Compilerbau

Übung

Dr. Klaus Ahrens Dorian Weber

Institut für Informatik
Humboldt-Universität zu Berlin

Inhalt

Aufgabe des Scanners

1. Erkennung von Token im Eingabestrom
   - etwa Zahlen, Zeichenketten, Bezeichner, Schlüsselworte, Operatoren
2. Ausfiltern von zu ignorierenden Zeichenketten
   - etwa Kommentare, Leerzeichen
3. Abfangen lexikalischer Fehler
   - etwa ungenutzte Zeichen, geöffnete Kommentare am Ende des Eingabestromes
4. Anreicherung der Token mit Positionsinformationen
   - Dateiname, Zeilen- und Spaltenintervall

Arbeitsschritte

1. Spezifikation der Tokenarten mithilfe einer geeigneten Menge von regulären Ausdrücken
2. Konstruktion eines NFA für den vereinigten regulären Ausdruck
3. Überführung des NFA in einen DFA mittels Potenzmengenkonstruktion
4. Minimierung der Anzahl der Zustände des DFA (optional)
5. Umwandlung des DFA in Code- oder Tabellenform

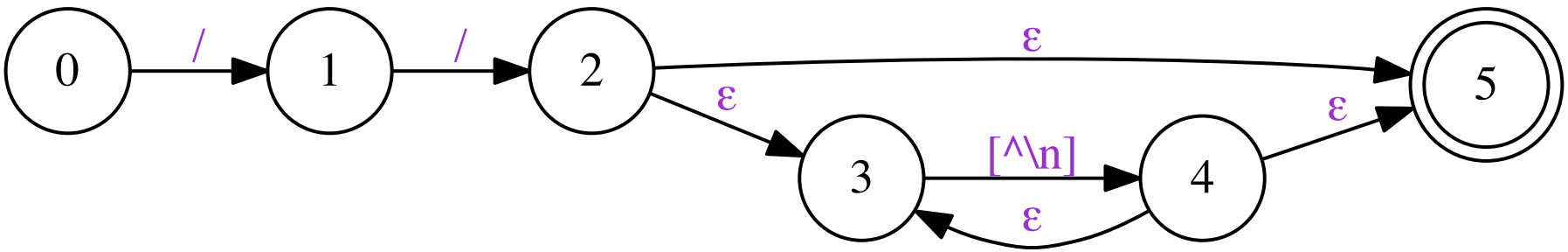
Reguläre Ausdrücke

Ausdruck	Bedeutung	Beispiel
c	ASCII-Zeichen c, außer Operatoren	`a`
\c	Zeichen c oder Escapesequenz	`\*` bzw. `\n`
$r_1 r_2$	r_1 gefolgt von r_2	`ab`
$r_1 \mid r_2$	r_1 oder r_2	`a|b`
(r)	Gruppiertes r	`(foo|bar)`
$r?$	Optionales r	`(word)?`
$r*$	Null oder mehr r's	`(words)*`
$r+$	Ein oder mehr r's	`(words)+`
.	alle Zeichen außer Zeilenende	`a.?c`
$[s]$	eines der Zeichen aus s	`[a-z]`
$[\hat{}s]$	keines der Zeichen aus s	`[^a-z]`

