

Übersetzerbau: Übung 06

von

Naja v. Schmude (4127652), Lisa Dohrmann (4130066)

Aufgabe 2

Wir weisen Schlüsselwörtern einen eindeutigen Tokennamen zu, Bezeichner haben als Attribut einen Zeiger auf den zugehörigen Eintrag in der Symboltabelle, Operatoren fassen wir zu geeigneten Gruppen zusammen und unterscheiden sie innerhalb der Gruppe durch eindeutige Attributwerte. Interpunktionssymbole erhalten wieder einen eindeutigen Tokennamen.

Lexem	Tokenname	Attributwert
function	function	-
begin	begin	-
end	end	-
if	if	-
then	then	-
else	else	-
integer	integer	-
max	id	Zeiger in Symboltabelle auf max
i	id	Zeiger in Symboltabelle auf i
j	id	Zeiger in Symboltabelle auf j
>	relop	GT
:=	op	ASGN
:	op	TASGN
(r_brack	-
)	l_brack	-
;	semicolon	-

Aufgabe 3

Wir haben uns etwas Java-Code für eine Implementierung der Funktion `nextChar()` überlegt.

```

/* pagesize */
int n = 4096;
char[] currBuf, buf1, buf2;
/* a special character that marks the end of a file */
char eof = '\eof';
int pointer;

void init() {
    buf1 = new char[n+1];
    buf2 = new char[n+1];
    currBuf = buf1;
    buf1[n] = eof;
    buf2[n] = eof;
    pointer = 0;

    /* read n characters from the file */

```

```

    readFromFile(buf1, n);
}

char nextChar() {
    char next = currBuf[++pointer];
    if(next == eof) {
        if (pointer == n) {
            swapBuffers();
            next = currBuf[pointer];
        }
        else // eof is within the buffer, end of input
            return eof;
    }
    return next;
}

void swapBuffers() {
    pointer = 0;
    if(currBuf.equals(buf1) currBuf = buf2;
    else currBuf = buf1;
    readFromFile(currBuf, n);
}

```

Aufgabe 4

- a) Wir haben die Operationen nach ihrer Bindungsstärke geordnet. Die Grammatik G sieht wie folgt aus:

```

digit -> 0 | .. | 9
digits -> digitdigits | ε
small -> a | .. | z
letter -> A | .. | Z | a | .. | z | _
letters -> letterletters | ε
decimal -> digits.digits
id -> smalllettersdigits
base -> id | digits | decimal | (exp)
trig -> sin(trig) | cos(trig) | tan(trig) | base
pot -> trig ** pot | trig
term -> term * pot | pot
exp -> exp + term | term

