Compilerbau

Lexer

Prof. Dr. Franz-Karl Schmatzer
schmatzf@dhbw-loerrach.de

Literatur

- C.Wagenknecht, M.Hielscher; Formale Sprachen, abstrakte Automaten und Compiler; 3.Aufl. Springer Vieweg 2022;
- U.Meyer; Grundkurs Compilerbau; Rheinwerkverlag, 1. Aufl. 2021
- A.V.Aho, M.S.Lam,R.Savi,J.D.Ullman, *Compiler – Prinzipien, Techniken und Werkzeuge*. 2. Aufl., Pearson Studium, 2008.
- Güting, Erwin; *Übersetzerbau –Techniken, Werkzeuge, Anwendungen*, Springer Verlag 1999

Grundlegende Konzepte

- Rolle eines Lexers
- Token, Muster, Lexeme
- Tokenbeschreibung
  - Reguläre Ausdrücke
  - Zustandsautomaten
- Eigener Lexer

Rolle eines Lexers

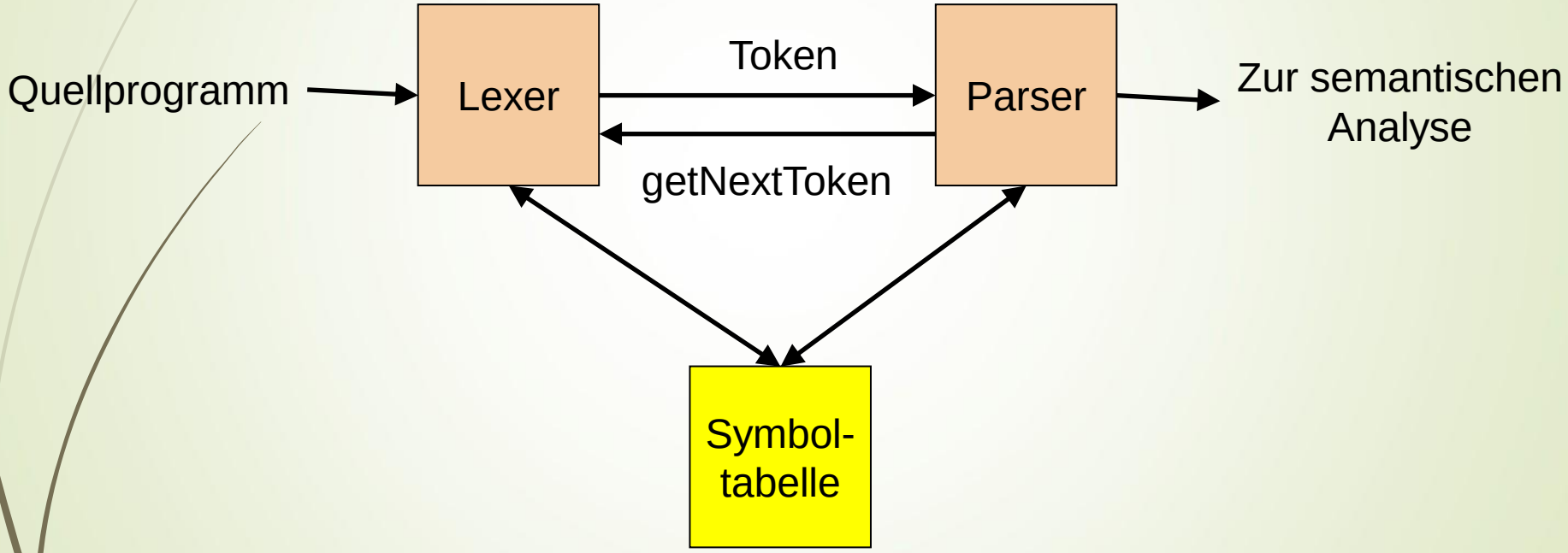
Einführung

- Die erste Phase eines Compilers ist der Lexer
- Die Hauptaufgabe ist
  - Die Zeichen des Quellprogramms zu lesen,
  - in Lexeme zu gruppieren und
  - als Folge von Tokens auszugeben
- Außerdem der Aufbau und das Interagieren mit einer Symboltabelle

