Systemnahe Programmierung eines Compilers
II. Der Parser

Manuel Giesinger gima1019@hs-karlsruhe.de
Arthur Jagiella jaar1013@hs-karlsruhe.de

16. Juli 2017

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit ist im Rahmen der Laborübung „Systemnahes Programmieren“ im Studiengang Informatik (B.Sc.) an der Hochschule Karlsruhe im Sommersemester 2017 entstanden.

1.1 Aufgabenstellung gesamt

Im Kurs „Systemnahes Programmieren“ geht es inhaltlich um die Programmierung in C++ unter weitgehendem Verzicht auf die Nutzung von Standardbibliotheken. Die Implementierung von Puffern, verketteten Listen, Hashtabellen und Baumstrukturen soll per Hand erfolgen. Hierzu wird als Anwendung ein Compiler für eine fiktive Sprache implementiert.

1.2 Aufgabenstellung Parser – diese Arbeit

Die Aufgabe des Parsers besteht darin, die vom Scanner gelieferten Token für die Erstellung eines Parse-Tree zu verwenden. Ein Parse-Tree besteht hierbei aus einem Root Knoten, der weitere Knoten als Kindknoten besitzt. Mithilfe einer Parse-Tabelle und des nächsten Tokens wird beim Aufbau des Parse-Tree entschieden, welche weiteren Knoten angelegt werden sollen. Nachdem der Parse-Tree fertiggestellt ist, wird ein Typ-check durchgeführt, der überprüfen soll, ob die Variablentypen des Input Programms korrekt sind. Hierbei wird in diesem Compiler nur zwischen integer Variablen und integer array Variablen unterschieden. Nach einem erfolgreichen Typ-check wird zum Schluss Assembler Code für das Input Programm erzeugt und in eine Datei geschrieben. Die Output Datei kann, sofern die Input Datei korrekt war, vom bereitgestellten Interpreter gelesen und ausgeführt werden.
Im Folgenden werden die Klassen `Node`, `ParseTree` und `Parser` beschrieben. Diese werden in den weiteren Kapiteln 3, 4 und 6 genauer erläutert. Weiterhin wird kurz auf die beiden Aufzähltypen `RuleType` und `DataType` eingegangen. Die Entscheidung, welcher Code ausgeführt werden soll wird anhand des `TokenType` des nächsten Tokens und der `Parse-Tabelle` getroffen. Die Parse-Tabelle wird im Kapitel 5 genauer beschrieben.

2 Die Aufzählltypen RuleType und DataType

Für den Aufbau des Parse-Tree sowie den anschließenden Typ-check werden die beiden Aufzählltypen `RuleType` und `DataType` verwendet und in den einzelnen Knoten des Parse-Tree abgespeichert.

2.1 RuleType

Der Aufzählltyp `RuleType` besitzt die Werte `prog`, `decls`, `decl`, `array`, `statements`, `statement`, `exp`, `exp2`, `index`, `op_exp`, `op` und `terminal`. Dadurch kann mittels einer switch-case Anweisung entschieden werden, welcher Code in den Funktionen checkType(Node* node) und makeCode(Node* node) ausgeführt werden soll.

2.2 DataType

Der Aufzählltyp `DataType` besitzt die Werte `noType`, `intType`, `intArrayType`, `arrayType`, `opPlus`, `opMinus`, `opMult`, `opDiv`, `opLess`, `opGreater`, `opEqual`, `opUnEqual`, `opAnd` und `errorType`. Diese werden ebenfalls dafür verwendet, um in den Funktionen checkType(Node* node) und makeCode(Node* node) zu entscheiden, welcher Code ausgeführt werden soll.

3 Die Klasse Node

3.1 Aufgabe der Node

Eine Node repräsentiert einen Knoten des Parse-Tree, weshalb für jedes terminal als auch jedes nicht-terminal ein Knoten angelegt wird. Ein Knoten enthält neben den oben genannten Aufzählltypen `RuleType` und `DataType` auch den Aufzählltypen `TokenType` aus dem Scanner Teil, sowie einen Zeiger zum Elternknoten, einen Zeiger auf seine Kindknoten als auch einen Zeiger auf seine Geschwisterknoten.
Die terminale `identifier` und `integer` haben ihre eigene spezielle `Node` Klasse, die beide jewils von der Klasse `Node` erben. Die Klasse für `identifier` heißt `NodeId` und besitzt zusätzlich einen Zeiger auf den Informationscontainer des identifiers. Die Klasse für `integer` Werte heißt `NodeInt` und besitz an Stelle des Informationscontainers einen integer Wert.