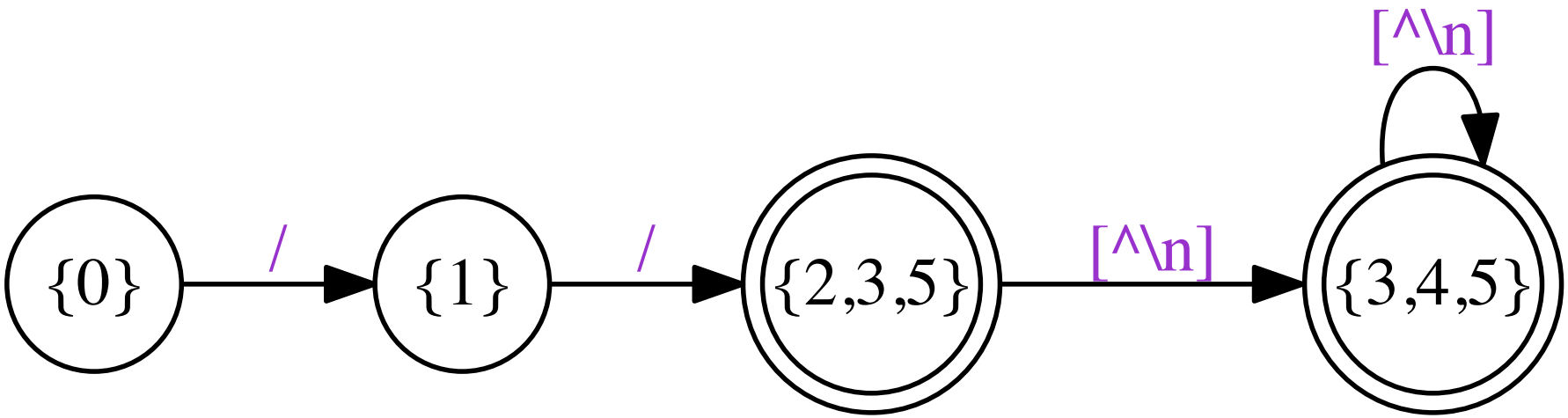
Spezifikation von Token

- Überlegen Sie sich reguläre Ausdrücke für:
  1. Zeilenkommentare, z.B. `// Kommentar`
  2. Zeichenkettenliterale, z.B. `"String"`
  3. Bezeichner, z.B. `Syntree` oder `tree_ptr`
  4. Integerliterale, z.B. `12` oder `0x12` oder `0755`
- Lösungsvorschläge:
  1. `//[^\n]*`
  2. `"[^"]*"`
  3. `[a-zA-Z_][a-zA-Z_0-9]*`
  4. `(0(x[0-9a-f]+|[0-7]*))|([1-9][0-9]*)`
- Konstruieren Sie für jeden Ausdruck einen NFA und wandeln diesen in einen DFA um.

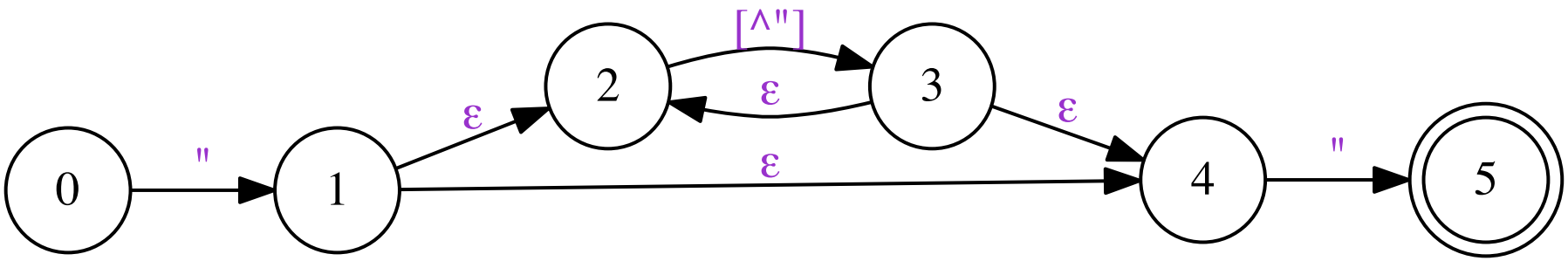
Kommentar (NFA)