Rolle eines Lexers

Interaktion mit dem Parser

- Interaktion zwischen dem Lexer und dem Parser



Token, Muster, Lexeme

Definition

- Token
  - Ein Paar aus Namen und optionalen Attributwert
  - Name ist ein abstraktes Symbol und wird vom Parser verarbeitet
- Muster
  - Beschreibung der Form, welches ein Lexem einnehmen kann.
  - Ist das Token ein Schlüsselwort, ist das Muster einfach diese Folge von Zeichen.
  - Bei Bezeichnern und andere Token kann die Struktur komplexer sein.
- Lexem
  - Eine Zeichenfolge, welches ein Muster für ein Token ist und vom Lexer als Instanz des Tokens erkannt wurde

Token, Muster, Lexeme

Beispiel

printf(“Total = %d\n“, score);

- printf und score sind Lexeme, die dem Muster für das Token id entsprechen
- das Lexem “Total = %d\n“ ist ein Literal.

- Beispiel für Token

Token	Beschreibung	Beispiel
if	Zeichen i, f	if
else	Zeichen e, l, s, e	else
comp	< oder > oder <= oder >= oder == oder !=	<=,!=
id	Buchstabe, auf den Buchstaben oder Ziffern folgen	pi, score, D2
number	Alle numerische Konstanten	3.123, 0, 6.0E3
literal	Alles was in Ausführungszeichen steckt	“core dumped“

Token, Muster, Lexeme

Attribute

- Stimmen mehrere Lexeme mit einem Muster überein, muss man dem Parser noch zusätzliche Information mitgeben (Attributwerte)
- Z.B kann das Token number sowohl 0 oder 1.0 oder … als Attributwert haben.
- Beispiel für eine Codezeile:

E = 2*M**5

wird übersetzt in folgende Tokenpaare (Tokenname, Attributwert)

<id, Zeiger auf dem Symboleintrag für E>

<=>

<number, Integerwert 2)

<*>

<id, Zeiger auf dem Symboleintrag für M>

<**>

<number, Integerwert 5)

Theoretischer Hintergrund der lexikalischen Analyse

- Die Struktur lexikalischer Symbole kann durch reguläre Ausdrücke beschrieben werden, die zu einer regulären Sprache gehören.
- Reguläre Sprachen werden durch rechtslineare (linkslineare) Grammatiken erzeugt.
- Diese werden durch nichtdeterministische endliche Automaten NEA erkannt
- Zu jedem NEA gibt es ein DEA, den man kanonisch ableiten und dann implementieren kann.

Reguläre Sprachen und Ausdrücke

- Reguläre Sprachen
  - $\varnothing$ und $\{\varepsilon\}$ sind reguläre Sprachen.
  - für jedes $a \in \Sigma$ ist $\{a\}$ eine reguläre Sprache.
  - Sind R und S reguläre Sprachen, dann sind auch $R \cup S$, RS und R^* reguläre Sprachen.
- Reguläre Ausdrücke werden induktiv definiert:
  - $\varnothing$ und ε sind reguläre Ausdrücke.
  - für jedes $a \in \Sigma$ ist $\{a\}$ ein regulärer Ausdruck.
  - Sind r und s reguläre Ausrücke, dann ist auch
    - $(r \mid s)$ ein regulärer Ausdruck
    - rs ist ein regulärer Ausdruck
    - r^* ist ein regulärer Ausdruck

Reguläre Sprache
Definition

- Um die Spezifikation von regulären Ausdrücken bequemer zu machen, führt man den Begriff der regulären Definition ein.

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$d_3 \rightarrow r_3$$
$$\ldots$$
$$d_n \rightarrow r_n$$

$\mathbf{d_i}$ ist ein Name für einen regulären Ausdruck (reguläre Symbole).

In $\mathbf{r_i}$ dürfen nur die Namen von $\mathbf{d_1}$ bis $\mathbf{d_{i-1}}$ vorkommen um Rekursion zu vermeiden.