3.2 Funktionen der Node

Sowohl die Klasse `Node`, als auch die abgeleiteten Klassen `NodeId` und `NodeInt` verfügen über alle relevanten `getter` und `setter` Methoden. Weiterhin können neue Kindknoten

und Geschwisterknoten über die Funktionen addChild(Node* child) und addSibling(Node* sibling) hinzugefügt werden. Zusätzlich gibt es die Funktion removeNode(Node* node), mit der es möglich ist einen Knoten aus dem Parse-Tree zu entfernen. Diese Funktion wird in unserem Compiler Projekt dazu verwendet diejenigen nicht-terminalen Knoten zu entfernen, die einen ϵ-Übergang gemäß der gegebenen Grammatik zulassen (z.B. DECLS) und einen solchen auch haben.

4 Der Parse-Tree

4.1 Aufgabe des Parse-Tree

Beim Parse-Tree handelt es sich um eine selbst erstellte Baumstruktur, die aus Knoten besteht. Der Parse-Tree verfügt über einen Root Knoten, bei dem es sich immer um einen Knoten mit dem `RuleType` prog handelt. Dieser Parse-Tree wächst durch das hinzufügen von Kindknoten und Geschwisterknoten. Nachdem der Parse-Tree für das Input Programm fertiggestellt ist, wird dieser in den Methoden `checkType(Node* node)` und `makeCode(Node* node)` rekursiv durchlaufen.

4.2 Funktionen des Parse-Tree

Der Parse-Tree verfügt in unserer Version nur über einen `getter` für den Root Knoten, sowie einige Funktionen, die lediglich zu debugging Zwecken dienen. Die Funktionen `checkType(Node* node)` und `makeCode(Node* node)` wurden in unserem Compiler Projekt in der Klasse `Parser` implementiert.

5 Die Parse-Tabelle

Die Parse-Tabelle gibt Aufschluss über die weitere Vorgehensweise bei der Erzeugung des Parse-Tree in Abhängigkeit des `TokenType` des nächsten Tokens und des aktuellen Knotens. Die Parse-Tabelle wurde durch die Anwendung der First_1 und Follow_1 Regeln, die im begleitenden Foliensatz erklärt wurden erstellt.

5.1 $First_1$

Diese Tabelle ist durch die Anwendung der Regeln zur Konstruktion von $First_1$ entstanden.

$First_1()$	Tokens
$First_1$(PROG)	int, identifier, write, read, if, while, {
$First_1$(DECLS)	int, ϵ
$First_1$(DECL)	int
$First_1$(ARRAY)	[, ϵ
$First_1$(STATEMENTS)	identifier, write, read, if, while, {, ϵ
$First_1$(STATEMENT)	identifier, write, read, if, while, {
$First_1$(EXP)	identifier, integer, (, -, !
$First_1$(EXP2)	identifier, integer, (, -, !
$First_1$(INDEX)	[, ϵ
$First_1$(OP_EXP)	+, -, *, :, <, >, =, =:=, &&, ϵ
$First_1$(OP)	+, -, *, :, <, >, =, =:=, &&

5.2 $Follow_1$

Diese Tabelle ist durch die Anwendung der Regeln zur Konstruktion von $Follow_1$ entstanden.

$Follow_1()$	Tokens
$Follow_1$(PROG)	ϵ
$Follow_1$(DECLS)	identifier, write, read, if, while, {, ϵ
$Follow_1$(DECL)	;
$Follow_1$(ARRAY)	identifier
$Follow_1$(STATEMENTS)	}, ϵ
$Follow_1$(STATEMENT)	else, ;
$Follow_1$(EXP)	else, ;,),]
$Follow_1$(EXP2)	else, ;,),], +, -, *, :, <, >,=, =:=, &&
$Follow_1$(INDEX)	else, ;,),], +, -, *, :, <, >, =, :=, =:=, &&
$Follow_1$(OP_EXP)	else, ;,),]
$Follow_1$(OP)	identifier, integer, -, !, (

5.3 Übergangstabellen

In diesem Abschnitt sind die Übergangstabellen für die einzenlen Regeln der Grammatik. Hierbei sind die daraus resultierenden abgeleiteten Regeln sowie die dafür zulässigen Tokens gegenübergestellt. Da eine große gemeinsame Tabelle zu viel Platz benötigt um anständig dargestellt zu werden, wurde für jede Regel eine eigene Tabelle angelegt.

