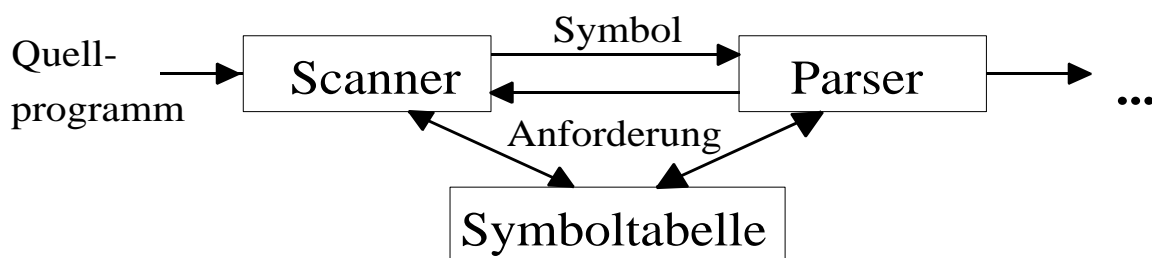


3 Lexikalische Analyse

3.1 Die Rolle des Scanners



Aufgabenbereiche

- Eingabestrom lesen und als Ausgabe eine Folge von Symbolen (Tokens) erzeugen, die der *Parser* syntaktisch analysiert, eventuell Ausgabe von Fehlermeldungen.
- Entfernung von Kommentaren, Leerzeichen, Tabulatoren, Zeilenwechsell.
- Erzeugung des Compiler-Listings.
- Einordnen von Fehlermeldungen des Compilers in das Listing.
- Übernahme von Funktionen eines Präprozessors, wenn ein solcher fehlt.

Motivation für Trennung von Parser und Scanner

- Der Teil der Grammatik, der zur Definition der Symbole dient, besitzt reguläre Produktionen, während der Rest kontextfreie Produktionen besitzt. Es liegt also nahe, die einfacheren Produktionen durch einen endlichen Automaten erkennen zu lassen, während der (nicht-reguläre) Rest durch eine komplizierteren und aufwendigeren Stackautomaten erkannt werden muß.
- Dies dient der Vereinfachung beider Teile des Compilers, damit der Effizienz
- Erhöhung der Portabilität, Modularisierung

Symbole, Muster und Lexeme

Diese Begriffe tauchen häufig im Zusammenhang mit der lexikalischen Analyse auf.

Lexem: Zeichenfolge im Quellprogramm

Muster: Regel zur Erzeugung von Lexemen, die ein Symbol repräsentieren

In der folgenden Tabelle werden die Zusammenhänge klar. Es wird allerdings keine bestimmte Programmiersprache betrachtet.

| Symbol | exemplarisches Lexem | informelle Musterbeschreibung | formale Musterbeschreibung |
|------------|----------------------|-----------------------------------------------|----------------------------|
| CONST | const | CONST, const, ... | |
| IF | if | IF, if, ... | |
| RELATION | <, <=, =, >=, >, <> | < oder <= oder ... | |
| IDENTIFIER | pi, zahl, x5 | Buchstabe, gefolgt von Buchstaben und Ziffern | [a-z][a-z0-9]* |
| LITERAL | 'Compiler' | Zeichenfolge zwischen '' ohne '' | |
| NUMBER | 3.14159, 0, -1.6E-5 | irgendeine numerische Konstante | [0-9]+ |

Aufwand der lexikalischen Analyse

Dies hängt sehr stark von den Konventionen der betreffenden Programmiersprache ab!

1. Große Unterschiede in der Behandlung von *Leerzeichen*.

Leerzeichen können nach Belieben zur besseren Lesbarkeit eingefügt werden.

Bsp.:

DO 5 I = 1.25 DO5I=1.25

Ein Scanner für FORTRAN braucht erheblich mehr Look-ahead als ein Pascal-Scanner.

2. *Schlüsselwörter und reservierte Wörter*

Auch hier gibt es unterschiedliche Konventionen, z.B. sind in PL/1 Schlüsselwörter *nicht* reserviert. Der Scanner muß dann entscheiden, ob es sich um ein Schlüsselwort oder einen benutzerdefinierten Namen handelt.

