Scanner Dokumentation

# Einführung

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.

Der Scanner beschäftigt sich mit der lexikalische Analyse. Grobbeschrieben, zerlegt der Scanner den gewünschten Quelltext in die bestimmte Teile und erzeugt eine Zwischendarstellung, die nachher von dem Parser gebraucht wird um den Code zu generieren.

Der Scanner besteht aus vier Modulen: der Automat, die Symboltabelle, der Puffer und der Scanner selbst. Der Scanner mit seiner Arbeitsweiße führt die lexikalische Analyse durch. Somit gewinnt man die schon erwähnte Zwischendarstellung, die die semantische Auskünfte über den Quelltext beinhält.

Bevor man über solche Auskünfte redet, muss noch verdeutlicht welcher Quelltext ist zu erkennen. Eine fundamentale Eigenschaft eines Texts ist seine Zugehörigkeit zu einer bestimmten Sprache. Der Text gehört zu der Sprache, wenn alle seine Worte mit der Grammatik der Sprache beschrieben werden können. Der Scanner muss den Text auf Zugehörigkeit zu der folgenden Sprache überprüfen: 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

Die lexikalische Analyse lässt sich mit dem Auffinden den Lexems beschreiben. Unter einer Lexem ist eine Ausprägung eines Tokens zu verstehen. Der Token, seinerseits, ist ein e semantische Einheit der Sprache. Die Liste von Tokens ist oben zu finden.

# Der Automat

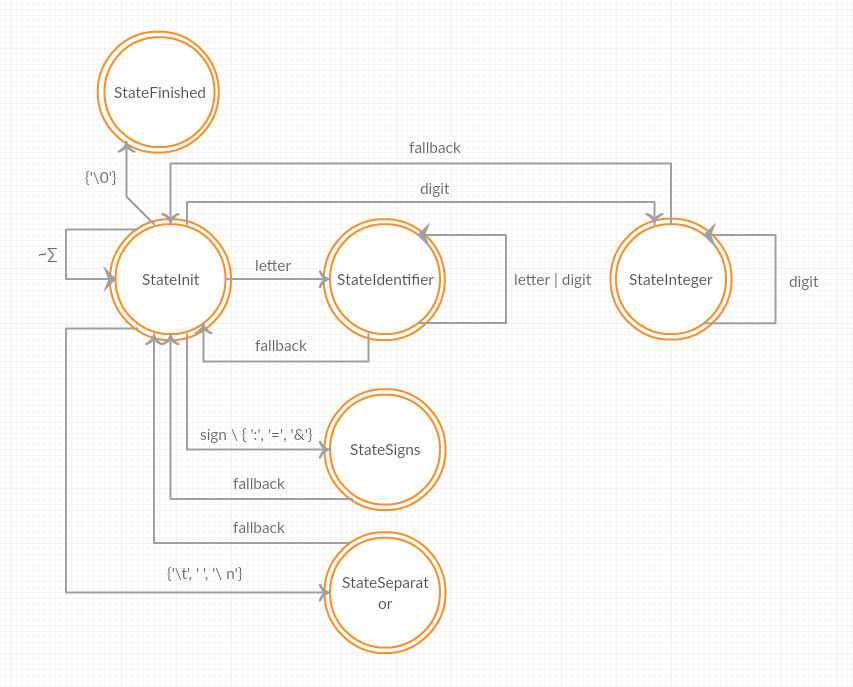
**Theoretisches**

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, char\* lexem)

Weiterhin, hat der Automat ein Zeiger auf den aktuellen Zustand, sowie ein Zeiger auf den letzten 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.

