Scanner Dokumentation

# Der Scanner

Bei der "Systemnahes Programmieren" Veranstaltung ist ein Compiler zu kreieren. Diese erfolgt, indem zwei Hauptkomponenten implementiert werden: der Scanner und der Parser. Jeder Modul ist ein wichtiges Bestandteil eines Compilers.

**Theoretisches**

Der Scanner beschäftigt sich mit dem Einscannen des Quelltexts. Der Inhalt wird nach Korrektheit im Bezug zu der gegebenen Grammatik geprüft. Diese erfolgt durch erkennen von Tokens. Tokens sind semantische Bestandteile der Sprache. Die Wörter sind dann die Ausprägung des jeweiligen Tokens. So ist, beispielsweiße, *TokenInteger* ein Token, der alle mögliche (im integer Bereich) ganzzahlige Werte haben kann.

Folgende Grammatik wird geprüft: L (sign+|...|sign]|integer|identifier|if|while). Dabei sind

* digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
* letter ::= A | B | C | ... | Z | a | b | c | ... | z
* sign ::= + | - | : | \* | < | > | = |:= | =:= | ! | && | ; | ( | ) | [ | ] | | { | }
* integer ::= digit digit\*
* identifier ::= letter (letter |digit) \*
* if ::= if |IF
* while ::= while |WHILE

**Vorgehensweise**

Der Scanner wird anhand eines Dateipfades sowie einer SymbolTable initialisiert. Beim Aufbereiten erstellt der Scanner zusätzlich die Puffer- und Automat Instanzen. Die jegliche privat Attributen (*currentColumn*, *currentLine*, *currentToken*) werden mit den Startwerten initialisiert.

Der Existenzgrund des Scanners liegt an der nextToken() Methode. Diese liefert den nächsten Token aus dem Quelltext. Der Parser (oder vielleicht anderer Client wie ScannerTest) ruft diese Methode auf, bis kein Token mehr zurück geliefert wird. Beim Token Auffinden und Erkennen werden alle Module beteiligt.

Der Scanner beinhält einen Verweis auf das aktuelle Token. Dieser Verweis ist zum Anfang leer. Es wird auch manuell bei jedem *nextToken()* Aufruf entleert (der Verweis, nicht der Inhalt). Die *nextToken()* Methode liefert aber genau diesen Verweis zurück. Zwischen der Rückgabe und dem Entleeren liegt aber eine große Zahl der Bearbeitungsschritten, die an weitere Modulen delegiert ist.

Nach dem Entleeren des *currentToken-*s prüft der Scanner, ob der Puffer noch weitere Zeichen liefern kann (e.g. der File wurde erfolgreich geöffnet, der Nullterminator wurde noch nicht abgelesen). Falls ja, liest der Scanner die einzelne Zeichen vom Puffer und übergibt diese an Automat.

Der Automat bearbeitet die an ihn übergegebene Zeichen und schaltet seine Zustände entsprechend. Wenn der Automat ein Token erkennt, ruft er *mkToken(TokenType, String\*)* bei Scanner auf.

Es ist zu verdeutlichen, dass der Automat nur eine grobe Abschätzung zurückliefert. So wird, beispielsweiße, eine *while* Lexem vom Automat als *identifier* erkannt. Dies reicht unsere Zwecken nicht aus. Aus diesen Gründen hat der Scanner schon erwähnte Methode *mkToken(TokenType, String\*)*.

Diese Methode anhand des Typs führt zusätzliche Überprüfungen aus. Ist der *tokenType* gleich *TokenInteger,* zum Beispiel, so wird es versucht die Lexem nach Integer zu umwandeln. Wenn das funktioniert und die Größe des Integeres ok ist, wird ein *IntegerToken* mit dem gegeben Wert instanziiert. Der *currentToken* Verweis wird auf das neu erstellten Token gesetzt. Die *nextToken()* Methode hat nun die Möglichkeit was entsprechendes zu liefern (tut sie auch).

