Systemnahe Programmierung eines Compilers
I. Der Scanner

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

21. Juni 2017

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Aufgabenstellung gesamt 1

1.2 Aufgabenstellung Scanner – diese Arbeit 1

2 Der Puffer **1**

2.1 Aufgabe des Puffers 1

2.2 Erklärung der Funktion 1

2.3 Implementierung 2

3 Der Automat **2**

3.1 Aufgabe des Automaten 2

3.2 Implementierung 3

3.3 Aufbau der Zustände 4

4 Die Symboltabelle **6**

4.1 Aufgabe der Symboltabelle 6

4.2 Implementierung 7

4.2.1 Hashmap 8

4.2.2 Stringtabelle 9

5 Der Scanner **9**

5.1 Aufgabe des Scanners 9

5.2 Implementierung 9

5.3 Programmaufruf 10

6 Tests **11**

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 Scanner – diese Arbeit

Als Teil des Compilers ist die Aufgabe des Scanners eine Source-Datei einzulesen und in ihre syntaktischen Bestandteile zu zerlegen - man pricht von *tokenizing*. Dazu bedarf es zum einen eines Puffer-Speichers für den Dateiinhalt. Zum anderen ist die Token-Erkennung mithilfe eines deterministischen endlichen Automaten umgesetzt. Parallel zur syntaktischen Analyse werden bereits erste Informationen über die Token gesammelt. Diese hält eine Symboltabelle bereit, welche als hash-map realisiert wird.

Im folgenden wird von den Datenstrukturen `Token`, `Key` und `Information` gesprochen. Die genauere Beschreibung dieser findet sich später in den Kapiteln 4 und 5.

2 Der Puffer

2.1 Aufgabe des Puffers

Um die Langsamkeit der I/O-Operationen der Festplatte abzufedern, wird die zu kompilierende Datei in den Arbeitsspeicher geladen. Da dies bei sehr großen Dateien widerrum Probleme bereiten kann, geschieht dies in Schritten von 512 Byte. Der Puffer bietet an den Inhalt der Datei Zeichen für Zeichen zu erhalten. Außerdem erlaubt er, wieder zurück im Zeichenstrom zu gehen (siehe Automat 3).

2.2 Erklärung der Funktion

Der Puffer besteht aus zwei Speicherbereichen (jeweils 512 Bytes groß). Diese werden wechselseitig mit dem Inhalt der zu kompilierenden Datei gefüllt. Die Speicherbereiche werden

dabei nur einmal angelegt und durch das Umlegen eines entsprechenden Zeigers angesprochen. Um das Neu-Einlesen des nächsten Datei-Chunks zu steuern, existiert außerdem eine Bool'sche Variable, die angibt, ob der nächste Speicherbereich alt ist und neu eingelesen werden muss. Oder ob er bereits neu befüllt wurde und wir durch ein `unget()` oder initial in den aktuell hinteren Bereich gelangt sind.

Am Ende der Datei liefert der Puffer ein Nullterminal `'0'`.

2.3 Implementierung

Um höchste Performanz zu erreichen, erfolgt die Allokierung des Speicherbereichs mittels `posix_memalign(...)` mit einem *alignment* des Speichers, welches der Puffergröße entspricht. Dies beugt Fragmentierung vor und beschleunigt das Einlesen von einem *block-device* unter Linux.

Beim ersten Einlesen mittels `ssize_t read (int, void*, size_t)` wird versucht gleich beide Bereiche zu füllen, wenn die Datei groß genug ist. Wir starten also im hinteren Puffer; die Bool'sche Variable ist initial entsprechend `was_swapped_back = true;` gesetzt. Dies erlaubt eine einfachere Behandlung des unwahrscheinlichen Falls, dass ein `unget()` zu Beginn der Datei aufgerufen wird.

Der Puffer bietet nach außen (gegenüber dem Automaten) in erster Linie zwei Funktionen an:

`char Buffer::get(void)` um das nächste Zeichen der Datei zu erhalten.

`void Buffer::unget(void)` um im Puffer ein Zeichen zurück zu gehen.

Alle weiteren Funktionen sind privat und dienen dem Management von Datei-Ein-/Ausgabe und Speicherverwaltung.

