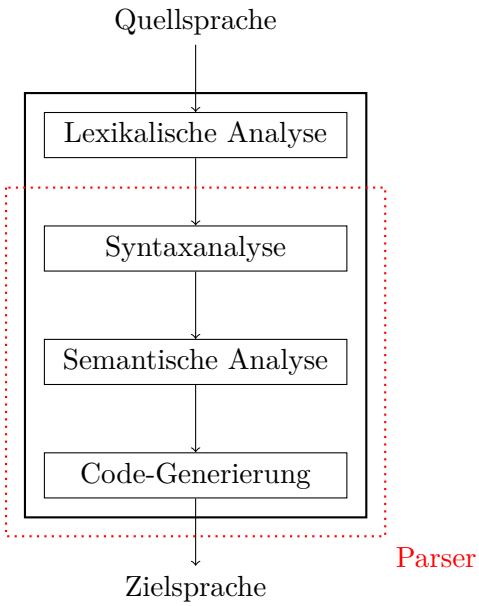

Abbildung 2.1: Schritte eines Compilers

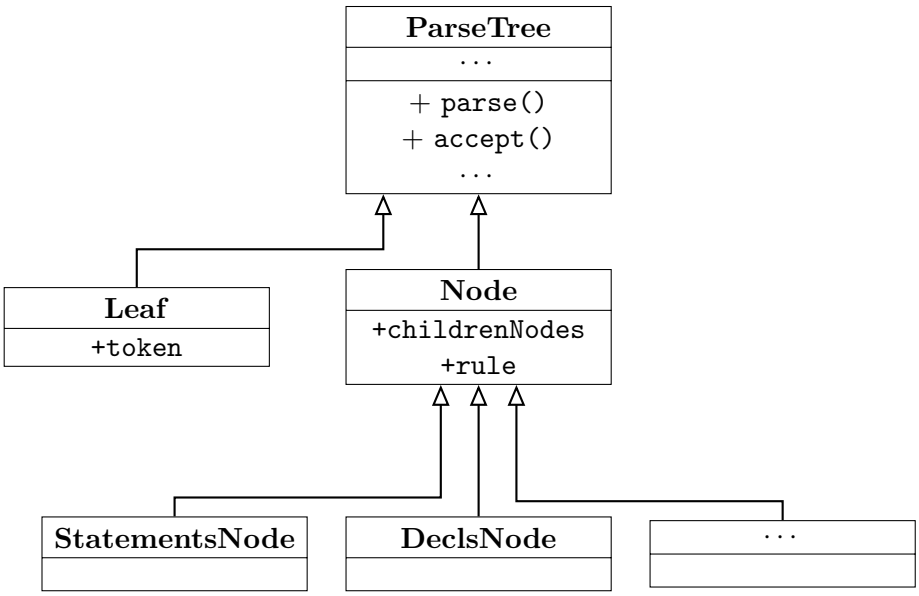

Abbildung 2.2: Vereinfachtes UML

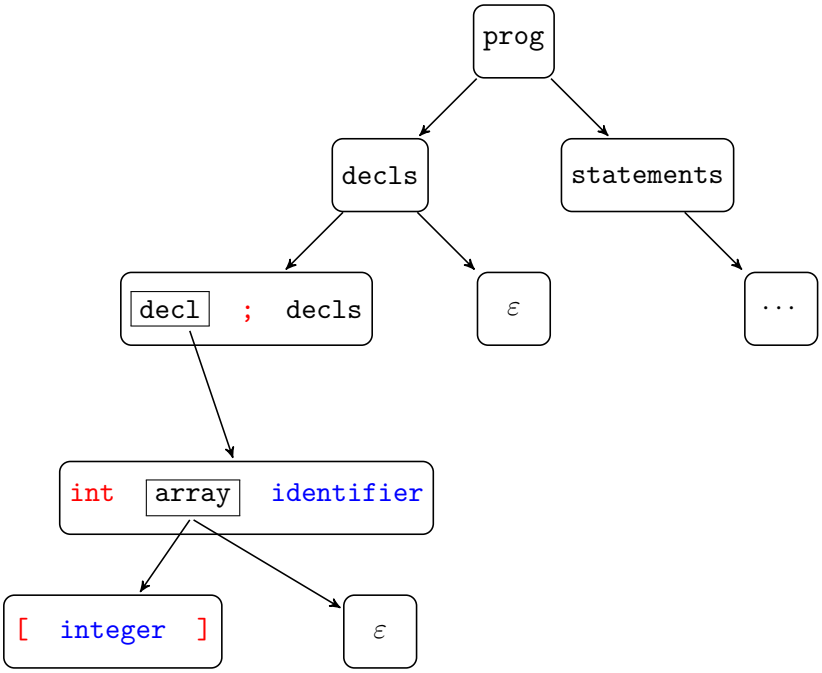

Abbildung 2.3: Baum nach dem Schema der Grammatik

Die Baumhierarchie ergibt sich daraus, dass Nodes wieder Kinder oder ChildrenNodes haben. Endpunkte oder Blätter im Baum sind vom Typ `Leaf`, haben keine Kinder (sie erben nicht von Nodes sondern dessen Basisklasse ParseTree) und ihnen lassen sich Tokens zuordnen. Der Prozess des Parsings gestaltet sich implizit aus den Regeln der Grammatik, der zu implementierenden Sprache. Ausgangspunkt für unseren Parser-Baum ist ein Knotenpunkt vom Typ `ProgramNode`, der wie sich aus der Grammatik zwingend ergibt ein `Decls-` und ein `Statement`-Node als Kinder haben muss. Abbildung 2.3 stellt den beschriebenen Sachverhalt grafisch dar.

Der Parser delegiert das Parsing mittels der `parse()` Funktion an die Knotenpunkte im Baum weiter. Wichtig ist, dass die `parse()`-Funktion jeweils einen Boolean