Die *mkToken()* Methode überarbeitet folgendes:

1. Ob das Token nicht zu lang ist (max. 256 Zeichen)
2. Ob der IntegerToken den passenden Wert bekommt
3. Das Token wird zu ErrorToken (das TokenType bleibt aber erhalten!), falls a) oder b) scheitert
4. Falls Token vom TokenIdentifier Typ ist, wird er in die SymbolTabele eingetragen
5. Die currentLine sowie currentColumn werden fürs nächsten Token angepasst mittels *adjustIndicies()*

Um die Token Position zu merken und die unnötige Abfragen zu vermeiden, werden die *currentLine* und *currentColumn* zum Schluss des *mkToken(TokenType, String\*)* Aufrufs gemacht. Dabei geht man alle Zeichen des Lexems nochmal durch. Trifft dabei das *'\n'* auf, so wird *currentLine* inkrementiert, *currentColumn* sonst.

Der Scanner verfügt noch über die *ungetChar(unsigned int)* Methode. Damit wird das aktuelle Position des zum auslesenden Zeichen um die gegebene Anzahl zurück gesetzt. Der Scanner ruft die gleichnamige Methode vom Puffer auf. Weitere Handlungen haben wir im Scanner als ungeeignet gesehen.

**Testen**

Der Scanner verwendet alle Komponenten um die Tokens zu liefern. Aus diesem Grund hat man ihn zum letzt getestet. Zum Testzwecken, hat man zusätzlich die Formatierungsmethoden geschrieben, die die erkannte Tokens entsprechend beschreiben. Die damit erzeugte Strings hat man ins output-<filename> geschrieben.

**Probleme (allgemein)**

Das Hauptproblem, das beim Implementieren entstanden ist, lag am Verstehen eines Interfaces in C++. Dabei war es unklar, wie das Zusammenspiel eines Interface Destruktors sowie den erbenden Klassen genau aussehen sollte. Lösung: leerer virtueller Destruktor im Interface.

Man hat sich auch mit dem richtigen Einfügen den Header-Dateien etwas beschäftigt. Die Guards haben zwar von den zyklischen Abhängigkeiten geschützt, jedoch gab es noch ein Paar stellen, wo die Klassen zu vordeklarieren waren. Im Endeffekt, hat man die Guards mit *#pragma once* ersetzt.

Der *std::ifstream* hat eine Menge Zeit gekostet, bis es verstanden wurde, wie die doppelte *'\n'* am Ende des *get()* Aufruf Ergebnisses bearbeitet werden müssen. Man hat immer noch den Eindruck, dass dieser Aufruf sich immer wieder (abhängig vom Text Editor, in dem der Quelltext gespeichert wurde) unterschiedlich verhält.

Das make hatte viel zu hohen Einstiegspunkt. Man hätte sich vielleicht lieber mit cmake beschäftigt.