3 Der Automat

3.1 Aufgabe des Automaten

Der Automat hat die Aufgabe in einer Zeichensequenz eine Tokensequenz zu erkennen. Dazu wird die Logik eines *endlichen deterministischen Automaten* (FSM, *finite state machine*) genutzt, wie in 2 abgebildet. Ein FSM ist dazu geeignet, eine reguläre Grammatik (CHOMSKY-Hierarchie 3) zu erkennen, was für diesen Zweck völlig ausreichend ist. Dies dient als Vorstufe, damit der Parser in der CHOMSKY-Hierarchie 2 der kontextfreien Sprachen arbeiten kann. Die Eingabe in den Automaten ist das jeweils nächste Zeichen in der zu kompilierenden Datei. In den Zuständen des Automaten passiert nichts, dafür bei den

Übergängen – wir haben also einen MEALY-Automaten. Da in Kommentaren der Sprache alle Zeichen erlaubt sind, ist das Alphabet des Automaten effektiv unendlich groß. Praktisch lesen wir zum einen nur 8-bit-Werte ein (C++ `char`-Typ) und prüfen zum anderen nur auf ASCII-Zeichen. So lässt sich das Alphabet mit $\sum = [0{-}9A{-}Za{-}z{-}+ : * <>=: !\&;()[]\{\}'\backslash 0']$ beschreiben.

Im Übrigen wird im Automaten aufgezeichnet, in welcher Zeile und Spalte der Datei wir uns gerade befinden, da hier (neben dem Buffer) sicher jedes Zeichen einmal behandelt wird und bekannt ist, ob per `unget()` doppelt gelesen wurde. Dazu wird auch die Variable `int Automat::count` verwendet, welche so einen mehrfachen Zweck erfüllt:

1. Sie zeichnet auf, wie weit zurück ein `unget()` gehen muss.
2. Sie merkt sich, wo der Beginn des aktuellen Tokens ist. In Compilern ist es üblich, Warnungen mit Zeilen- und Spaltennummern auf den Beginn eines Tokens zu beziehen. Dies tun wir damit genau so.
3. Sie gibt die länge des aktuell zwischengespeicherten Strings an, der entweder mit `long strtol(char*)` ausgewertet, oder in der Symboltabelle gespeichert, oder als unbekanntes Zeichen ausgegeben wird.

3.2 Implementierung

Für die Umsetzung eines FSM gibt es hauptsächlich zwei Ansätze:

Per Adjazenz-Matrix mit der Dimension $|Alphabet| \times |States|$, in welcher der Zielzustand eines Übergangs von einem Startzustand bei einem Eingabezeichen verzeichnet ist. Der Nachteil ist die Größe dieser Tabelle bei unserem Eingabealphabet.

Durch direkte Implementierung der Übergänge als Funktionen, bzw. als `switch-case`-Weichen innerhalb der `read(char)` Methode eines jeden Zustands.

Wir wählten den zweiten Ansatz. Dazu erstellten wir eine Klassenhierarchie wie in 1 zu sehen.

Wir folgten bei der Implementierung der Logik, dass diese Zustände für sich stehen, und nicht mit jedem für eine zu kompilierende Datei erstellten Automaten instantiiert werden müssen. Daher sind diese Klassen `static` definiert. Der Automat übergibt sich selbst (`this`) an den lesenden Zustand. Dieser setzt dann den neuen Zustand im Automaten und ruft u.U. über diesen den Scanner auf, um ein Token zu erzeugen. Vgl. dazu Quelltext 1.