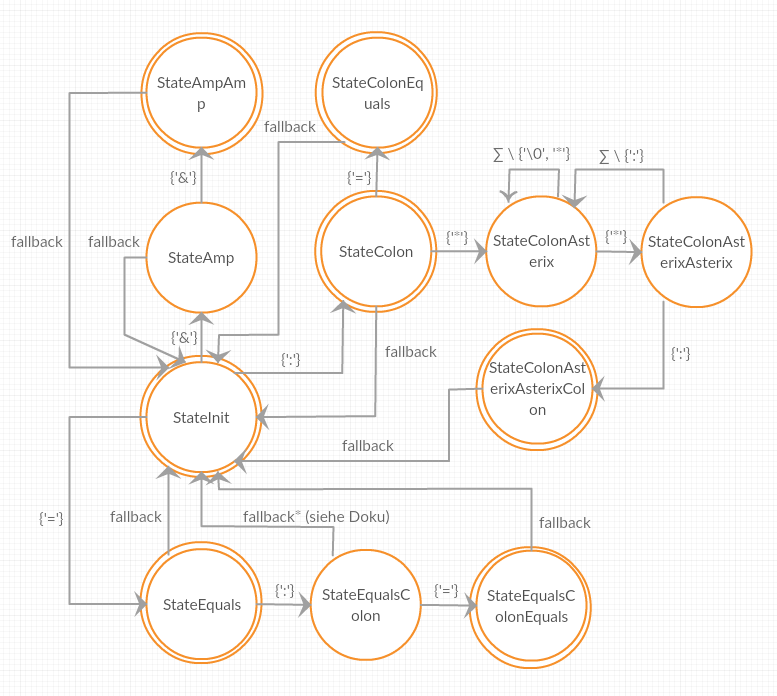


Zum Anfang befindet sich der Automat im StateInit. Das ist der zentrale Zustand, zu dem immer wieder zurückgekehrt wird. Der allgemeiner Tokenakzeptanz-Vorgang sieht folgender maßen aus:

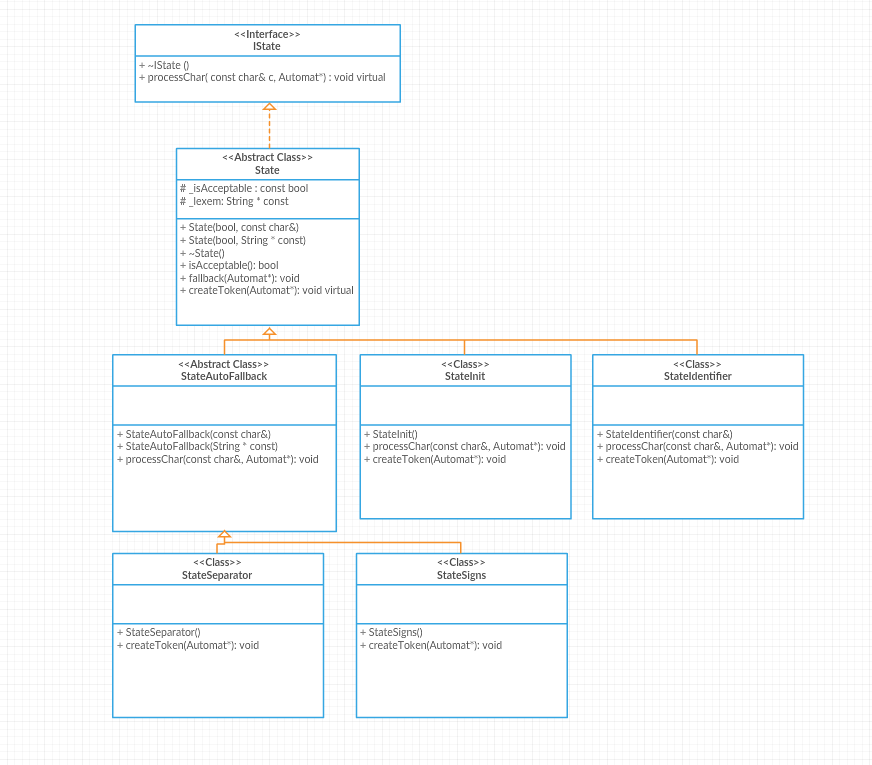
1. Der Automat geht von StateInit aus.
2. Der Automat bearbeitet die Zeichen eindeutig abhängig von den aktuellen Zustand. Diese kann zu folgenden Abläufen führen:
   1. Dem Zustandswechsel
   2. Im Zustand bleiben
   3. Nach StateInit zurückkehren.
3. Der Automat stoppt, wenn er '\0' Zeichen liest.

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

* Ist man in einem gültigen Zustand, so wird der entsprechende Token erzeugt. Danach meldet sich der Automat bei dem Scanner, dass das aktuelle Zeichen von StateInit neu bearbeitet werden muss. Schließlich, wird nach StateInit gesprungen. Diesen Prozess haben wir mit 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 Automaten sind objektorientiert entworfen. Zugrunde liegt der IState Interface, der von eine abstrakte Klasse State implementiert wird. 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 werden muss. Die einfache Klassen wie StateSigns, StateSeparator erben von StateAutoFallback. Diese Klasse sorgt dafür, dass ihre Kinder über einfache Schaltungslogik verfügen, die Fallback heißt.

Das Projekt wird mittels make-File kompiliert und gelinkt. Die Regeln waren entsprechend angepasst. Das Testen erfolgt im separaten File. Für diese Zwecken wurde ein StubScanner angelegt, der den IScanner Interface implementiert.

**Probleme**

Das Hauptproblem, das beim Implementieren des Automaten entstanden ist, lag am Verstehen eines Interfaces in C++. Dabei war der angestellten Person unklar, wie das Zusammenspiel eines Destruktors des Interfaces 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 uns von den zyklischen Abhängigkeiten geschützt, jedoch gab es noch ein Paar stellen, wo die Klassen zu vordeklarieren waren.