Bsp.:

```

IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
|   |   |   |   |   |   |   |   |
S   N   S   N   N   S   N   N

```

S steht für Schlüsselwort, N für Name.

Fehlerbehandlung

Da ein Scanner nur eine sehr beschränkte Sicht eines Quellprogramms hat, kann er nur wenige Fehler identifizieren.

Bsp.:

fi (a == f(x)) ...

Hier kann der Scanner nicht erkennen, daß if falsch geschrieben wurde, sondern interpretiert fi als Namen!

3.2 Implementationsmöglichkeiten

- Unterprogramm der Syntaxanalyse (der Scanner stellt dem Parser auf Anforderung neue Symbole zur Verfügung).
- Selbstständiger Compilerpaß, sequentieller Ablauf .
- Selbstständiger Compilerpaß, konkurrenter Ablauf (Erzeuger-Verbraucher-Problem).

Generierung von Scannern

1. von Hand

Entwurf eines Diagramms, das die Struktur der Symbole der Quellsprache beschreibt. Übersetzung des Diagramms von Hand in ein Programm, das Symbole erkennt. So kann man effiziente Scanner erstellen.

2. durch Software-Werkzeuge

Es gibt *Scanner-Generatoren*, die keine besonderen Vorkenntnisse verlangen.

Eingabe: Beschreibung der Symbole der Programmiersprache.
Beschreibung der zu treffenden Aktionen.

Ausgabe: Tabelle, die von einem syntax-unabhängigen Analyseprogramm für die lexikalische Analyse weiterverwendet wird.

Bsp.: LEX, FLEX (verfügbar unter UNIX, DOS, NT, OS/2, VMS für Sprachen wie C, C++, Pascal), ALEX (in Modula-2 geschrieben, auch für MS-DOS erhältlich), COCO u.a.m.

3.3 Lexikalische Analyse von PL/0 als Beispiel

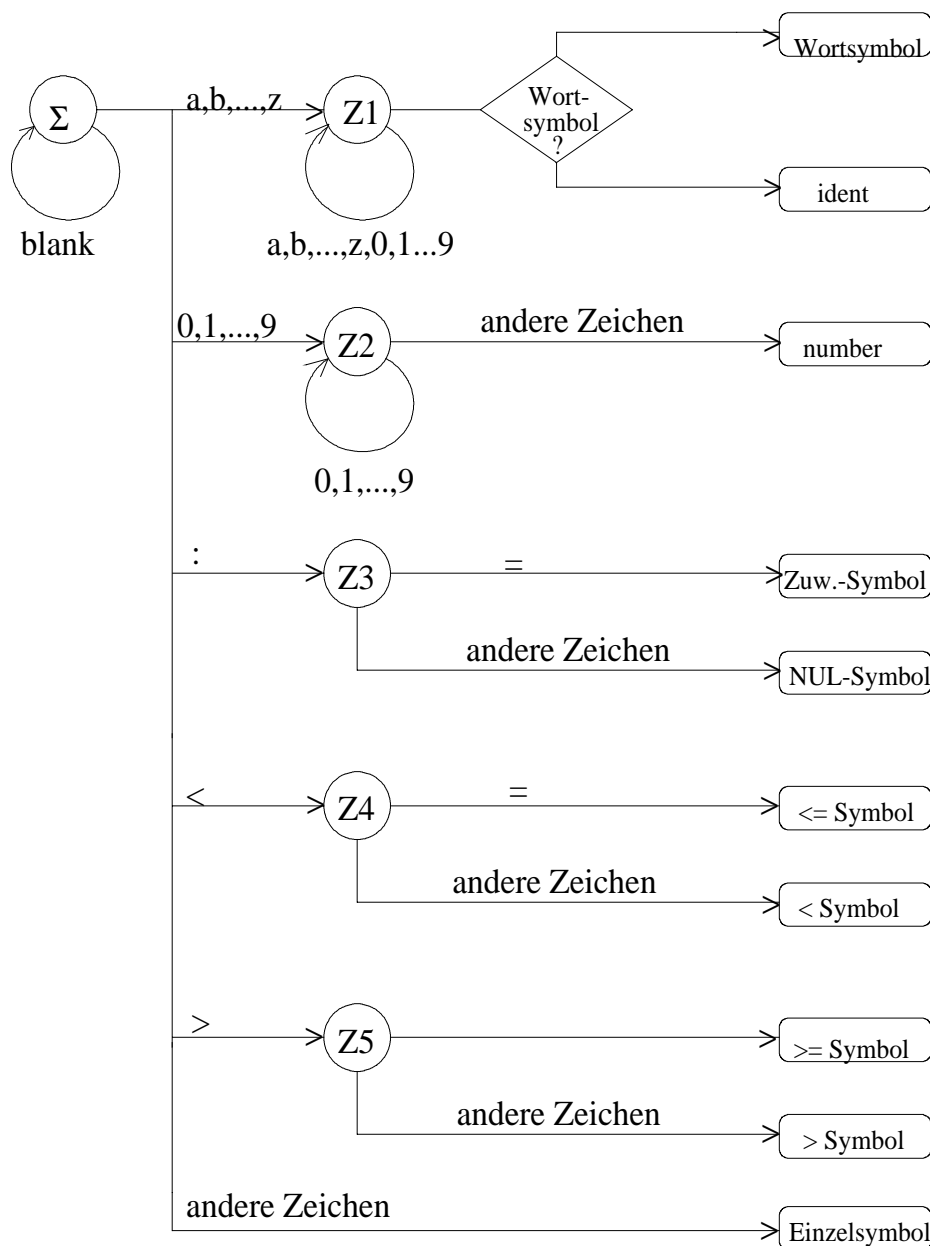
In unserem in C geschriebenen PL/0-Compiler ist die lexikalische Analyse als eigener Paß implementiert.