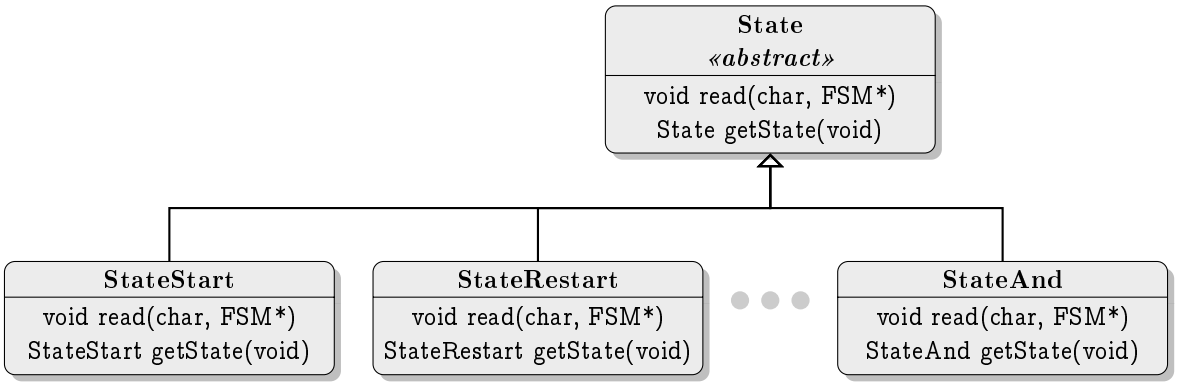

Abbildung 1: *(Gekürztes) Klassendiagramm der Zustände.*

Bei diesem Ansatz ist die ODR-Regel [i] in C/C++ zu beachten, nach der die Definition der entsprechenden statischen Instanzen nur genau einmal erfolgen darf und muss. Wir brachten diese Definitionen dazu in einer eigenen Header-Datei unter, welche mit den entsprechenden *include guards* versehen ist.

Die Zustände erwarten über den Automaten die Methode `Scanner::mkToken(TokenType)` aufrufen zu können. Außerdem fügen sie mittels `Automat::incrementAndAppend(char)` an entsprechender Stelle Zeichen zur späteren Verarbeitung hinzu.

Wir haben mit dem Zustand `StateRestart` einen besonderen Zustand eingebaut, der dazu da ist einen kontrollierten Übergang nach einem `mkToken()` zurück in `Start` durchzuführen. Restart wird betreten, wenn die Variable `Automat::count` zurückgesetzt werden muss und das letzte Zeichen nochmal von Start eingelesen wird (sprich, ein `unget(1)` passiert). Dazu ist nicht unbedingt dieser spezielle Zustand nötig. Wir fanden das aber eine elegante Lösung als Ersatz für sich überall wiederholenden Code.

Andere Zustände, die logisch erscheinen, haben wir widerrum eingespart. Es gibt zum Beispiel fast keine finalen Zustände jenseits von `Start`. Stattdessen geschieht das Akzeptieren einer Zeichensequenz in anderen Zuständen mit einem Übergang in Start, respektive Restart, und der zugehörigen Tokenerzeugung. Somit verfügen unsere Zustände auch nicht über ein Attribut `bool State::isFinal` oder dergleichen.

3.3 Aufbau der Zustände

Der Aufbau des Automaten bzw. die Zusammenhänge der Zustände in `states.cpp` ist in Bild 2 gezeigt. Es sind zwar logische Endzustände als solche markiert. Dies hat für den

