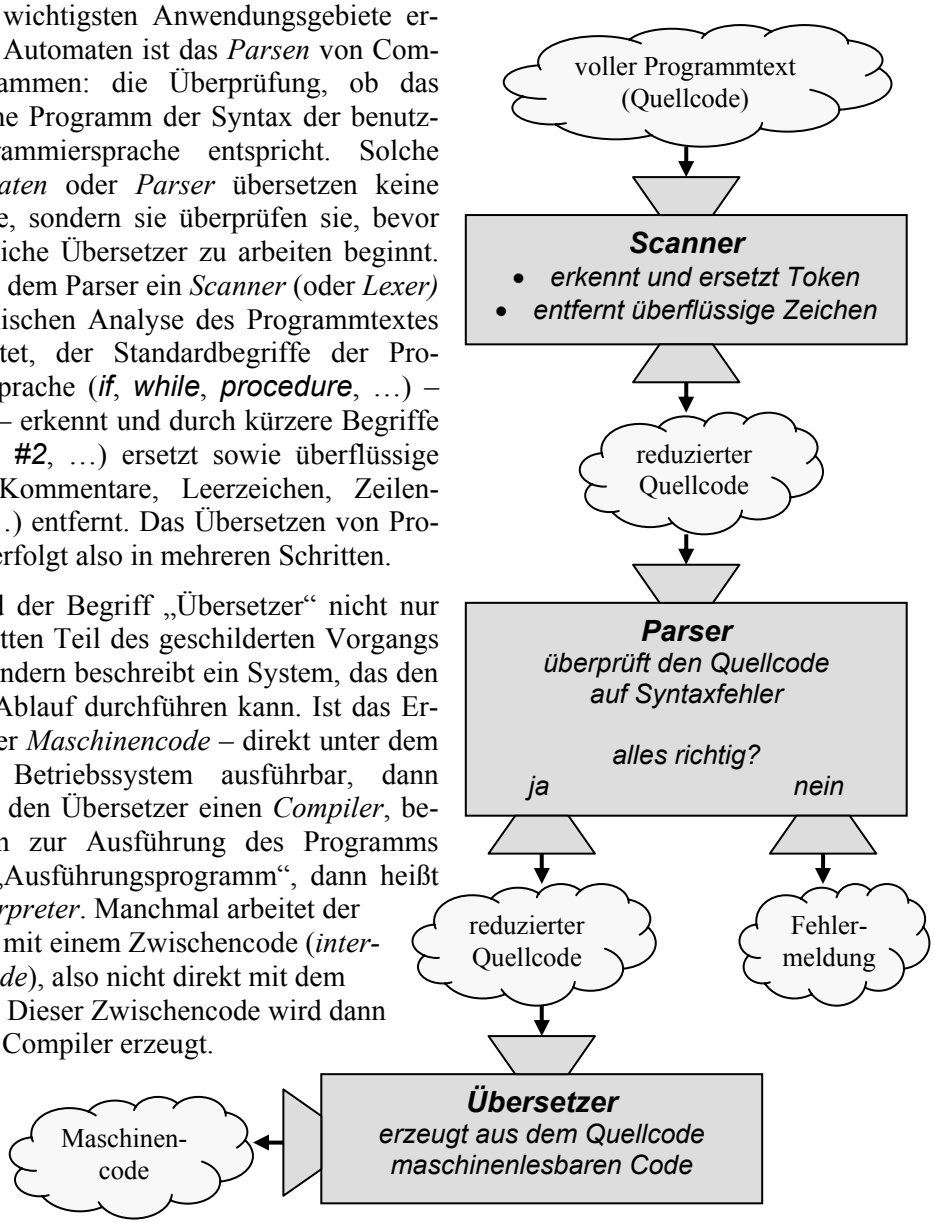
4 Syntaxanalyse mit erkennenden Automaten

4.1 Das Übersetzen von Programmen: Parser

Eines der wichtigsten Anwendungsgebiete erkennender Automaten ist das *Parsen* von Computerprogrammen: die Überprüfung, ob das eingegebene Programm der Syntax der benutzten Programmiersprache entspricht. Solche *Prüfautomaten* oder *Parser* übersetzen keine Programme, sondern sie überprüfen sie, bevor der eigentliche Übersetzer zu arbeiten beginnt. Meist wird dem Parser ein *Scanner* (oder *Lexer)* zur lexikalischen Analyse des Programmtextes vorgeschaltet, der Standardbegriffe der Programmiersprache (*if*, *while*, *procedure*, …) – die *Token* – erkennt und durch kürzere Begriffe (z. B. *#1, #2*, …) ersetzt sowie überflüssige Zeichen (Kommentare, Leerzeichen, Zeilenwechsel, …) entfernt. Das Übersetzen von Programmen erfolgt also in mehreren Schritten.