Reguläre Sprache

Beispiel

LETTER → A | B | … | Z | a | b | … | z

DIGIT → 0 | 1 | … | 9 |

ID → LETTER (LETTER | DIGIT)*

SIGN → + | -

INT → (SIGN | ε) DIGIT DIGIT*

- Nützliche Erweiterungen:
  - **+:** unärer Postfixoperator und steht für ein oder mehrmaliges Auftreten des vorangehende Symbols
  - **?:** unärer Postfixoperator und steht für kein oder einmaliges Auftreten des vorangehende Symbols
  - Zeichen **[abc]** := a | b | c
  - Zeichenklassen **[a-c]** := a | b | c

Reguläre Sprache
Beispiel

- Damit lässt sich das Beispiel von vorher kürzer fassen

  LETTER → [A – Z a - z]

  DIGIT → [0 - 9]

  SIGN → [+ -]

  ID → LETTER (LETTER | DIGIT)*

  INT → SIGN? DIGIT+

- Erstellen Sie mithilfe von FLACI eine Grammatik und testen Sie diese, ob Sie ID und INT ableiten kann.

Lösung Grammatik

S -> ID | INT

ID -> LETTER IDH

IDH -> LETTER IDH | DIGIT IDH | EPSILON

INT -> SIGN INTH | INTH

INTH -> DIGIT INTH | DIGIT

DIGIT -> 0 ... 9

SIGN -> + | -

LETTER -> a ... z | A ... Z

Reguläre Ausdrücke

- Reguläre Ausdrücke werden eingesetzt für
  - Suchen
  - Ersetzen
  - Filtern
  - Verifikation

  Von Zeichenketten
- Grammatiken einfacher schreiben
- Jedoch sind die zu verwendeten Metazeichen nicht genormt. Je nach System gibt es teilweise andere Notationen.

Tabelle für Metazeichen I

Metazeichen	Bedeutung
.	Steht für ein beliebiges Zeichen (außer \n und \r
a*	Steht für 0 bis beliebig viele Regeln
a+	Steht für 1 bis beliebig viele Regeln
a?	Steht für ein optionales Zeichen
{m,n}	Steht für m bis n Vorkommen des Zeichens oder Metazeichen
{n}	Steht für n Vorkommen des Zeichens oder Metazeichen
(ab)	Klammern erlauben Teilausdrücke zu einer Einheit zusammenzufassen um anschließend Operationen wie +,* oder | für den gesamten Teilausdruck anzuwenden
A | B	Steht für Teilausdruck A oder B
[ABC]	Eckige Klammern beschreiben eine Auswahl für Zeichen
[a-zA-Z]	Der Bindestrich innerhalb der eckigen Klammern bestimmen einen Bereich. In diesem Beispiel für genau ein Zeichen von a bis z oder A bis Z.
[^A]	Das Dach ^ negiert den Ausdruck. Hier beliebiges Zeichen außer A.

Tabelle für Metazeichen II

Metazeichen	Bedeutung
\n	Ein Zeilenumbruch
\r	Ein Wagenrücklauf (unter Windows \r\n für einen Zeilenumbruch)
\t	Ein Tabulatorzeichen
\s	Ein einzelnes Leerzeichen
\d	Der reguläre Teilausdruck [0-9]
\w	Alphanumerische Zeichen: ein Buchstabe, eine Ziffer oder der Unterstrich [a-zA-Z_0-9]

Beispiele für reguläre Ausdrücke

Metazeichen	Bedeutung
…	Trifft 3 beliebige Zeichen (außer \r und \n)
.{1,5}	Trift 1 bis 5 beliebe Zeichen (außer \r und \n)
[0-9]+	Trift eine beliebige Ziffernfolge mit mindestens einem Zeichen.
a*b	Trift beliebig viele a gefolgt von einem b
a* | b*	Trift beliebig viele a oder beliebig viele b
\d+,\d\d	Trifft eine Fließkommazahl mit beliebig viele Ziffern und 2 Nachkommastellen. [0-9]+,[0-9][0-9]
[1-9][0-9]*	Trifft eine beliebig lange Zifferfolge ohne vorangestellte Null.
[\r\t\n\s]	Trifft ein typisches Whitespace
[1-5]0	Trifft genau die Ziffernfolgen 10,20,30,40 oder 50.