# Der Automat

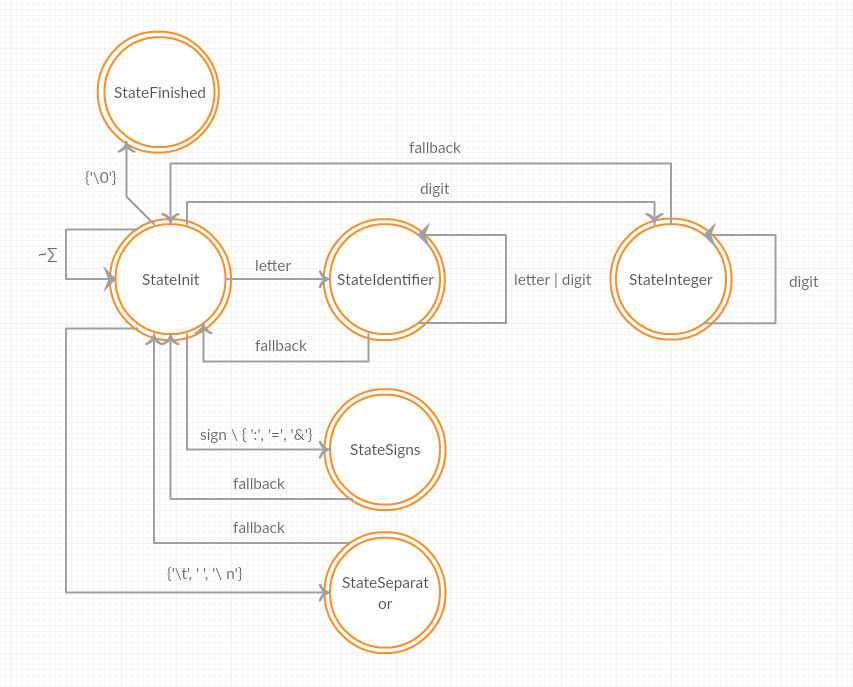
**Theoretisches**

Wie es schon erwähnt wurde, muss der Scanner den Text nach Lexems durchsuchen. Die in der Einführung angegebene Sprache hat eine Grammatik. Diese Grammatik lässt sich mit einem endlichen und deterministischen Automat darstellen. Die Diagrammen veranschaulichen den Automat sowie die zugehörige Zustände und Übergänge.

Der Automat beinhält den Scanner Interface, über den der Zugriff auf die nötige Rückrufe des Scanners erfolgt. Dazu gehören:

* ungetChar(int count)
* mkToken(TokenType tType, String\* lexem)

Weiterhin, hat der Automat ein Zeiger auf den aktuellen Zustand, sowie ein Zeiger auf den zum letzt akzeptierten Zustand. Diese werden bei jedem Zustandswechsel aktualisiert. Die syntaktische Analyse erfolgt dadurch, dass der Scanner Zeichen für Zeichen den Automat versorgt. Die Verarbeitung eines Zeichens entspricht einem deterministischen Übergang. Sind die Tokens während der Verarbeitung erkannt, so werden solche entsprechend generiert. Der Automat ist fertig mit der Bearbeitung, wenn er den EOF Zeichen '\0' bekommt.



Zum Anfang befindet sich der Automat im *StateInit*. Das ist der zentrale Zustand, zu dem immer wieder zurückgekehrt wird. Der allgemeiner Tokenakzeptanz-Vorgang sieht folgender maßen aus:

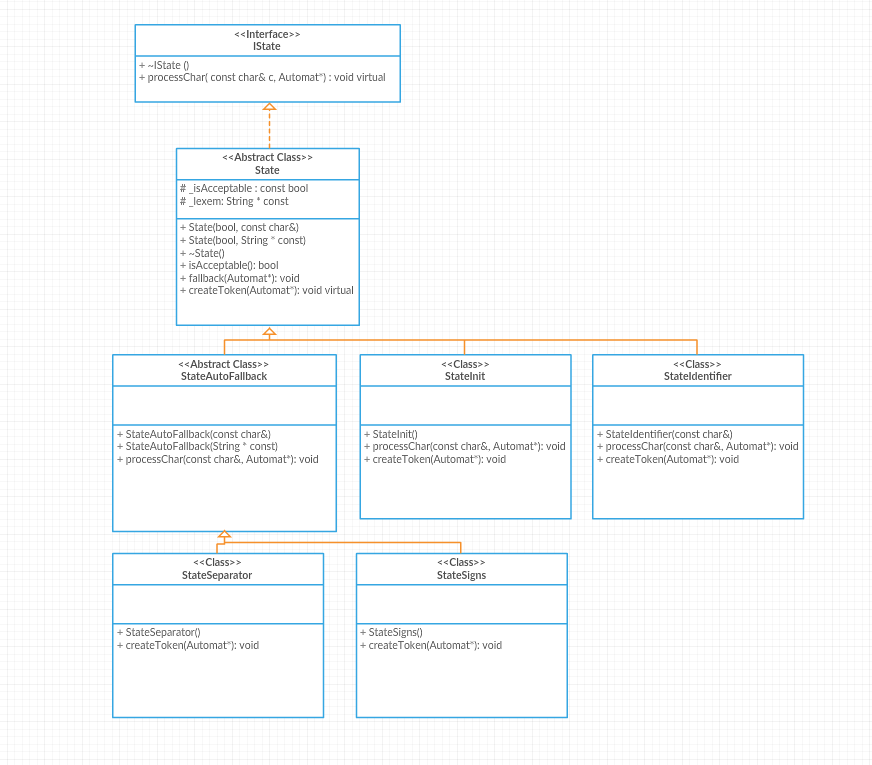
1. Der Automat geht von *StateInit* aus.
2. Der Automat bearbeitet die Zeichen eindeutig abhängig von den aktuellen Zustand. Diese kann zu folgenden Abläufen führen:
   1. Dem Zustandswechsel
   2. Im Zustand bleiben
   3. Nach *StateInit* zurückkehren.
3. Der Automat stoppt, wenn er *'\0'* Zeichen liest.