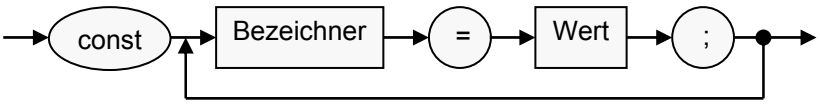
Meist wird der Begriff „Übersetzer“ nicht nur für den dritten Teil des geschilderten Vorgangs benutzt, sondern beschreibt ein System, das den gesamten Ablauf durchführen kann. Ist das Ergebnis – der *Maschinencode* – direkt unter dem benutzten Betriebssystem ausführbar, dann nennt man den Übersetzer einen *Compiler*, benötigt man zur Ausführung des Programms noch ein „Ausführungsprogramm“, dann heißt dieses *Interpreter*. Manchmal arbeitet der Interpreter mit einem Zwischencode (*intermediate code*), also nicht direkt mit dem Quellcode. Dieser Zwischencode wird dann von einem Compiler erzeugt.

4.2 Die Analyse von Konstantenvereinbarungen

Die Zeichen, die z. B. ein Delphi-Programm bilden, können wir als eine einzige Eingabezeichenfolge auffassen. Übergeben wir diese einem Automaten – einem Parser –, dann sollte dieser *erkennen*, ob es sich um ein syntaktisch korrektes Programm handelt – oder nicht. Er sollte ggf. auch helfen zu erkennen, wo sich die aufgetretenen Fehler befinden und um welche Fehler es sich handelt – wenigstens beim ersten. Grosse Teile der Programmiersprachenkonstrukte lassen sich auf diese Weise schon von endlichen Automaten analysieren. Sie werden von diesen *geparst*.

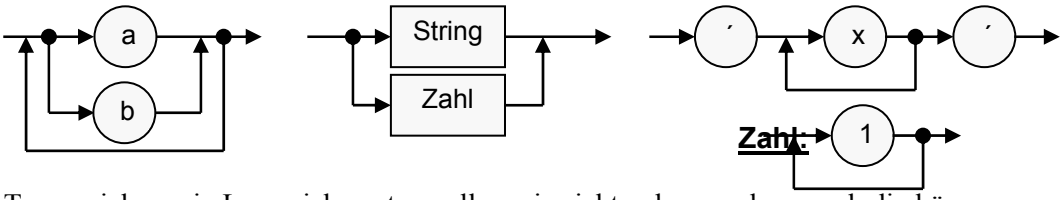
Als Beispiel wollen wir etwas vereinfachte Konstantenvereinbarungen von Delphi parsen.

Konstantenvereinbarung:



Da wir hier modellhaft arbeiten wollen, vereinfachen und verkürzen wir die üblichen Bezeichner etwas. Wir sparen dadurch erheblich Platz, ohne das Problem selbst wesentlich zu verändern. Das Token *const* verkürzen wir zu einem einzigen *c*. Das ist keine große Einschränkung, denn diese Reduktion kann durch den vorgeschalteten Scanner geschehen! Bezeichner sollen nur aus *a*´s und *b*´s bestehen und als Werte lassen wir Zahlen aus lauter Einsen sowie Zeichenketten nur aus *x*-en zu.

Bezeichner: **Wert:** **String:**



Trennzeichen wie Leerzeichen etc. wollen wir nicht zulassen, denn auch die können vom Scanner leicht entfernt werden.

Gültige Konstantenvereinbarungen wären z. B.:

```
ca=1;
ca=1;b=11;ab='xxxx';
```