5.3.1 PROG

Übergang	Tokens
DECLS STATEMENTS	identifier, int, write, read, while, if, {

5.3.2 DECLS

Übergang	Tokens
DECL; DECLS	int
ϵ	identifier, write, read, while, if, {

5.3.3 DECL

Übergang	Tokens
int ARRAY identifier	int
ϵ	;

5.3.4 ARRAY

Übergang	Tokens
[integer]	[
ϵ	identifier

5.3.5 STATEMENTS

Übergang	Tokens
STATEMENT; STATEMENTS	identifier, write, read, while, if, {
ϵ	}

5.3.6 STATEMENT

Übergang	Tokens
identifier INDEX := EXP	identifier
write(EXP)	write
read(identifier INDEX)	read
while(EXP) STATEMENT	while
if(EXP) STATEMENT else STATEMENT	if
{STATEMENTS}	{
ϵ	else, ;

5.3.7 EXP

Übergang	Tokens
EXP2 OP_EXP	identifier, integer, (, -, !
ϵ	else,],), ;

5.3.8 EXP2

Übergang	Tokens
identifier INDEX	identifier
integer	integer
(EXP)	(
-EXP2	-
!EXP2	!
ϵ	else,],), +, *, :, <, >, =, :=, =:=, &&, ;

5.3.9 INDEX

Übergang	Tokens
[EXP]	[
ϵ	else,],), +, -, *, :, <, >, =, :=, =:=, &&, ;

5.3.10 OP_EXP

Übergang	Tokens
OP EXP	+, -, *, :, <, >, =, =:=, &&
ϵ	else,],), ;

5.3.11 OP

Übergang	Tokens
+	+
-	-
*	*
:	:
<	<
>	>
=	=
=:=	=:=
&&	&&
ϵ	identifier, integer, (, !

6 Der Parser

6.1 Aufgabe des Parsers

Der `Parser` ist die zentrale Einheit und führt die oben genannten Klassen zu einem Modul zusammen. Die Aufgabe des Parsers besteht darin, den `Parse-Tree` durch die Erzeugung neuer Knoten aufzubauen und anhand der Übergänge der `Parse-Tabelle` den gewünschten Code auszuführen. Weiterhin wird im Parser der Typ-check und die Erzeugung des Assembler Codes durchgeführt. Der Parser besitzt einen Zeiger auf den `Scanner` und initialisiert diesen mit dem übergebenen Input Programm. Er besitzt außerdem Variablen für den aktuellen `RuleType`, einen Zeiger auf die aktuelle `Node` sowie einen Zeiger auf den `Parse-Tree`.

6.2 Funktionen des Parsers

Der Parser verfügt über eine Funktion für jede Regel der gegebenen Grammatik (prog, decls, decl, usw.). Diese werden gemäß der Übergangstabellen aufgerufen und enthalten den Code zur Erzeugung der entsprechenden Knoten, welche dann zum `Parse-Tree` hinzugefügt werden. Hierfür gibt es Funktionen zum hinzufügen eines bestimmten Knotentyps (id, integer, terminal und nicht-terminal) zum `Parse-Tree` als auch zum Entfernen eines Knotens vom `Parse-Tree`. Der Parser verfügt außerdem über die beiden Funktionen `checkType(Node* node)` und `makeCode(Node* node)`, in denen der Typ-check und die Codeerzeugung gemäß der begleitenden Folien dieses Labors durchgeführt werden. Der Parser enthält auch die Funktionen `errorParse()` und `errorType(Node* node)`, welche

bei einem Fehlerhaften Input Programm aufgerufen werden und eine Fehlernachricht ausgeben. Durch die Funktion `nextToken()` wird der Zeiger auf das aktuelle Token auf das nächste Token, durch den Aufruf der gleichnamigen Funktion des Scanners, gesetzt und die `freeToken()` Funktion des Scanners aufgerufen um den Speicher freizugeben. Dadurch wird stets nur ein einziges Token im Speicher gehalten.