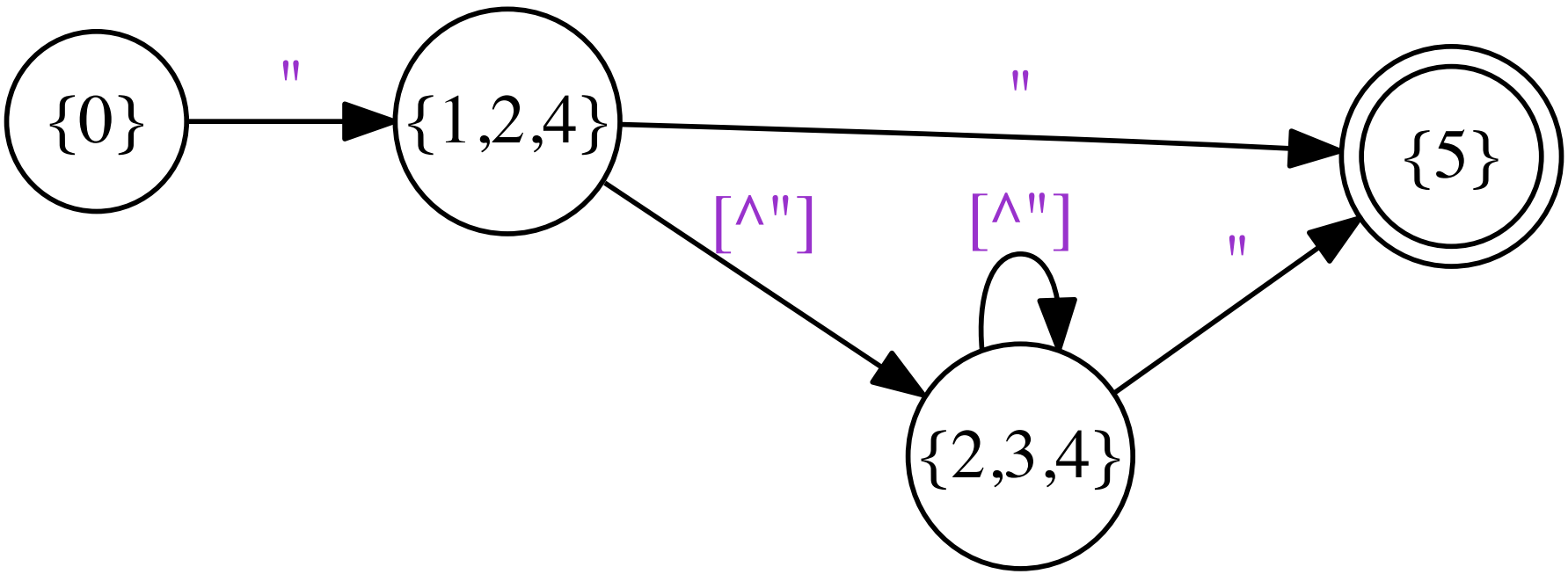
Kommentar (DFA)



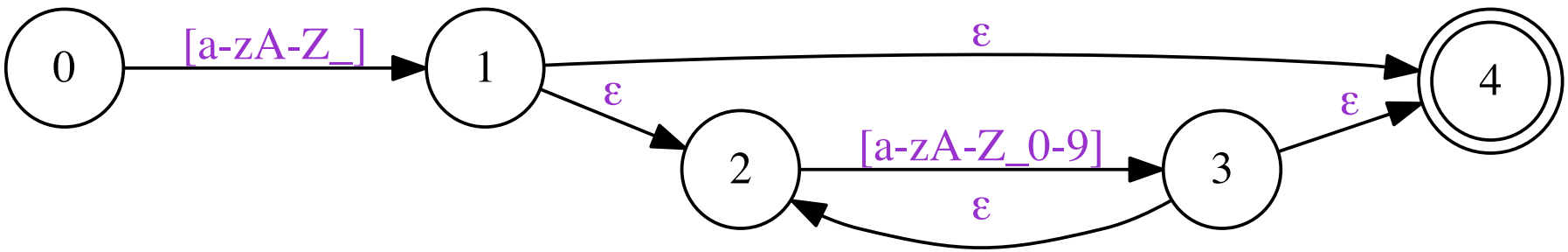
String (NFA)



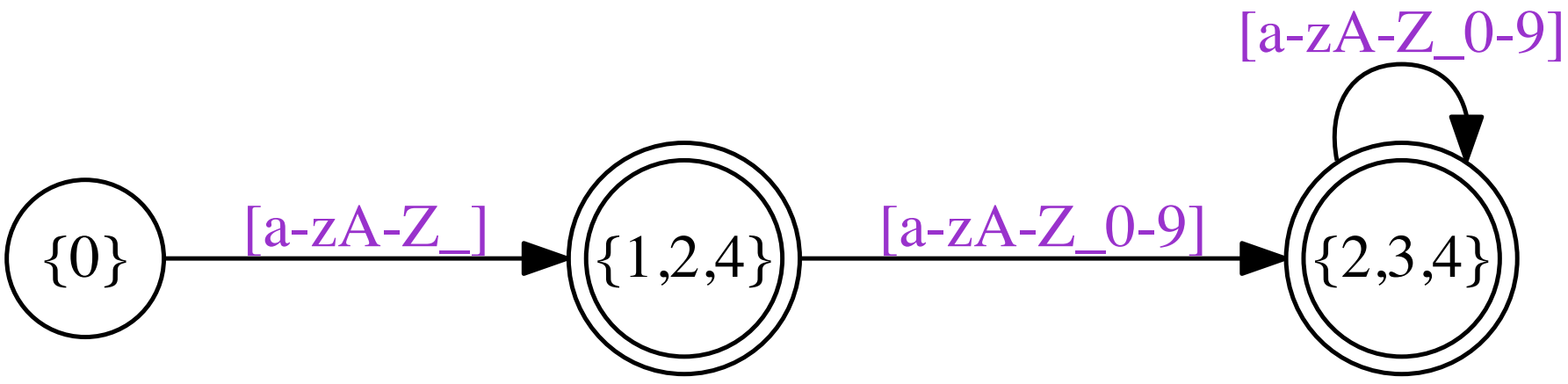
String (DFA)