Aufgabe reguläre Ausdrücke

- Entwickeln Sie einen regulären Ausdruck für:
  - **Postleitzahlen:** Die alten Postleitzahl begannen mit einem Länderzeichen gefolgt von einem – und fünf Ziffern.
  - **Datumsangaben:** Die Tage und Monate können sowohl 1 oder zweistellig sein. Die Jahreszahl kann sowohl 2 oder 4 stellig sein. Die Tage, Monate und Jahr sollen durch einen . Getrennt sein.
  - **Userid:** Diese sollen mindestens 6 aber weniger als 10 alphanumerische Zeichen haben. Die ersten 3 Zeichen sollen aus Groß- oder Kleinbuchstaben bestehen. Der Rest können beliebige Alphanumerische Zeichen sein.
  - **E-Mail Adressen:** E-Mails können Zeichen, Zahlen, Punkt, Plus und Minus enthalten. Dann kommt ein @ und wieder beliebige Zeichen, Zahlen, Punkt, Plus sowie Minus. Am Schluss eine Subdomain mit 2 bis 4 Zeichen.
  - **Tastaturzeichen**, die mit Metazeichen verwechselt werden können müssen mit einem vorangestellten Escape-Zeichen \ gekennzeichnet sein. Z.B. \+ oder \- oder \. So können +,- und als Zeichen verwendet werden.
  - **Testen Sie ihre Lösung mittels FLACI**

Lösung

- Entwickeln Sie einen regulären Ausdruck für:
  - **Postleitzahlen:** Die alten Postleitzahl begannen mit einem Länderzeichen gefolgt von einem – und fünf Ziffern.

    LSG: [A-Z]{1,2}\-[0-9]{5}
  - **Datumsangaben:** Die Tage und Monate können sowohl 1 oder zweistellig sein. Die Jahreszahl kann sowohl 2 oder 4 stellig sein. Die Tage, Monate und Jahre sollen durch einen . Getrennt sein.

    LSG: \d(\d)?\.\d(\d)?\.\d\d(\d\d)?
  - **Userid:** Diese sollen mindestens 6 aber weniger als 10 alphanumerische Zeichen haben. Die ersten 3 Zeichen sollen aus Groß- oder Kleinbuchstaben bestehen. Der Rest können beliebige Alphanumerische Zeichen sein.

    LSG: [a-zA-Z]{3}(\w){3,6}
  - **E-Mail Adressen**: E-Mails können Zeichen, Zahlen, Punkt, Plus und Minus enthalten. Dann kommt ein @ und wieder beliebige Zeichen, Zahlen, Punkt, Plus sowie Minus. Am Schluss eine Subdomain mit 2 bis 4 Zeichen.

    LSG: [a-zA-Z0-9\.\+\-]+@[a-zA-Z0-9\.\+\-]+\.[a-zA-Z]{2,4}

Aufgabe Token 1

- Erstellen Sie mithilfe von FLACI folgende Grammatik, welche Zeichenfolgen und INT erkennen soll.

  ID → [A – Z a - z]+

  INT→ [1 - 9] [0 - 9]*

  Testen Sie ihre Grammatik

Aufgabe Lösung Token

- Erstellen Sie mithilfe von FLACI folgende Grammatik, welche Zeichenfolgen und INT erkennen soll.

  ID → [A – Z a - z]+

  INT→ [1 - 9] [0 - 9]*

- **Lsg:**

Start -> ID | INT

ID -> LETTER ID | LETTER

INT -> 0 | DIGIT1 INTH

INTH -> DIGIT INTH | EPSILON

LETTER -> A ... Z | a ... z

DIGIT1 -> 1 ... 9

DIGIT -> 0 ... 9

Aufgabe Token 2

- Erstellen Sie mithilfe von FLACI folgende Grammatik
  - Erweitern Sie die vorherige Grammatik durch vorzeichenbehaftete Integer und Floating Point Zahlen.
  - -1.2
  - -2.0E-4
  - -2.E+4

Aufgabe Token 2
Lösung

- Erweitern Sie die vorherige Grammatik durch vorzeichenbehaftete Integer und Floating Point Zahlen.
  - **Lsg:**

Start -> ID | INT | FLOAT

ID -> LETTER ID | LETTER

INT -> 0 | DIGIT1 INTH

INTH -> DIGIT INTH | EPSILON

FLOAT -> SIGN INT "." INTH |

SIGN INT "." INTH "E" SIGN INT

LETTER -> A ... Z | a ... z

DIGIT1 -> 1 ... 9

DIGIT -> 0 ... 9

SIGN -> + | - | EPSILON

Aufgabe Grammatik

- Entwickeln Sie Grammatik für folgende regulären Ausdrücke :