[i] *ODR: one definition rule*

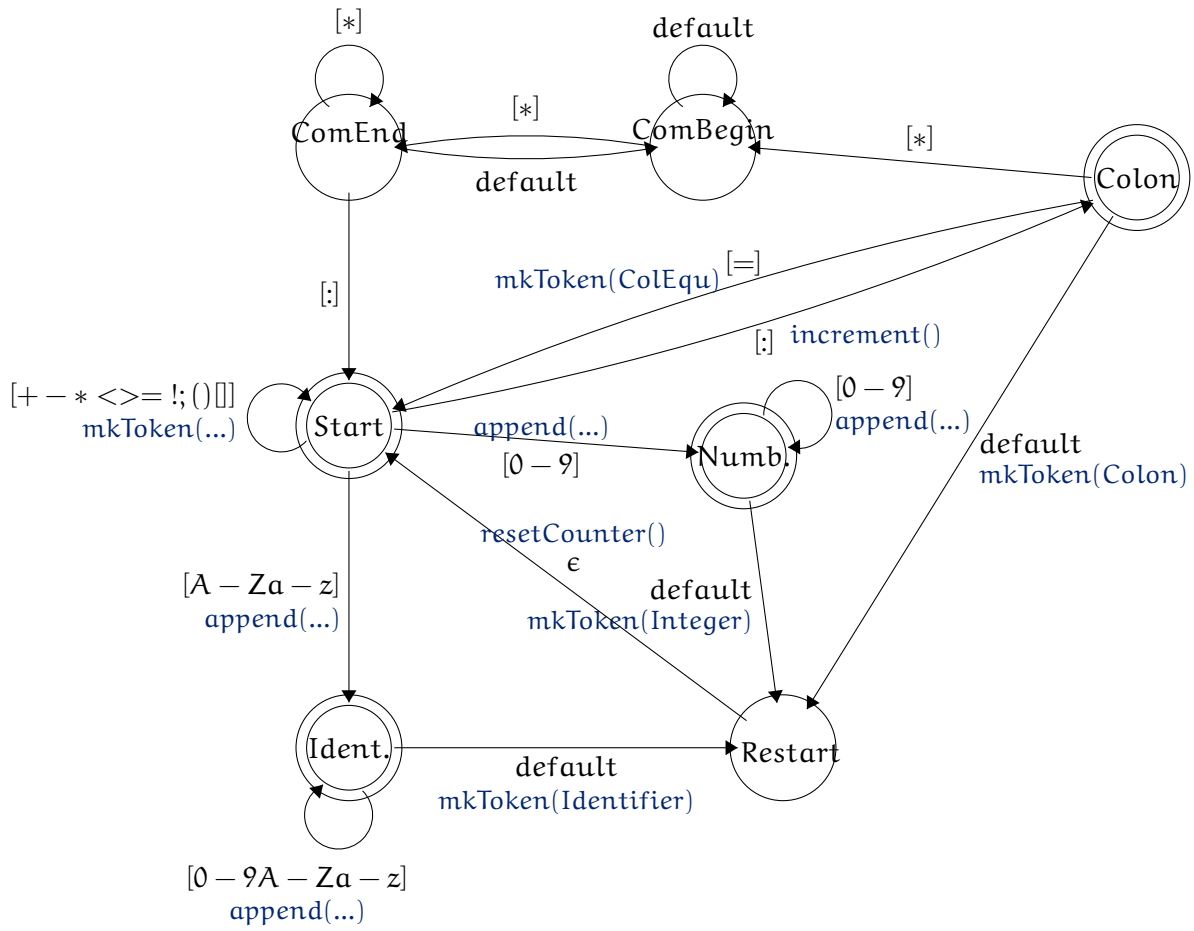

Abbildung 2: *Die Zuständes des Automaten, welche die Token(gruppe)* `SingleSign`, `Identifier` *und* `Number` *erzeugen, sowie Kommentare übergehen.* default *steht hier für den Rest von* $\sum$*, der keinen anderen Übergang aus diesem Knoten hat.*

Ablauf aber keiner weitergehende Bedeutung. Es zählt vor allem, welche Funktionen bei den Übergängen aufgerufen werden.

Die Methode `Scanner::mkToken(TokenType)` löst im Scanner die Erzeugung eines Tokens aus. Die Methode `void Automat::increment(void)` erhöht die Variable `int Automat::count` um 1, wohingegen `void Automat::incrementAndAppend(char)` dies ebenfalls tut, aber zusätzlich das jeweilige Zeichen für die spätere Aufzeichnung in der Symboltabelle bzw. Berechnung mittels `strtol()` vorhält.

Jeder der Zustände implementiert dann die entsprechenden Übergänge mittels einer switch-case-Struktur. Diese ist beispielhaft für den Zustand `Number` in Listing 1 gezeigt.