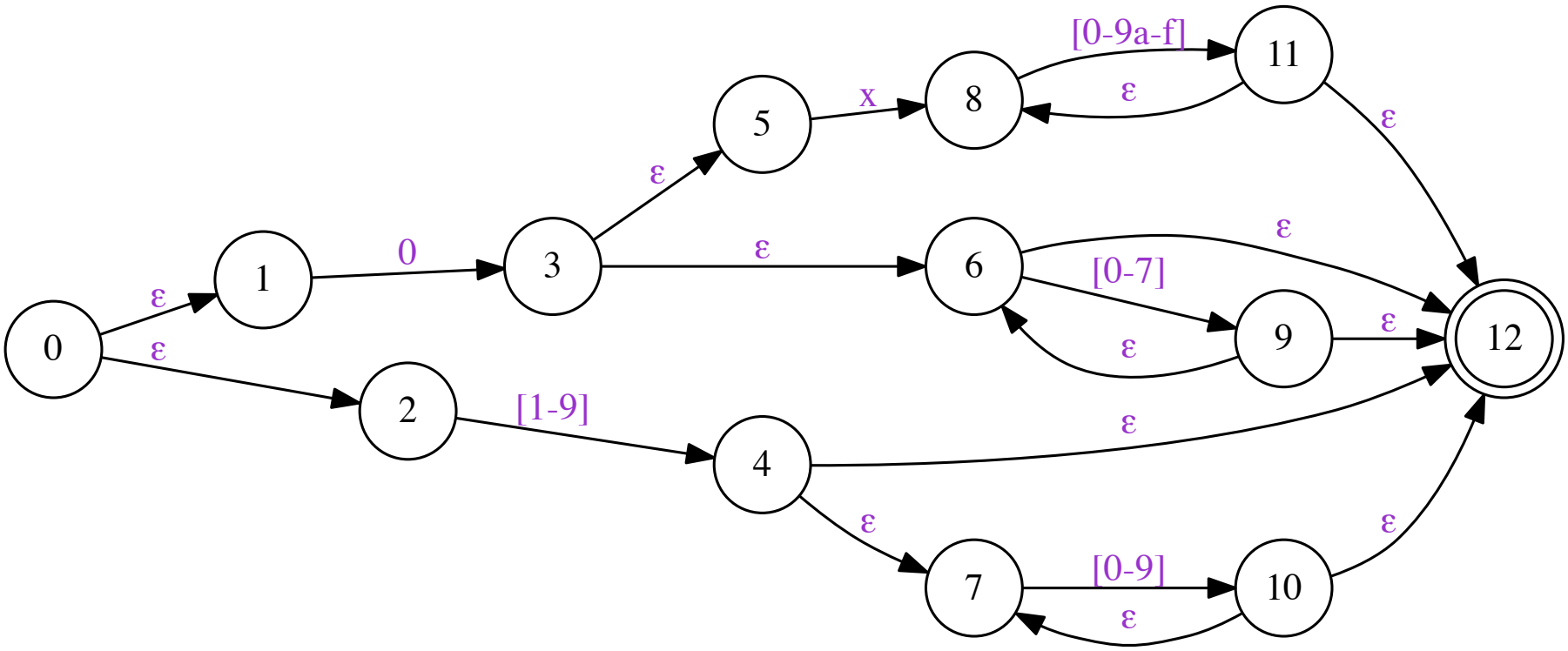
Bezeichner (NFA)