  [A-Z]{1,2}\-[0-9]{5}

  [a-zA-Z]{3}(\w){2,3}

- Testen Sie ihre Lösung mittels FLACI

Aufgabe Grammatik
Lösung

- Entwickeln Sie Grammatik für folgende regulären Ausdrücke :

  [A-Z]{1,2}\-[0-9]{5}

LSG:

Start -> LETTER ZEICHEN | LETTER LETTER ZEICHEN

ZEICHEN -> '-' DIGIT DIGIT DIGIT DIGIT DIGIT

DIGIT -> 0 ... 9

LETTER -> A ... Z

Aufgabe Grammatik
Lösung

- Entwickeln Sie Grammatik für folgende regulären Ausdrücke :

  [a-zA-Z]{3}(\w){2,3}

LSG:

Start -> LETTER LETTER LETTER WORT WORT |

LETTER LETTER LETTER WORT WORT WORT

LETTER -> a ... z | A ... Z

DIGIT -> 0 ... 9

UNDERSCORE -> '_'

WORT -> LETTER | DIGIT | UNDERSCORE

Tokenbeschreibung

Zustandsdiagramme Einleitung

- Klassische Vorgehensweise
  - Konstruktion der regulären Ausdrücke, dann
  - Konstruktion eines endlichen nicht deterministischen Automaten
  - Umwandeln in einen endlichen deterministischen Automaten
  - Optimieren
- Wiederholung deterministischer endlicher Automat
  - Sei $A = (Q, \Sigma, \delta, s_0, F)$ ein DEA

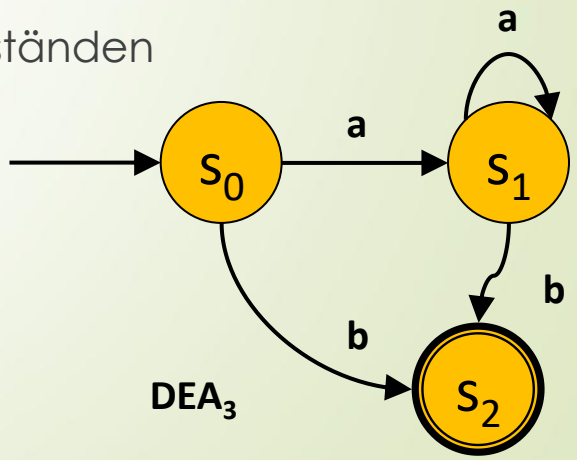
    Q eine endliche nicht leere Menge von Zuständen

    Σ ist ein Alphabet von Eingabezeichen

    $\delta : Q \times \Sigma \rightarrow Q$ eine Überführungsfunktion

    $s_0 \in Q$ ist ein Anfangszustand

    $F \subseteq Q$ ist eine Menge von Endzustände



Tokenbeschreibung

Zustandsdiagramme Beispiel

- Wir wählen folgendes Grammatikfragment

LETTER → [A - Za - z]

DIGIT → [0 - 9]

SIGN → + | -

PKT → .

ID → LETTER (LETTER | DIGIT)*

INT → SIGN? DIGIT+

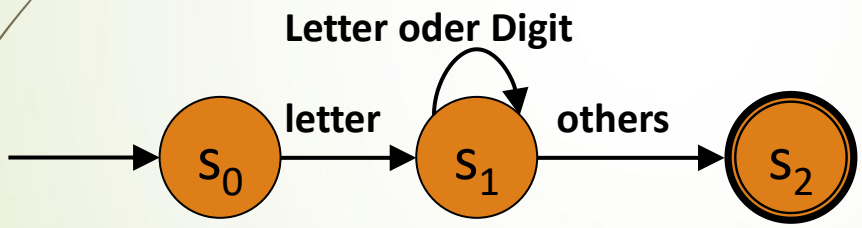
FLOAT → SIGN? DIGIT+ (PKT DIGIT+)? (E SIGN? DIGIT+)?