6.3 Implementierung

6.3.1 Erstellung des Parse-Tree

Bei der Erstellung des `Parse-Tree` wird hauptsächlich durch den Einsatz von `switch-case` Anweisungen über den aktuellen `TokenType` entschieden, welcher Code ausgeführt werden soll. Des weiteren wird durch die Verwendung von verschachtelten `if-else` Anweisungen innerhalb einer `switch-case` Anweisung geprüft, ob die nachfolgenden Tokens den Regeln der Grammatik entsprechen. Bei einem Fehler wird die `errorParse()`-Funktion aufgerufen. In allen Funktionen, die für die Erzeugung des `Parse-Tree` aus dem Input Programm relevant sind wird am Anfang der Funktion der `RuleType` gesetzt und die entsprechende `Node` angelegt. Falls es hierbei einen echten ϵ-Übergang gibt, also einen ϵ-Übergang, der als solcher in den Regeln der Grammatik gekennzeichnet ist, wird diese `Node` in unserem Compiler Projekt wieder entfernt (z.B. DECLS mit ϵ-Übergang). Dies führte allerdings im Nachhinein zu Kompilkationen in den Funktionen `checkType(Node* node)` und `makeCode(Node* node)`. Am Ende jeder Funktion, die zur Erzeugung des `Parse-Tree` relevant ist, wird die Funktion `nextToken()` aufgerufen.

6.3.2 Typ-check

Die Funktion `checkType(Node* node)` ist für den Typ-check zuständig. Hierbei wird der `Parse-Tree` vom Root Knoten ausgehend rekursiv durchlaufen. Auch in dieser Funktion wird mittels einer `switch-case` Anweisung entschieden, welcher Code ausgeführt werden soll. Die `switch-case` Anweisung ist in dieser Funktion abhängig vom `RuleType` der aktuellen `Node`. Durch die zuvor erwähnte Entfernung von Knoten mit einem echten ϵ-Übergang mussten in unserer Version zusätzliche Prüfungen auf `NULL` eingebaut werden. Weiterhin mussten verschiedene Vorgehensweisen implementiert werden für den Fall, dass ein bestimmter Knoten existiert und für den Fall, dass er nicht existiert. Ein Beispiel hierfür wäre die Unterscheidung zwischen einer Variablen vom Typ `intType` und `intArrayType`, da die `checkType(Node* node)` Funktion für einen array Knoten aufgerufen wird und einen

Typen `DataType` zugewiesen bekommt, bevor der zugehörige identifier einen Typen zugewiesen bekommt.

6.3.3 Codeerzeugung

Die Funktion `makeCode(Node* node)` ist für die Codeerzeugung zuständig. Auch in dieser Funktion wird der `Parse-Tree` vom Root Knoten ausgehend rekursiv durchlaufen. Genauso wie beim Typ-check wird in dieser Funktion über eine `switch-case` Anweisung, die vom `RuleType` der aktuellen `Node` abhängig ist, entschieden, welcher Code ausgeführt werden soll. Hierbei fiel ebenfalls durch die zuvor erwähnte Entfernung von Knoten mit einem echten ϵ-Übergang einiges an Zusatzarbeit an. Es handelt sich dabei im Grunde um die selben Probleme, die auch beim Typ-check aufgetreten sind, welche auf die selbe Art und Weise gelöst wurden. Für die Vergabe einer eindeutigen Sprungmarke im erzeugten Code wurde die Hilfsmethode `makeLabel()` implementiert, die an das Wort "label" fortlaufend die Buchstaben "a - z" und anschließend "A - Z" anhängen sollte. Bei der Funktion `makeCode` handelt es sich des weiteren um eine überladene Funktion. Die Funktion `makeCode(char* inputFileName)` bekommt das Input Programm und erzeugt eine Output Datei mit dem selben Namen wie die Input Datei, allerdings mit der Dateinamenserweiterung ".code" statt ".txt" und ruft im Anschluss die Funktion `makeCode(Node* node)` mit dem Root Knoten auf.