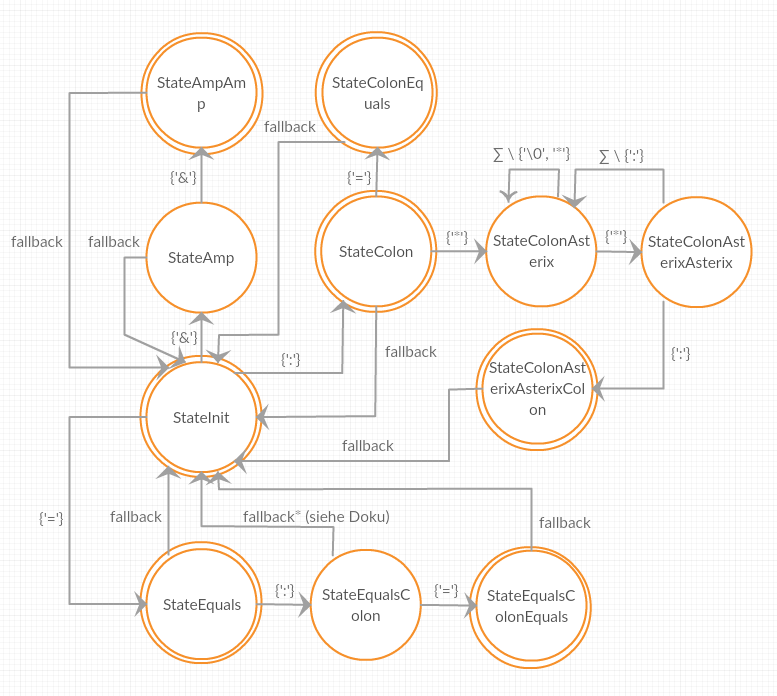
Nach *StateInit* wird erst dann gesprungen, wenn mit dem aktuellen Zeichen keinen weiteren Fortschritt gemacht werden kann. Dabei werden 2 Fälle unterschieden:

* Ist man in einem gültigen Zustand, so wird vom Zustand abhängig das entsprechende Token erzeugt. Danach meldet sich der Automat beim Scanner, dass das aktuelle Zeichen neu bearbeitet werden muss. Schließlich, wird nach *StateInit* gesprungen. Das ganze haben wir als *fallback* benannt.
* Ist man in einem nicht gültigen Zustand, so gibt es wieder zwei Fälle, die sich aus der Grammatiklogik ergeben
  + In einem Fall muss der Automat ein Token von dem zuletzt akzeptierten Zustand generieren, den Scanner mitteilen, dass mehrere letzte Zeichen neu bearbeitet werden müssen (z.B der Fallback von *StateEqualsColon* auf dem 2. Bild). Schließlich, wird wieder nach *StateInit* gesprungen.
  + Im anderen Fall wird der fallback-Prozess ausgeführt. Dabei wird allerdings der *TokenUnknown* erzeugt, was jedem ungültigen Zustand entspricht.