DELIM → [‘ ‘\t\n]

Tokenbeschreibung

Zustandsdiagramme Beispiel

ID → LETTER (LETTER | DIGIT)*



return(ID)

Aufgabe

- Erstellen Sie den endlichen Automaten für die Regeln

  INT → SIGN? DIGIT+

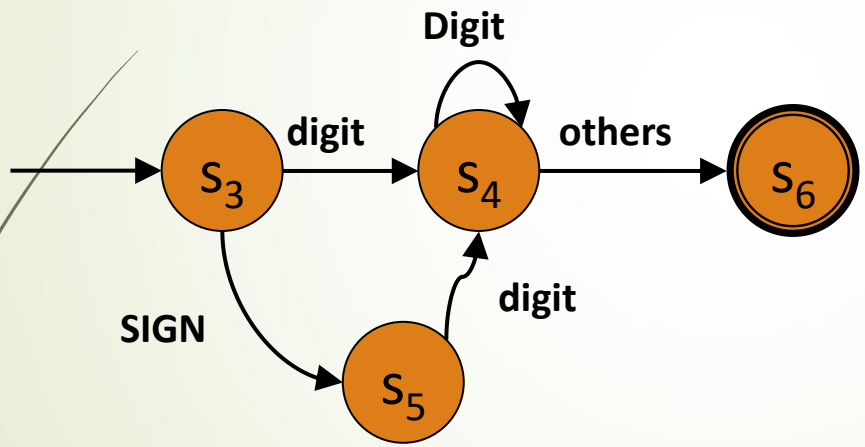
  FLOAT → SIGN? DIGIT+ PKT (DIGIT+)? (E SIGN? DIGIT+)?