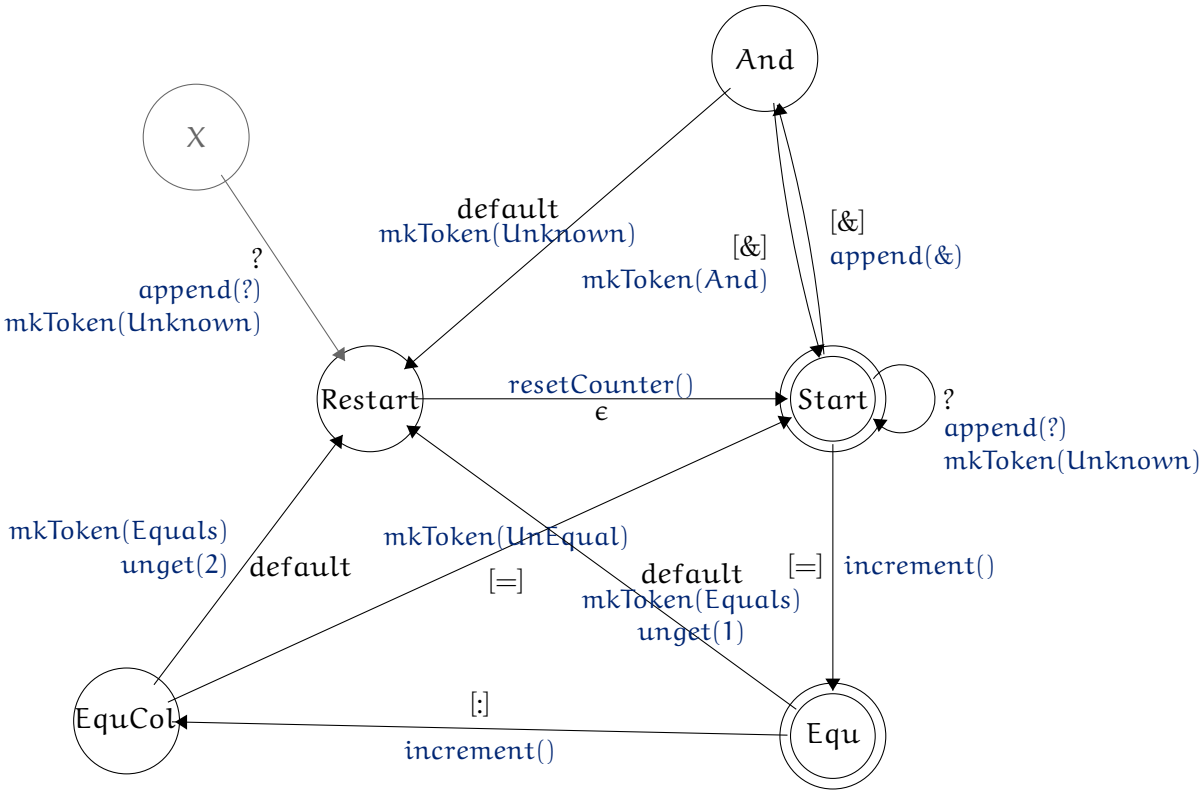

Abbildung 3: *Die Zuständes des Automaten, welche die Token* `And`, `Equal`, `ColonEqual`, `UnEqual` *und* `Unknown` *erzeugen. Die Zustände* `Start` *und* `Restart` *sind die selben wie in Abbildung 2. Der Zustand* `X` *steht für jeden möglichen der anderen Zustände dort, außer* `comBegin` *und* `comEnd`*, da dort nicht-Alphabet Buchstaben erlaubt sind. Das* `?` *steht für ein Zeichen außerhalb des Automaten-Alphabets. Das Nullterminal (EOF) ist der Übersichtlichkeit wegen nicht verzeichnet.*

4 Die Symboltabelle

4.1 Aufgabe der Symboltabelle

Die Symboltabelle enthält weitergehende Informationen über Identifier-Token, insb. ob gefundene Identifier einem *keyword* der Sprache entsprechen oder sonst bereits bekannt sind. Hierzu verfügt die Symboltabelle über einen Speicher aller einzigartigen Identifier-Strings, welche mit den Keywords initialisiert wird.

In Abbildung 4 ist der Zusammenhang zwischen dem `struct Token` und den Klassen `Key`, `Information` und der Stringtablle gezeigt. Das Token, der letztlich vom Scanner zurückgegeben wird, enthält zusätzlich zu seinem Typ, Zeile und Spalte noch einen Key-Pointer, falls es sich um einen Identifier handelt, oder ein `long int value` im Falle eines TokenInteger. Der Key kommt von der Symboltabelle und enthält neben dem `hash` ein In-

```
116 void StateNumber::read(char c, Automat* m) {
117   switch (c) {
118   case '0' ... '9':
119     // integer continues
120     m->incrementAndAppend(c);  // remember c for later strtol()
121     // stay in StateNumber
122     break;
123   case ' ':
124   case '\n':
125   case '\t':
126     // shortcut: don't re-read whitespace
127     m->setCurrentState(StateStart::makeState());
128     m->getScanner()->mkToken(TokenType::TokenInteger);
129     break;
130   default:
131     // integer is finished:
132     m->getScanner()->mkToken(TokenType::TokenInteger);
133     m->setCurrentState(StateRestart::makeState());  // re-read c
134     m->getCurrentState()->read(c, m);      // epsilon transition
135   }
136 }
```