**Implementieren**

Die Zustände des Automats sind objektorientiert entworfen. Zugrunde liegt der *IState* Interface, den die abstrakte Klasse State implementiert. Die Klasse State stellt alle mögliche Aufrufe für die Unterklassen bereit. Viele Klassen wie *StateInit*, *StateInteger*, *StateIdentifier* usw. erben direkt von State. Solche Klassen haben eine etwas komplizierte Schaltungslogik, die entsprechend implementiert ist. Die einfache Klassen wie *StateSigns*, *StateSeparator* erben von *StateAutoFallback*. Dabei müssen die erbende Klassen nur noch das passende *TokenType* mittels *mkToken(TokenType, String\*)* liefern.





# Der Puffer

**Arbeitsweiße**

Der Puffer ist fürs einwandfreies Lesen eines Files zuständig. Er bietet eine klare Schnittstelle, die die furchtbare Implementierungsdetails des *std::ifstreams* verbirgt. Dabei werden noch ein paar zusätzliche Eigenschaften bereitgestellt:

* Um die *MAX\_STEPBACK* Zahl die aktuelle Position zurücksetzten
* Ab den gewissen Block anfangen zu auslesen
* Fehlerbehandlungen

Der Puffer funktioniert über die *Buffer* Klasse. Diese Einheit benutzt weitere Komponenten um die erwähnte Funktionalität anbieten zu können.

Beim Aufbereiten erhält der Buffer vom Scanner den Dateipfad, mit dem der *FileReader* instanziiert wird. Falls damit ein Problem gab (z.B. der File nicht vorhanden) setzt der Buffer den *noFile* Flag. Weiterhin, werden die nötige Attributen initialisiert (e.g. *currentBufferBlock, eofConsumed, stepBackAmount, currentBlockIndex, currentCharIndex*).

Die zwei wichtigsten Methoden des Buffers sind:

* char getChar()
* void ungetChar(unsigned int count)

Der *getChar()*Aufruf holt aus dem aktuellen *BufferBlock* das nächste Zeichen. Falls er nicht vorhanden ist und der Nullterminator nicht bearbeitet wurde und *FileReader* erfolgreich instanziiert wurde, wird der *currentBufferBlock* aus dem *FileReader* geholt mit dem Index gleich 0.

Nun kommt die "elastische" Logik des Puffers. Die Buffer Blöcke sind als Elemente des doppelt verkehrten Lists angelegt. Einer Block kennt sein Vorgänger und Nachfolger. Die zwei sind nicht unbedingt vorhanden, aber deren Existenz nutzt man um die *ungetChar(unsigned int)* auf alle Stellen zu ermöglichen.

Die aktuelle Position des Buffers beinhalten die *currentBlockIndex* und die *currentCharIndex* Variablen. Die maximale mögliche Rücksprung Zahl wird im *stepBackAmount* gespeichert. Dies wird benötigt um die unmögliche *ungetChar(unsigned int)* Aufrufe zu verbieten (e.g. *ungetChar(MAX\_STEPBACK + 1)*. Die Änderungen dieser Variablen erfolgt in den zwei oben genannten Methoden.

Der *currentCharIndex* sowie der *stepBackAmount* werden bei jedem *getChar()* inkrementiert. Der *currentCharIndex* wird aufs 0 gesetzt, falls der *currentBufferBlock* sich aufs nächste wechselt. Der *stepBackAmount* hört sich auf zu inkrementieren, falls er gleich *MAX\_STEPBACK* ist.