```

- b) Wie überlegen uns zunächst reguläre Definitionen für die wichtigen Tokennamen, um daraus Übergangsdiagramme erzeugen zu können.

```

small -> [a-z]
letter_ -> [a-zA-Z_]
digit -> [0-9]

id -> small letter_* digit*
num -> digit+
decimal -> digit+.digit+
+ -> +
* -> *
** -> **
( -> (
) -> )

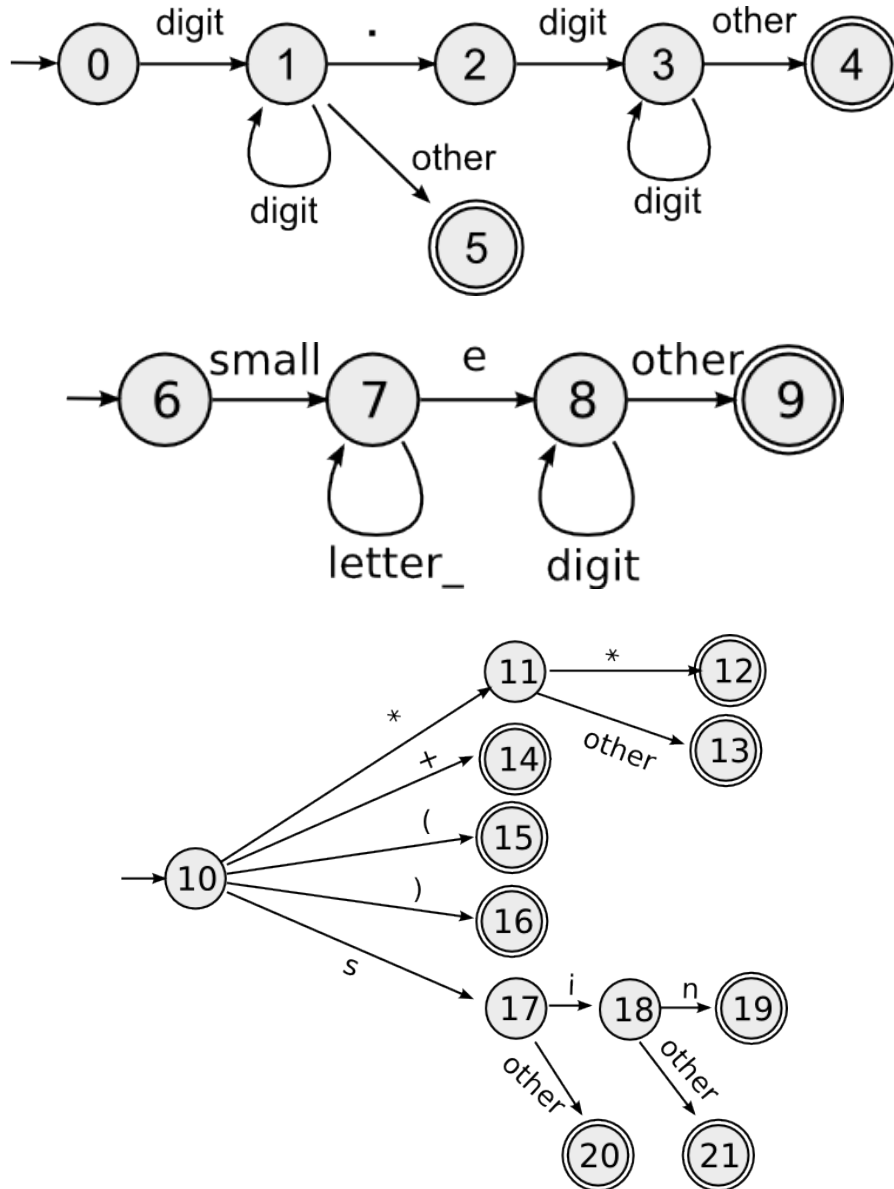
```

```

sin -> sin
cos -> cos
tan -> tan

```

Jetzt haben wir uns die Übergangsdiagramme überlegt.



Hat man einen der Endzustände erreicht, wird Folgendes zurückgegeben. In buffer stehe hierbei das gelesene Lexem und die `installXX`-Funktionen schreiben das Lexem in die Symboltabelle und liefern einen Zeiger auf diesen Eintrag zurück.

- (4) return new Token(decimal, installDecimal())
- (5) return new Token(num, buffer)
- (9) return new Token(id, installID())
- (11) return new Token(op, buffer)
- (12) return new Token(l_brack)
- (13) return new Token(r_brack)
- (14) return new Token(trig, buffer)