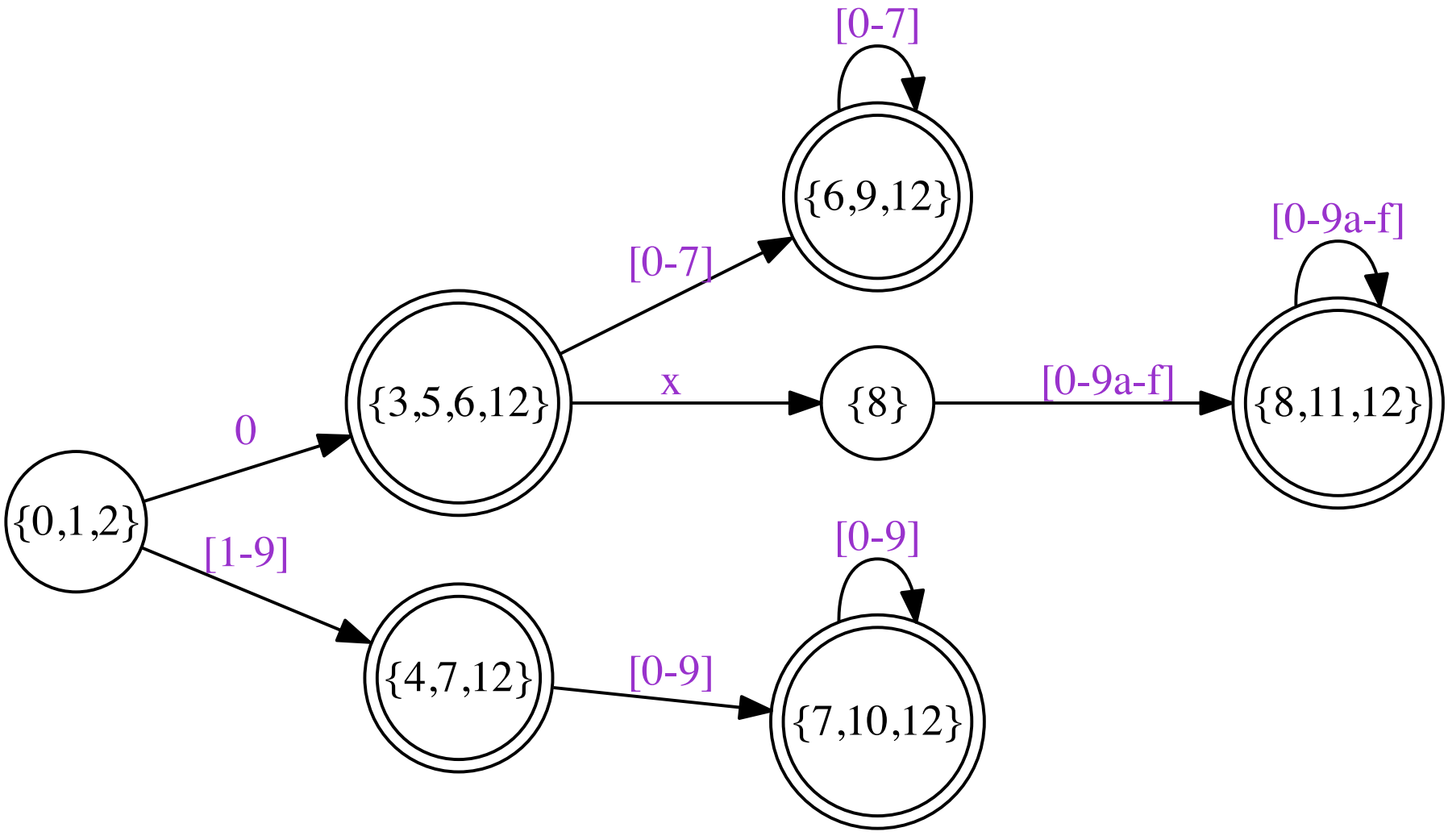
Bezeichner (DFA)



Ganzzahl (NFA)



Ganzzahl (DFA)



Überblick über das zweite Aufgabenblatt

- im Rahmen der Übungen soll ein Interpreter für eine abgespeckte Variante von C (C1) entwickelt werden
- die erste Aufgabe auf dem Aufgabenblatt beinhaltet die Implementation des Scanners bzw. Lexers
  - dazu geben wir eine Beschreibung der lexikalischen Token in Form von regulären Ausdrücken vor
  - die Aufgabe besteht darin, diese Spezifikation unter Nutzung von `flex` in eine C-Funktion zu überführen
  - diese Funktion wird später der Parser für jedes Token rufen
- in der zweiten Aufgabe soll ein Lexer für die Extraktion von Links und Linktext aus HTML-Text geschrieben werden
  - dazu geben wir nur eine textuelle Beschreibung vor
  - auch hier soll eine C-Funktion durch `flex` generiert werden
  - diesen Teil benötigen wir später nicht mehr