Bei jedem *getChar()* Aufruf wird geprüft, ob der *currentCharIndex* die drei wichtige Schwellen erreicht hat. Diese sind:

* MAX\_STEPBACK
* BLOCK\_SIZE - MAX\_STEPBACK
* ENDE

Beim *MAX\_STEPBACK* wird der Vorgänger des aktuellen Buffer Blocks gelöscht, da man ihn sowieso via *ungetChar(unsigned int)* nicht mehr erreichen kann. Beim *BLOCK\_SIZE - MAX\_STEPBACK* wird der Nachfolger des aktuellen Buffer Blocks instanziiert, falls möglich. Bei Ende des Blocks wird geprüft, ob der Nachfolger existiert. Falls ja, wird der aktueller Buffer Block auf ihn geschaltet.

Diese Logik hat man extra dafür gemacht, damit es kein zusätzlicher IO-Aufwand bei mehreren *ungetChar(unsigned int)* Aufrufe am Anfang des Folgeblocks entsteht.

Der *FileReader* bietet die *getBufferBlockAt(int index)* an. Damit kann man direkt mit den gewünschten Block anfangen (wird aber nicht benutzt). Der *FileReader* beschäftigt sich mit dem Erzeugen eines korrekten Blocks anhand des *std::ifstream* -s. Genau hier wird ermittelt, ob der zu kreierender Block der letzte ist. Diese Eigenschaft wird benutzt bei *BLOCK\_SIZE - MAX\_STEPBACK.*

Der *BufferBlock* außer den schon erwähnten Eigenschaften bietet zusätzlich die *getCharAt(unsigned int index)* Aufruf an. Der Index darf die Länge des Blocks nicht überschreiten. Die Länge wird beim Kreieren mitgeteilt. Bei allen Blöcken, außer den letzten ist diese gleich *BLOCK\_SIZE*.

# Die SymbolTable

Die Aufgabe der Symbol Tabele ist die Infos über den jeglichen Identifier Tokens schnell abzurufen. Die Infohaltende Klasse ist die *Symbol* Klasse.

**Arbeitsweise**

Der Scanner ruft die *create(String\*, TokenType)* Methode jedes mal auf, wenn der Automat ein Identifier erkennt. Mit diesem Aufruf prüft die *SymbolTable*, ob die gegebene Lexem in der Tabelle schon vorhanden ist. Falls nicht, wird ein neues Symbol mit dem gegebenen *TokenTyp* kreiert und eingetragen. Der *create(String\*, TokenType)* Aufruf liefert den Symbol zurück, falls der schon in der Tabelle existiert.

Das wesentliche an der Symbol Tabelle ist ihre Geschwindigkeit. Diese wird mit dem Einsetzen vom Hashing erreicht. Initial hat die Tabelle 64 Einträge. Mittels der *strhash(String)* Methode wird die Position des zu Lexem gehörigen Symbol in der Tabelle bestimmt. Es ist natürlich möglich, dass mehrere Symbole unter dem gleichen Hash Index zu auffinden sind. Aus diesem Grund, liegt in der Tabelle unter jedem Hash Index eine Liste, die mehrere Lexem beinhaltende Symbole hat. Der *indexOf(String)* liefert den Index des Symbols innerhalb der inneren Liste.

Wie bereits erwähnt, wird es beim Einfügen eines neuen Symbols geprüft, ob die Tabelle den Eintrag hat. Dafür wird erst mal nachgeschaut, ob eine Liste unter dem berechneten Hash Index existiert. Wenn solche schon vorhanden ist, wird jedes Lexem innerhalb des jeweiligen Symbols mit dem gegebenen Lexem verglichen. Ist das nicht der Fall, wird der Symbol an Ende der Liste angehängt.

Es kann sein, dass 64 Einträge nicht genug sind um alle *Identifiers* zu aufbewahren. Die Untersuchungen zeigen allerdings, dass schon nach 50% Auffüllung der Tabelle sinkt die Performanz. Aus diesen Gründen wird beim Einfügen eines Symbols geprüft, ob die *elemCount* schon größer als Hälfte der Tabellengröße ist. Falls ja, wird die Tabelle verdoppelt. Dabei werden alle Symbolen neu ge-hashed und schon anders in die Tabelle platziert.