  DELIM → [‘ ‘\t\n]+

Tokenbeschreibung

Zustandsdiagramme Beispiel

INT→ SIGN? DIGIT+

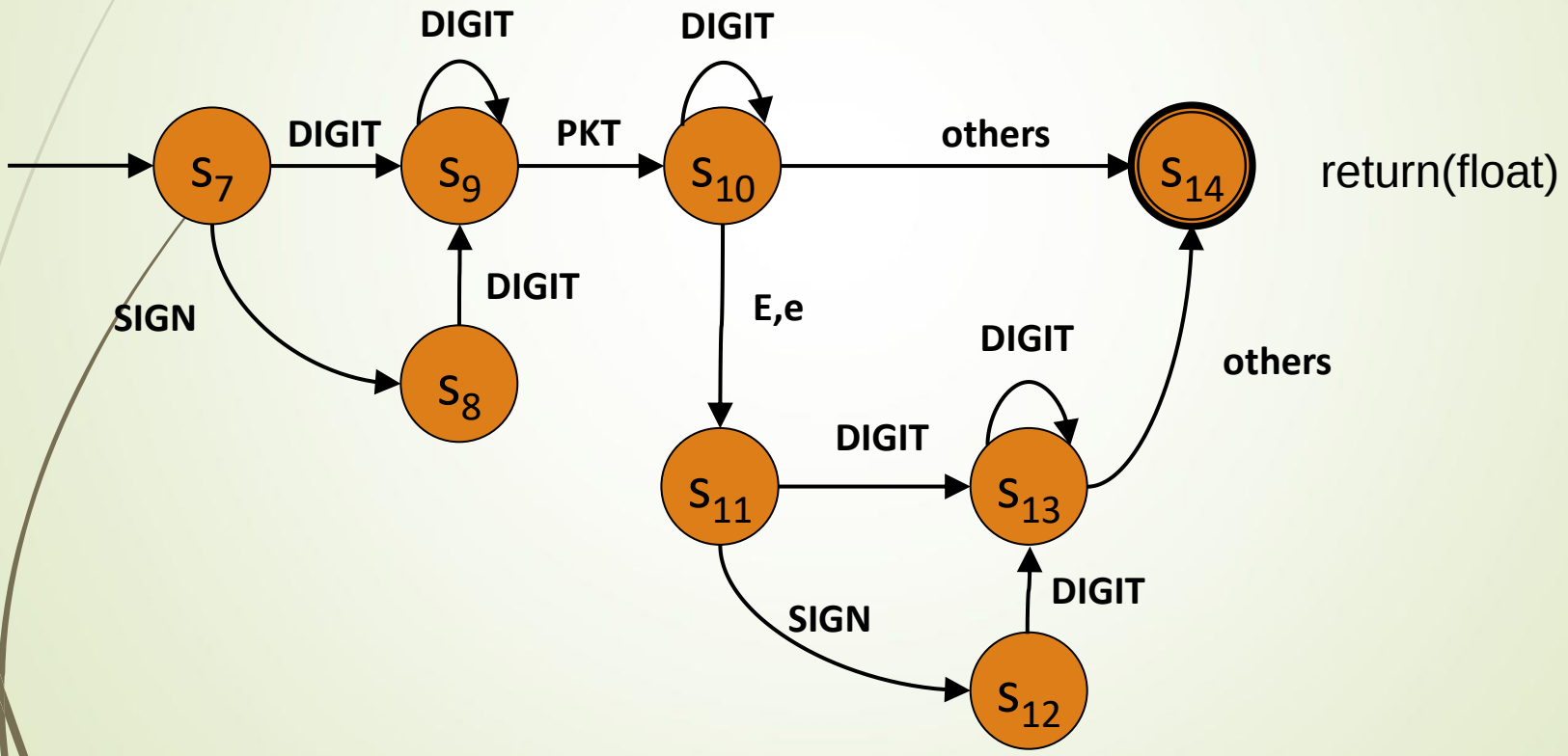


return(INT)

Tokenbeschreibung

Zustandsdiagramme Beispiel

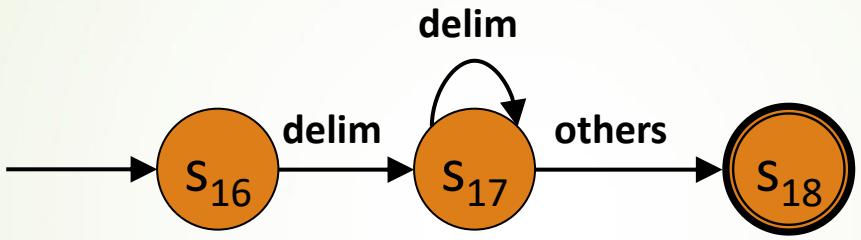
FLOAT → SIGN? DIGIT+ PKT (DIGIT*)? ((E|e) SIGN? DIGIT+)?



Tokenbeschreibung

Zustandsdiagramme Beispiel DELIM

DELIM → [‘ ‘\t\n]

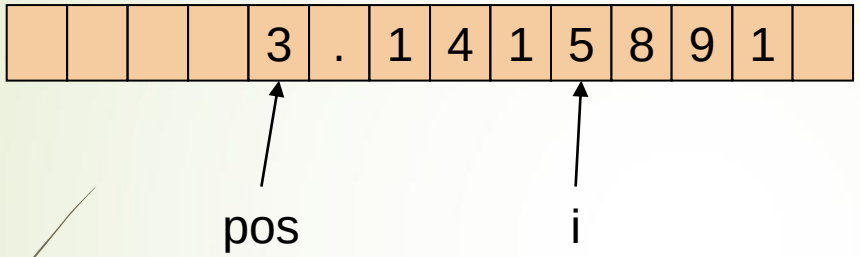


Implementierung

- Einrichten eines Einlesepuffers
- Aufbau des Einlesepuffers
- Aufbau des Main-Programms
- Der Token-Automat
- Aufbau der endlichen Automaten zu den Lexer-Regeln

Implementierung - Einlesepuffers

- Die Eingabezeichen stehen in einem Pufferbereich




```
StringBuffer s = new StringBuffer();
```

- Zur Verwaltung benötigen wir 2 Zeiger
  - pos: zeigt auf das nächste zu lesende Token
  - i: zeigt auf das nächste zu lesende Zeichen

Aufbau des Einlesepuffers

- Einlesen der Daten in einen Puffer:

```
BufferedReader f = New BufferedReader(new
    FileReader("file.txt"));
int c = f.read(); // get the first character
int i, pos, state; // Die Puffervariablen
StringBuffer s = new StringBuffer();
while ( c != -1) {
    s.append((char)c); // Füllen des Puffers
    c = f.read();
    }
```