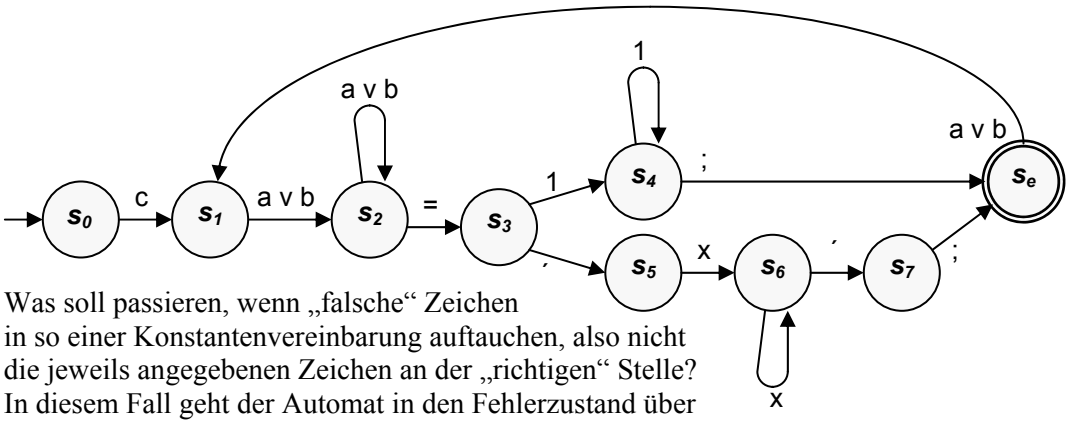
Ungültige wären:

```
ca=1          (Semikolon fehlt)
cabc='xxxxx';  (falscher Bezeichner)
```

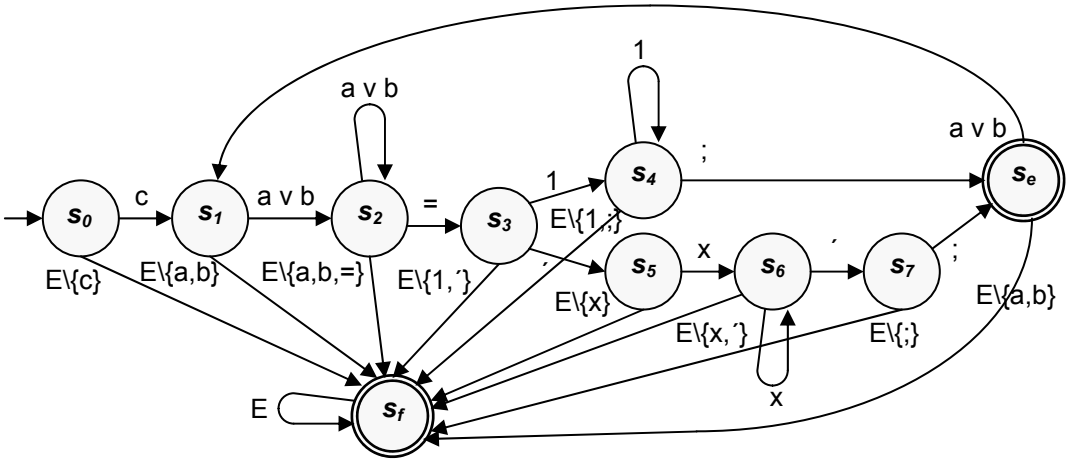
Wir wollen jetzt einen Parser für solche Konstantenvereinbarungen als erkennenden Automaten entwickeln. Dafür ist das übliche Vorgehen angebracht: Wir berücksichtigen zuerst nur die „interessanten“ Zeichenfolgen. Dafür können wir das angegebene Syntaxdiagramm fast abschreiben. Zuerst aber müssen wir das Eingabealphabet bestimmen. Dazu sammeln wir alle Zeichen, die irgendwo in den angegebenen Syntaxdiagrammen vorkommen.

Eingabealphabet: E = { c , a , b , = , 1 , ´ , x , ; }

Ein Ausgabealphabet existiert für einen erkennenden Automaten natürlich nicht, und die Zustandsmenge ergibt sich aus dem Transitionsgraphen. Dessen erster Teil lautet:

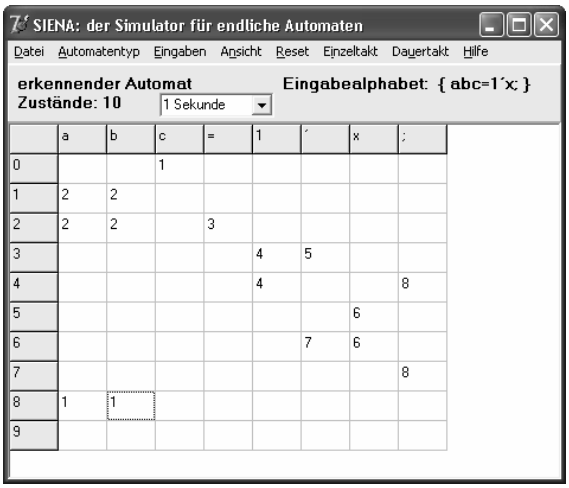


Was soll passieren, wenn „falsche“ Zeichen in so einer Konstantenvereinbarung auftauchen, also nicht die jeweils angegebenen Zeichen an der „richtigen“ Stelle? In diesem Fall geht der Automat in den Fehlerzustand über – und bleibt dort auch. Da meist alle bis auf sehr wenige Zeichen des Eingabealphabets an einer bestimmten Stelle der Zeichenkette falsch sind, wählen wir wieder die bewährte Methode, mit E \ {…} alle bis auf die in den Mengenklammern aufgeführten Zeichen zu bezeichnen, mit E \ {a,b} z. B. alle Zeichen außer *a* und *b*.