Quelltext 1: *Ausscchnitt aus* `states.cpp`. *Die* `read()`*-Methode des Zustands* `Number`.

formationsobjekt. Dieses erfüllt hier noch keinen großen Zweck, sondern wird erst später beim Parser nützlich werden. Soweit enthält es einen Pointer auf das Lexem des Identifiers in der Stringtabelle.

4.2 Implementierung

Die Symboltabelle verfügt über drei öffentliche Methoden:

`Key* Symboltable::insert(const char* lexem)` erlaubt das Einfügen eines neuen Strings. Es wird ein Key-Objekt zurückgegeben, welches u.a. einen Pointer in die Stringtabelle enthält. Existiert der String bereits in der Symboltabelle, wird der bereits existierende Key zurückgegeben.

`Information Symboltable::lookup(Key key)` holt exitierende Einträge aus der Tabelle. Nützlich für den Parser.

`unsigned int Symboltable::hash(const char* lexem)` ermittelt den hash eines Lexems, ohne in die Tabelle einzufügen. Eine Hilfsfunktion für den Scanner, der damit schneller überprüfen kann, ob ein Identifier ein Keyword ist. Siehe 5.2.

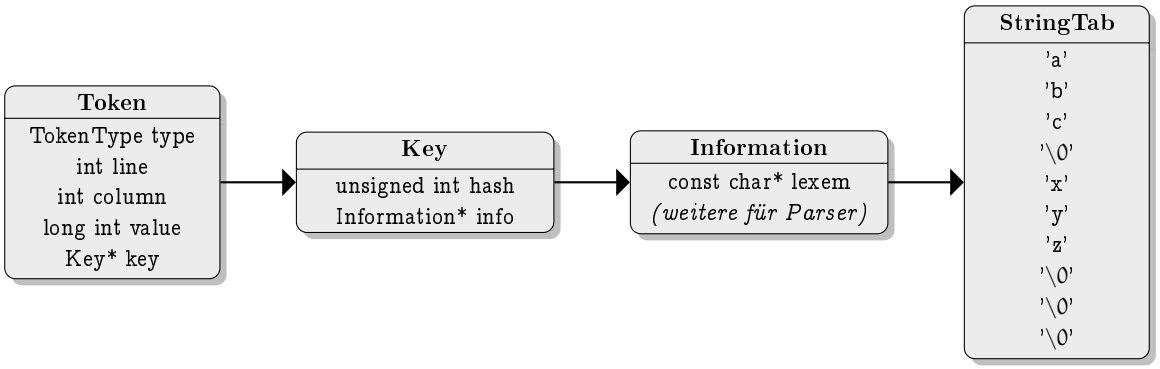

Abbildung 4: *Zusammenhang zwischen dem* `struct` *bzw. den Klassen* `Token`, `Key`, `Information` *und* `Stringtable`.

Die Symboltabelle setzt sich aus folgenden zwei Bestandteilen zusammen.

4.2.1 Hashmap

Die eigentliche Tabelle ist eine *Hashmap*, die ein schnelles Einfügen und Extrahieren von Einträgen erlaubt. Die Hashfunktion ist in 2 zu sehen. Es handelt sich um eine recht einfache multiplikative Hashfunktion, die bei unseren Tests sehr gute Resultate ergab was Kollisionszahl und Berechnungszeit angeht.

```
172 unsigned int Symboltable::hash(const char* lexem) {
173   // lexem = NULL won't work Check beforehand
174   unsigned int hash = 0; // SEED = 0;
175   int i = 0;
176
177   while (lexem[i]) {
178     // optimized: SALT = 65599 = (2^16 + 2^6 - 1)
179     hash = (hash << 16) + (hash << 6) - hash + lexem[i];
180     i++;
181   }
182   return hash;
183 }
```

Quelltext 2: *Die Hashfunktion.*

`hash(const char*)` gibt den *vollen Hash* zurück, welcher ebenfalls im Key gespeichert wird. Um den eigentlich Tabellenplatz zu finden, muss noch `hash % current_table_size` berechnet werden.