- Die Daten liegen nun unstrukturiert im Puffer

Das Main-Programm

```
public static void main(String[] args) {
        BufferedReader f;
        StringBuffer input;
        input = new StringBuffer(); // Initialisieren des StringBuffer
        int c;
// Füllen des Eingabepuffers
        try {
            f = new BufferedReader(new FileReader("file.txt"));
            c = f.read();
            while ( c != -1) {
                    input.append((char)c); // Füllen des Puffers
                    c = f.read();
            }
            System.out.printf("Puffer der Länge %d gefüllt\n",input.length());
// Fehlerbehandlung
        }catch (FileNotFoundException e) {System.out.printf("File not
    found");
        }catch (IOException e) {System.out.printf("could not read file");
```

Das Main-Programm

```
// Ermitteln der Token
        int pos =0;
        token t = new token();
        while(t.gettoken(input,pos)){
            System.out.printf("Name:%s  Wert:%s Länge:%d\n",t.name,t.value,t.length);
            pos +=t.length;
            if(pos >= input.length()-1)break;
            }
        }
```

Implementierung – gettoken

- Soll das nächste Token zurückgeben

```
private boolean gettoken(StringBuffer in, int pos)
{
    char c;
    c = in.charAt(pos);
    while(true) {
        if(c == ' ' || c == '\t') return WS(in,pos); //Whitespace
        else if(Character.isDigit(c)) return FLOAT(in,pos); // Integer-Werte
        else if(Character.isLetter(c))return LETTER(in,pos); // Letter-Werte
        else {
            this.name = "SP";
            this.value=Character.toString(c);
            this.length = 1;
            pos += 1;
            return true;
            }
        }
}
```

Implementierung – LETTER

```
private boolean LETTER(StringBuffer in, int pos){
        int state,i;
        state = 0; i = 0;
        while(true) {
                switch (state){
                // Die Zustände (case) in Switch entsprechen den
                // Zuständen des endlichen Automaten
                case 0:while(Character.isLetter(in.charAt(pos+i)))i++;
                        state = 1;
                        break;
                case 1:this.name = "LETTER";
                        this.value=in.substring(pos, pos+i);
                        this.length = i;
                        pos += i;
                        return true;
                        }
                }
        }
```

Programmieraufgaben Java

- Entwickeln Sie den Code für
  - Whitespace
  - Float und INT

Lösung Java
Whitespace

```
// Behandeln von Whitespace
private boolean WS(StringBuffer in, int pos){
    int state,i,nmax;
    state = 0;
    i = 0;
    nmax = in.length();
    while(true) {
        switch (state){
        case 0:
                while(in.charAt(pos+i)== ' ' |in.charAt(pos+i)== '\t'){
                    i++;
                    if(i+pos >= nmax)break;
                }
                state = 1;
                break;
        case 1:this.name="WS";
                this.value=" ";
                this.length = i;
                this.error="";
                pos += i;
                return true;
            }
    }
}
```

Lösung Java
INT + Float

```
// Behandeln von INT und Float-Werten
    private boolean FLOAT(StringBuffer in, int pos){
        int state,i,nmax;
        state = 0;
        i = 0;
        nmax = in.length();
        while(true) {
            switch (state){
            case 0:while(Character.isDigit(in.charAt(pos+i))){
                    i++;
                    if(i+pos >= nmax)break;
                    }
                    state = 1;
                    break;
            case 1:if(in.charAt(pos+i) == '.' ){
                       i++;
                       state = 2;
                       break;
                    }
                    this.name="INT";
                    this.value=in.substring(pos, pos+i);
                    this.length = i;
                    pos += i;
                    return true;

            case 2: while(Character.isDigit(in.charAt(pos+i))){
                       i++;
                       if(i+pos >= nmax)break;
                       }
                    state = 3;
                    break;
            case 3:if(in.charAt(pos+i) == 'E' ){
                    i++;
                    state = 4;
                    } else state = 5;
                    break;
            case 4:
while(Character.isDigit(in.charAt(pos+i))){
                       i++;
                       if(i+pos >= nmax)break;
                       }
                    state = 5;
                    break;
            case 5: this.name="FLOAT";
                    this.value=in.substring(pos, pos+i);
                    this.length = i;
                    pos += i;
                    return true;
                }
            }
        }
```