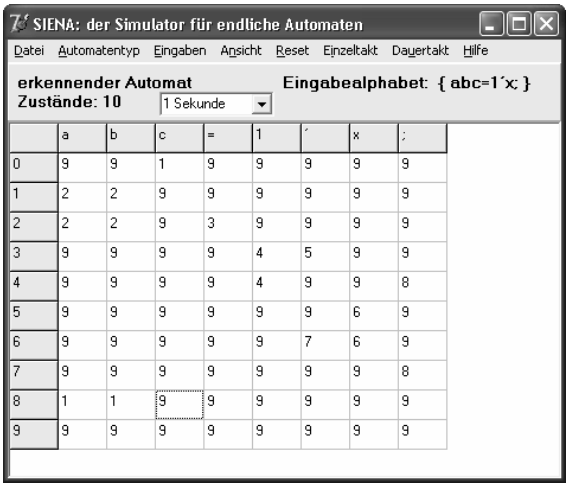
Da wir jetzt den Automaten gefunden haben, können wir ihn auch als Programm realisieren. Mit der Beschreibung des Konstantenvereinbarungs-Parsers haben wir gleichzeitig eine fertige Lösung.

Zum Test des Automaten geben wir die beschreibenden Größen in *SIENA* ein und probieren, ob die Maschine richtig arbeitet. Da die Zustände *0* bis *7* für die „internen“ Übergänge benutzt werden soll der Zustand *8* der Endzustand und der Zustand *9* der Fehlerzustand sein. Am einfachsten ist es, zuerst die Automatentafel für die „richtigen“ Übergänge einzugeben.



	a	b	c	=	1	´	x	;
0			1					
1	2	2						
2	2	2		3				
3					4	5		
4					4			8
5							6	
6						7	6	
7								8
8	1	1						
9								

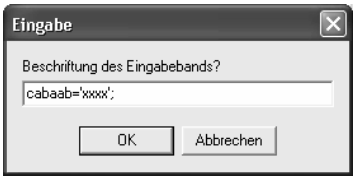
Danach füllen wir die Lücken aus mit Übergängen in den Fehlerzustand.



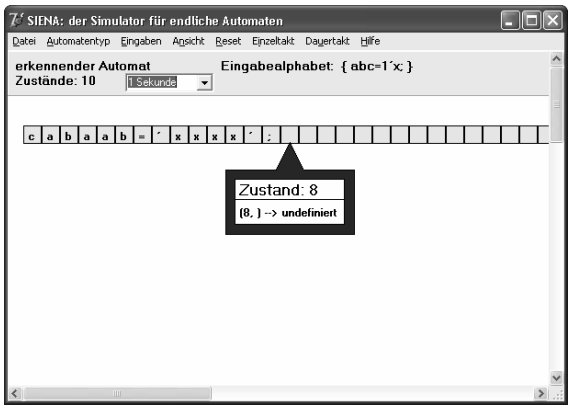
	a	b	c	=	1	´	x	;
0	9	9	1	9	9	9	9	9
1	2	2	9	9	9	9	9	9
2	2	2	9	3	9	9	9	9
3	9	9	9	9	4	5	9	9
4	9	9	9	9	4	9	9	8
5	9	9	9	9	9	9	6	9
6	9	9	9	9	9	7	6	9
7	9	9	9	9	9	9	9	8
8	1	1	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9

Fertig! Der Test kann beginnen.

Wir geben eine richtige Konstanten-Vereinbarung ein. Diese sollte den Automaten im Endzustand (8) hinterlassen.



Tut sie auch!


SIENA: der Simulator für endliche Automaten
Datei Automatentyp Eingaben Ansicht Reset Einzeltakt Dauertakt Hilfe
erkennender Automat
Zustände: 10
1 Sekunde
Eingabealphabet: { abc=1´x; }
c a b a a b = ´ x x x ´ ;
Zustand: 8
(8,) --> undefiniert