Mit zunehmender Anzahl von Einträgen, vergrößert sich die Tabelle. Die Hashmap startet mit einer Größe, so dass die anfangs eingefügten 9 Keywords (mit alternativen Schreibweise) kein `resize()` erzeugen. Dieser passiert bei einem *Füllstand* von 75 %.

Der Resize verdoppelt die Größe der Tabelle. Danach müssen alle Einträge neu eingefügt werden, da sich ihr Position ändern kann. Da wir den vollen Hash bei den Keys gespeichert haben und dieser sich bei konstantem Lexem nicht mit der Tabellengröße ändert, erleichtern wir uns an dieser Stelle die Arbeit, indem wir nur `Key.getHash() % current_table_size` berechnen müssen, um die neue Position zu erhalten.

4.2.2 Stringtabelle

Die Stringtabelle ist ein Speicherbereich, in dem jeder angetroffene Identifier genau einmal eingetragen wird. Hierbei liegen bei unserer Implementierung die *C-style* Strings direkt hinereinander, d.h. sie werden von Nullterminatoren getrennt. Beim Einfügen in die Symboltabelle wird im Falle einer Nicht-Kollision das entsprechende Lexem in die Stringtabelle kopiert und dabei ein Information-Objekt mit einem Zeiger auf den neuen Eintrag erzeugt. Ist der Speicherbereich zu klein für den neuen String, wird

1. Ein neuer doppelt so großer Bereich allokiert.
2. Der Inhalt des alten Bereichs hinüberkopiert.
3. Der alte Bereich freigegeben.
4. Für alle ausgegebenen Information-Objekte der Pointer angepasst.

5 Der Scanner

5.1 Aufgabe des Scanners

Die Scannerklasse fasst alle vorherigen Bestandteile zusammen und ermöglicht die Generierung einer Tokensequenz mittels wiederholtem Aufruf von `Token Scanner::nextToken(void)`. Die Sequenz endet mit dem speziellen Token `TokenStop`, welches durch das Einlesen des Nullterminals `'\0'` bzw. `EOF` erzeugt wird.

5.2 Implementierung

Die Methode `Token Scanner::nextToken(void)` 'füttert' solange Zeichen an den Automaten, bis in einem seiner Zustände `void Scanner::mkToken(TokenType)` aufgerufen wird.

Dies setzt die Schleifenvariable `no_token = false`, so dass das Token erzeugt und zurückgegeben wird.

In Abhängigkeit des TokenTyps, geschehen noch weitere Aktionen:

Bei TokenUnknown wird das das unbekannte Zeichen vom Automaten geholt und im Token gespeichert.

Bei TokenIdentifier wird geprüft, ob es sich um ein Keyword handelt. Dazu wird erst verglichen, ob der `hash` des Lexems einem Keyword-hash entspricht. Wenn dies der Fall ist, werden die Lexeme der jeweiligen Keywords mit den neuen String zeichenweise verglichen. Falls dies Gleichheit ergibt, wird der Typ entsprechend angepasst (TokenIf, TokenWhile, etc.). Falls der Identiier kein Keywordist, wird das Lexem vom Automaten geholt und in die Symboltabelle eingefügt. Der von dieser erhaltene `Key` wird im Token gespeichert.

Bei TokenInteger wird der geparste Zahlenwert vom Automaten geholt und im Token gespeichert.

In jedem Fall wird Zeile und Spalte im Token gespeichert.

5.3 Programmaufruf

Die ausführbare *ScannerTest* liest als Parameter eine beliebig lange Liste von Dateinamen ein. Zum Beispiel wie in Listing 3 alle Dateien eines Ordners mittels Bash-Expansion.

```
1 ~/project/$ ./Scanner/debug/ScannerTest tests/*
```