6.4 Programmaufruf

Um den Compiler nutzen zu können, muss diesem nur eine Input Datei übergeben werden, die vom Scanner genutzt wird. Die Output Datei wird vom Compiler selbst erzeugt und erhält den Namen der Input Datei, allerdings mit der Dateiendung ".code". Ein genereller Aufruf des Compilers bzw. des ausführbaren Programms `ParserTest`:
pfad/zur/ausführbaren/datei/ParserTest pfad/zur/input/datei
Ein beispielhafter Aufruf des Compilers bzw. des ausführbaren Programms aus dem Ordner, in dem sich die ausführbare Datei `ParserTest` befindet:
./ParserTest ./test.txt

7 Tests

Für diverse Szenarien haben wir eine ganze Reihe von Testdateien erstellt. Einige sind im Folgenden umschrieben. Für die Validierung haben wir uns unserer debug Funktionen bedient, welche in der Lage waren einen Parse-Tree mithilfe von Ascii-Zeichen auszugeben. Dadurch konnten wir relativ schnell sehen, wo der Fehler zu finden ist.

`parserTest.txt` Das kleine Testprogramm auf Seite 16 der begleitenden Folien.

`declsOnly.txt` Ein Besipielprogramm, das nur aus Deklarationen besteht. Diese wurde zu debugging Zwecken für den `DECLS` Pfad erstellt.

`statmtsOnly.txt` Ein Beispielprogramm, das nur aus Anweisungen besteht. Diese wurde zu debugging Zwecken für den `STATEMENTS` Pfad erstellt.

`ruleIfElse.txt` Ein Beispielprogramm, welches mehrere `if-else` Anweisungen, sowie `read` und `write` Anweisungen enthält. Ein Auszug findet sich in Quelltext 1.

```
1  if(abc[0])
2      write(0)
3  else
4      write(-1)
5  ;
6  if(7 * 7 =:= 56)
7      write(0)
8  else {
9      write(1);
10     write(2);
11     write(3);
12 }                    ;
13
14 if(1) x := 3;else write(1) ;
```

Quelltext 1: *ruleIfElse.txt*

`empty.txt` Eine leere Eingabedatei. Diese Testdatei hat sehr bei der Überprüfung, ob leere Knoten wieder vom `Parse-Tree` entfernt werden geholfen.

`ruleExp.txt` Eine kleine Beispieldatei, die aus verschiedenen aufeinanderfolgenden Ausdrücken besteht. Ein Auszug findet sich in Quelltext 2.

```
1  abc := (abc) + (abc - abc * abc) : abc < abc >
2  abc = (abc =:= ((abc) && abc));
3
4  true := ---(---7);
5
6  not := x =:= y;
```

Quelltext 2: *ruleExp.txt*

`ruleBrackets.txt` Eine kleine Beispieldatei, die aus vielen Ausdrücken innerhalb von geschachtelten Klammern besteht. Ein Auszug findet sich in Quelltext 3.

```
1 write(
2   i[(abc[1]) + (abc - abc * abc) :
3     abc < abc > abc = (abc =:=
4     ((abc) && abc))]
5   : 2
6 );
```

Quelltext 3: *ruleBrackets.txt*

`ruleWhile.txt` Eine Beispieldatei, die sich hauptsächlich mit while-Schleifen beschäftigt.

`ruleRead.txt` Eine kleine Beispieldatei, die verschiedene read-Anweisungen enthält.

`ruleWrite.txt` Eine kleine Beispieldatei, die verschiedene write-Anweisungen enthält.

8 Schluss

Insgesammt hat uns die Arbeit am Compiler Projekt viel Spaß gemacht und wir haben sowohl die Verwendung als auch die Implementierung von Datenstrukturen weiter vertiefen können. Durch die großzügige Verwendung von Zeigern in diesem Projekt konnten wir auch unser Verständnis diesbezüglich erweitern und festigen. Es war auch spannend zu sehen, wie ein Compiler funktioniert und wie die beteiligten Module miteinander Verbunden sind und miteinander interagieren. Obwohl es sich bei diesem Projekt um einen eher einfachen und sehr abgespeckten Compiler handelt, hat dieser zum Ende trotzdem eine ordentlich Größe erreicht und besteht auch vielen verschiedenen Klassen, interfaces und Aufzählungstypen. Dadurch konnten wir wie zum Beispiel im Kapitel 3 beschrieben nocheinmal den Vererbungsmechanismus der Sprache C++ anwenden und vertiefen. Wir konnten uns im Rahmen dieser Arbeit auf jeden Fall vieles an praktischem Wissen aneignen und sind mit dem von uns erreichten Gesammtergebnis durchaus zufrieden. Nach diesem Labor wird es uns in Zukunft auch leichter fallen, die Ursachen für bestimmte Fehlermeldungen zu finden.