- Leistungen:**
- Einlesen des Quellprogramms, das zeilenweise zwischengepuffert wird.
 - Erkennen der Symbole von PL/0.
 - Umrechnung von Zahlenkonstanten in die interne Darstellung.
 - Erstellung des Compiler-Listings.

Der Scanner gibt den Typ des erkannten Symbols in einem Zwischenfile aus. Bei Namen und Zahlenkonstanten werden zusätzlich Name und Wert abgeliefert.

Das folgende Diagramm stellt den Scanner durch das Zustandübergangsdiagramm eines *endlichen Automaten* dar.

Ein Listing des PL/0-Scanners ist im *Anhang* zu finden.



Das kleine PL/0-Programm test.pl0

```

const n=10000;
var s, i, j;

begin
  write('Test');
  writeln;
  i:= 0;
  while i < n do
  begin
    i:= i + 1;
    write(i);
    writeln;
    s:= 0;
    j:= 0;
    while j < n do
    begin
      j:= j + 1;
      s:= s + j;
    end;
  end;
end.

```

wurde durch den Aufruf

```
pl0scan test
```

gescant und ergab eine Token-Datei test.tok mit Inhalt

| | | | | | |
|----|-------|----|---|----|---|
| 27 | | 1 | n | 1 | n |
| 1 | n | 25 | | 25 | |
| 8 | | 20 | | 20 | |
| 2 | 10000 | 1 | i | 1 | j |
| 17 | | 19 | | 19 | |
| 28 | | 1 | i | 1 | j |
| 1 | s | 3 | | 3 | |
| 16 | | 2 | 1 | 2 | 1 |
| 1 | i | 17 | | 17 | |
| 16 | | 31 | | 1 | s |
| 1 | j | 14 | | 19 | |
| 17 | | 1 | i | 1 | s |
| 20 | | 15 | | 3 | |
| 31 | | 17 | | 1 | j |
| 14 | | 32 | | 17 | |
| 33 | Test | 17 | | 21 | |
| 15 | | 1 | s | 17 | |
| 17 | | 19 | | 21 | |
| 32 | | 2 | 0 | 17 | |
| 17 | | 17 | | 21 | |
| 1 | i | 1 | j | 18 | |
| 19 | | 19 | | | |
| 2 | 0 | 2 | 0 | | |
| 17 | | 17 | | | |
| 24 | | 24 | | | |
| 1 | i | 1 | j | | |
| 10 | | 10 | | | |