zurückgibt, der Aussage darüber gibt, ob der Knoten-Punkt geparst werden konnte. Die Knotenpunkte definieren je nach Typ (Prog, Decls, Statement, Decl, etc.) welche Knotenpunkte sie zum Parsen auffordern und gegebenenfalls bei Erfolg signalisierendem Rückgabewert aus der `parse()`-Methode als ihre Kinder hinzufügen. Zudem wird beim Parsing in den Knoten im Feld `rule` eingespeichert nach welcher Regel ein neuer Teilbaum entsteht. Betrachten wir beispielsweise das „Statement“-Symbol aus der Grammtik, so kann dieses nach sechs Regeln einen Teilbaum aufspannen. Wird ein Endknotenpunkt beziehungsweise Leaf entsprechend des vom hinzufügenden Knoten erwarteten Tokentyps erfolgreich geparst, so gibt es dem Parser, durch Aufruf auf dem Parser der `nextToken()`-Funktion, dadurch

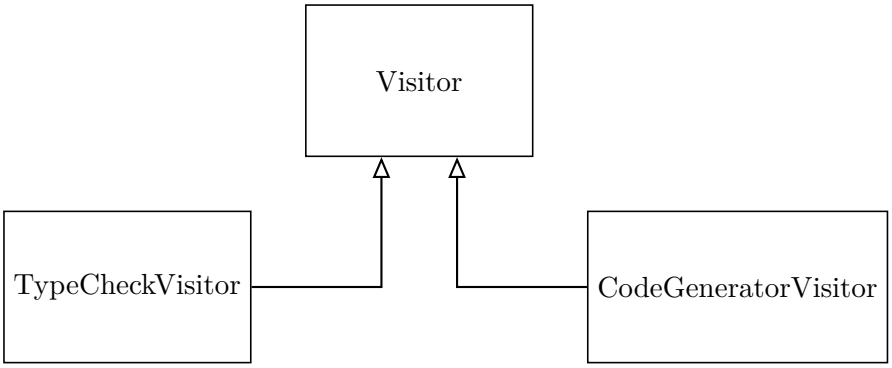

Abbildung 2.4: Visitor

Bescheid, dass ein neues Token vom Scanner angefordert wird. Während sich der Baum implizit durch seine Knoten aufbauen lässt, sorgt dies dafür, dass die Leafs Token für Token den Quellcode verarbeiten.