Quelltext 3: *Aufruf von ScannerTest*

Im Programm wird für jede Datei jeweils ein Scanner-Objekt erzeugt. Dieses beauftragt den Buffer damit, die Datei zu öffnen. Es werden alle Zeichen eingelesen - gescannt - und die jeweiligen Ergebnistoken für alle Eingabedateien zusammen in eine Datei out.txt geschrieben. Dies sieht dann zum Beispiel aus wie in Listing 4.

```
57 [...]
58 TokenStop            in line 3     in column 0
59    --- END OF Scanner/tests/empty.txt ---
60    --- BEGIN Scanner/tests/theBible.txt ---
61 TokenIdentifier      in line 1     in column 1      Lexem: The
62 TokenIdentifier      in line 1     in column 5      Lexem: Project
63 TokenIdentifier      in line 1     in column 13     Lexem: Gutenberg
64 TokenIdentifier      in line 1     in column 23     Lexem: EBook
```

```
65 [...]
```

Quelltext 4: *Dateiausgabe von ScannerTest*

Parallel werden mögliche Fehler und weitere Informationen auf der Console ausgegeben.

6 Tests

Für diverse Szenarien haben wir eine ganze Reihe von Testdateien erstellt. Einige sind im Folgenden umschrieben.

`anything.txt` Ein potentiell sinnvolles Codebeispiel mit einer bunten Mischung verschiedener Tokens und Kommentare. Ein allgemeiner Testfall ohne besonderen Fokus. Eher als Parsertest nützlich.

`empty.txt` Eine komplett leere Datei, 0 Byte groß. Fokus dieses Testfalls ist die Funktion des Buffers – also ob etwas schief geht, wenn schon zu Beginn nichts einzulesen ist.

`desert.txt` Eine Datei mit verschiedenen Whitespaces: Space, Newline und Tabulator. Fokus dieses Tests ist der Grenzfall, dass der Automat zwar viele Zeichen zu verarbeiten hat, aber nie ein Token entstehen sollte.

`EquCol2.txt` Da wir in diesem Bereich des Automaten lange Zeit Probleme hatten, die `line` und `column` korrekt anzugeben, haben wir unter `http://textmechanic.com/text-tools/combination-permutation-tools/permutation-generator/` alle möglichen Permutationen von `:`, `=`, `:=`, `=:=` und `:**:` erzeugen lassen. Ein Auszug findet sich in Quelltext 5.

```
1  =:=:=:=:**:
2  =:=::**:=:=
3  =:==:=::**:
4  =:==:=:**::
5  =:=:**::=:=
6  =:=:**:=:=:
7  =::=:**:=:=
8  =::==:=:**:
9  =:=:=:**::=
10 line10
11 =:=:=:=:**:
12 [...]
```

Quelltext 5: *EquCol2.txt*

Anschließend haben wir verglichen, ob

1. `Automat::line` der Anzahl der Zeilen in der Datei entspricht,
2. `Automat::column` niemals < 1 oder > 11 ist,
3. Die Token korrekt und an der richtigen Stelle erkannt werden. (zumindest stichprobenweise)

Prinzipiell wäre es wünschenswert, alle möglichen Kombinationen aller Tokens zu testen. Dies würde aber mit $21! = 5 \cdot 10^{19}$ Möglichkeiten unsere Kapazitäten sprengen.

`legalComments.txt` Dieser Test fokussiert sich auf alle erdenklichen Fälle um und in Kommentaren (: ∗∗ :). Innerhalb sind alle Zeichen erlaubt. Insbesondere auch " : " und " ∗ " allein. Desweiteren endet der letzte Kommentar nicht (bzw. mit EOF).

`theBible.txt` Dieser Test enthält den vollständigen Text der King-James-Bibel unter `http://www.gutenberg.org/cache/epub/10/pg10.txt` und dient vor allem der Performanz- und Speicherleck-Überprüfung in der Symboltabelle.

`longWords.txt` Enthält Identifier mit versch. Wortlängen, auch jenseits der doppelten Pufferbreite. Fokus ist Performanzüberprüfung der Symboltabelle.

`manyWords.txt` Enthält viele, auch sich wiederholende Identifier. Fokus ist Performanz- und Konsistenzüberprüfung der Symboltabelle bei wiederholtem `resize()`.

`unknown.txt` Enthält alle dem Scanner unbekannten ASCII-Zeichen, sowie entsprechend zu händelnde Situation wie unmittelbar aufeinander folgende unbekannte Zeichen und unbekannte Zeichen inmitten von Zahlen und Identifiern.

`values.txt` Enthält viele Zahlenwerte, die mit `strtol()` zu parsen sind.