Es folgt der Pseudocode für einen Scanner, der $L(G)$ erkennt.

```

int start = 0; state = 0; lexemBegin = 0; char c;
char[] delims = {'_', '\t', '\n'};

Token nextToken() {
    switch(state) {
        case 0:
            c = nextChar();

            /* discard delimiters */
            while (contains(delim, c)) {
                lexemBegin++;
                c = nextChar();
            }

            if(!isInt(c)) {
                state = fail();
                break;
            }

            int buffer = 0;
            while (isInt(c)) {
                buffer *= 10;
                buffer += (int) c;
                c = nextChar();
            }
            if(c == '.') {
                state = 4;
                double decimal = 0;
                int i = 1;
                c = nextChar();
                while (isInt(c)) {
                    decimal += (int) c / 10^i;
                    ++i;
                    c = nextChar();
                }
                decimal += buffer;
                return new Token(decimal, installDecimal());
            }
            else {
                state = 5;
                return new Token(num, buffer);
            }
            break;
        case 6:
            c = nextChar();
            if(!('a' <= c && c <= 'z')) {
                state = fail();
                break;
            }
            String id = "" + c;
            c = nextChar();
            if (! isLetter_(c)) {
                state = fail();
                break;
            }
            state = 7;
            while (isLetter_(c)) {
                id += c;
                c = nextChar();
            }
            if (!isInt(c)) {
                state = fail();
                break;
            }
            state = 8;
            while (isInt(c)) {
                id += c
            }
    }
}

```

```

        return new Token(id, installID());
    case 10:
        c = nextChar();
        if (c == '+') {
            state = 14;
            return new Token('+');
        }
        else if (c == '*') {
            c = nextChar();
            if (c == '*') {
                state = 12;
                return new Token('**');
            }
            else {
                state = 13;
                return new Token('*');
            }
        }
        else if (c == '(') {
            state = 15;
            return new Token('(');
        }
        else if (c == ')') {
            state = 16;
            return new Token(')');
        }
        else if (c == 's') {
            state = 17;
            c = nextChar();
            if (c == 'i') {
                state = 18;
                c = nextChar();
                if (c == 'n') {
                    state = 19;
                    return new Token(sin);
                }
                else {
                    state = fail();
                }
            }
            else {
                state = fail();
            }
        }
        // äquivalent für cos und tan ...
        else if () {...}
    }
}

int fail() {
    forward = lexemBegin;
    switch(start) {
        case 0: start = 6; break;
        case 6: start = 10; break;
        default: report("Lex-error");
    }
    return start;
}

```

Aufgabe 5

- a), b) Das LEX-Programm soll um das Schlüsselwort **while** erweitert werden und zusätzlich sollen die Vergleichsoperatoren mit denen aus Java ersetzt werden.

```

%{ /* Wir erwarten die Verwendung der syntaktischen Konstruktoren
LE,LT, ..., GT, IF, WHILE, ID, NUM im Parser */

```

```

%}
/* Jetzt die regulären Definitionen */
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter} | {digit})*
number {digit}+ (\.{digit}+)?(E[+-]? {digit}+)?
%%
{ws} {/* es ist nichts zu tun bei einem Whitespace*/}
if {return (IF)}
else {return (ELSE)}
while {return (WHILE)}
...
{id} {yyLval = (int) installID(), return (ID);}
{number} {yyLval = (int) installNum(); return (NUM);}
"<" {setval = LT; return (RELOP);}
"<=" {setval = LE; return (RELOP);}
">" {setval = GT; return (RELOP);}
">=" {setval = GE; return (RELOP);}
"==" {setval = EQ; return (RELOP);}
"!=" {setval = NE; return (RELOP);}
%%
installID() {...}
installNUM() {...}

```

- c) Um den `_` als zusätzlichen Buchstaben zuzulassen, muss die Zeile `letter [A-Za-z]` nach `letter [A-Za-z_]` abgeändert werden.
- d) Nun sollen noch Strings erkannt und in einer eigenen Tabelle abgespeichert werden.

```

%{ Im Deklarationsteil wird folgendes eingefügt %}
punctuation [.,\-\_?!\"\'\\+\*]
string \"({letter} | {punctuation} | {digit} | {ws})* \"
%%
%{ Und folgendes bei den Transitionen %}
{string} {yyLval = (int) installSTRING(); return (STRING);}
%%
%{ Und schlussendlich dies bei den Hilfsfunktionen %}
installSTRING() {...}

```