2.1.1 Visitor-Pattern

Mithilfe des Visitor-Patterns kann der im vorherigen Schritt gebaute Baum für die Semantische Analyse und die Codegenerierung durchlaufen werden. Beide Komponenten bedienen sich dem gleichen Prinzip, daher erben der TypeCheckVisitor und der CodeGenerationVisitor beide von derselben Superklasse (Abbildung 2.4). Durch dieses Patterns ist es möglich, die Operationen von den Knoten und Blättern selbst zu kapseln. Die Superklasse Visitor setzt voraus, dass ihre erbenden Klassen hierfür die Besuchsfunktion visit implementieren müssen. Selbstverständlich unterscheidet sich die Implementierung des jeweiligen abgeleiteten Visitors. Der TypeCheckVisitor implementiert in seinen visit-Funktione die Logik für die Semantische Analyse und der CodeGeneratorVisitor implementiert die Logik für die Generierung des Maschinencodes der Zielsprache.

2.2 Semantische Analyse

In der semantischen Analyse wird die Einhaltung der Regeln der Grammatik geprüft und im Fehlerfall ein entsprechender Fehler ausgegeben. Die semantische Analyse wird vom `TypeCheckVisitor` implementiert. Die Analyse des Baumes verläuft hierfür mithilfe des Visitor-Patterns nach dem Prinzip des rekursiven Abstiegs. Durch diese Typisierung der Knoten des Baumes und der Blätter ist zum Beispiel es möglich, Fehler wie die Deklarierung eines Arrays mit Größe null aufzudecken. Jeder Knoten und jedes Blatt speichert also genau einen Typ. Die Besonderheit dabei ist, dass dieser Typ für die Blätter in der Symboltabelle hinterlegt wird. Somit ist es beispielsweise mög-

lich festzustellen, ob eine Identifier bereits deklariert wurde.

2.3 Codegenerierung

Die eigentliche Codegenerierung wird im letzten Schritt im `CodeGenerator-Visitor` implementiert. Hier wird ausgehend vom Baum und den gesetzten Typen seiner Knoten der Assemblercode entsprechend erzeugt und mit dem `FileWriter` in eine Datei geschrieben.

Zur Verdeutlichung der Codegenerierung soll das Listing 2.1 angeführt werden. Der Quellcode ist gekürzt und soll nur die wesentliche Vorgehensweise aufzeigen.

```
1  void CodeGeneratorVisitor::visit(StatementNode &
       statementNode) {
2    if (statementNode.rule == ASSIGN_RULE) {
3      Leaf *identifierLeaf = (Leaf *) statementNode.
       getChildrenNodes()->get(0);
4      statementNode.getChildrenNodes()->get(3)->accept
       (*this);
5      fileWriter->write("\nLA $");
6      fileWriter->write(identifierLeaf->token->lexem);
7      statementNode.getChildrenNodes()->get(1)->accept
       (*this);
8      fileWriter->write("\nSTR");
9    } else if (statementNode.rule == WRITE_RULE) {
10     // ...
11   }
12   //...
13 }
```

Listing 2.1: Codegenerierung

Eine erwähnenswerte Besonderheit liegt bei der Erzeugung von `if` beziehungsweise `else` Statements und dem `while` Statement. Hier werden nämlich dynamische Sprungmarken erstellt, die nur einmal benutzt werden. Im Falle eines `if` beziehungsweise `else` Statements werden zwei unterschiedliche Sprungmarken benötigt. Dies wurde mithilfe einer Laufvariable realisiert, die für jede neue Sprungmarke erhöht wird und am Ende des `#label` hinzugefügt. Somit wird eine einzigartige Sprungmarke generiert.