Vorgabe

- wir geben für beide Aufgaben ein Testprogramm, eine kleine Testeingabe sowie die erwartete Ausgabe vor
- darüber hinaus geben wir jeweils eine Headerdatei vor, in der die Tokentypen enumeriert und die Tokendaten definiert sind
- (dieser Headerdateien werden später durch den Parsergenerator erzeugt, aber hier sind sie erstmal vorgegeben)
- zusätzlich gibt es wieder Makefiles mit den bekannten Zielen

Vorgabe für den C1-Scanner

```c
YYSTYPE yylval;
int main(int argc, char *argv[]) {
  int token;
  yyin = (Eingabestrom öffnen);

  while ((token = yylex()) != EOF) {
    printf("Line: %3d\t", yylineno);
    switch (token) {
    case ID:
      printf("ID: %s\n", yylval.string);
      break;
    (weitere Tokenarten behandeln)
    default:
      if (token <= 255)
        printf("Token: '%c'\n", token);
      else
        printf("Token: %d\n", token);
    }
  }
}
```

```c
#ifndef YYSTYPE
typedef union {
    char *string;
    double floatValue;
    int intValue;
} yystype;
#define YYSTYPE yystype
#endif

#define AND 257
#define OR  258
#define EQ  259
// ...
#define ID  280

extern YYSTYPE yylval;
extern int yylex();
extern FILE *yyin;
extern int yylineno;
```

Flex

- 1975 wurde der Lexergenerator `Lex` von Mike Lesk und Eric Schmidt in C geschrieben
- 1987 als Open Source Variante `Flex` von Kevin Gong und Vern Paxson neu entwickelt und verbessert
- beide automatisieren die Konstruktion von effizienten DFA aus regulären Ausdrücken und erzeugen entsprechenden C Code:
  1. konstruiere NFA aus regulärem Ausdruck
  2. konstruiere DFA aus NFA
  3. minimiere Zustände des DFA
  4. erzeuge daraus C-Code für die Scanfunktion
- für angenehmere Nutzung wird die Scanfunktion `int yylex()` im pull-Modus generiert, d.h. es wird ein Token pro Funktionsruf erkannt und zurückgegeben
  - die Signatur und der Name lassen sich später noch anpassen, das spielt für diese Aufgabe aber keine Rolle

Formatübersicht

```
1  %option Flex-Optionen
2  Flex-Makro <Regulärer Ausdruck>
3
4  %{
5      #include <Header>
6  %}
7
8  %%
9
10 Muster1 <Aktionscode1>
11 Muster2 <Aktionscode2>
12 ...
13
14 %%
15
16 Nutzercode
```

Beispiel

```
1  %option noinput nounput noyywrap
2  IDENT [a-zA-Z_][a-zA-Z_0-9]*
3
4  %{
5      #include <stdio.h>
6  %}
7
8  %%
9
10 {IDENT}                                              return 1;
11 (0(x[[:xdigit:]]+|[0-7]*))|[1-9][[:digit:]]* return 2;
12 \"[^"]*\"                                           return 3;
13
14 %%
15
16 int main() {
17     int token;
18     while ((token = yylex()) != 0)
19         printf("%i\n", token);
20 }
```

Erweiterung der regulären Ausdrücke

Ausdruck	Bedeutung	Beispiel
"s"	Zeichenkette s	`"ab1*"`
$r\{m\}$	m r's	`a{5}`
$r\{m,\}$	m oder mehr r's	`a{3,}`
$r\{m,n\}$	m- bis n-mal r's	`a{1,5}`
^	Match am Zeilenanfang	`^abc`
$	Match am Zeilenende	`abc$`
r_1/r_2	r_1, falls dahinter r_2 folgt	`abc/123`
$[s_1]\{+\}[s_2]$	Vereinigung von s_1 und s_2	`[a-d]{+}[c-f]`
$[s_1]\{-\}[s_2]$	Differenz zwischen s_1 und s_2	`[a-d]{-}[c-f]`

Reservierte Zeichen

- Operatoren haben in regulären Ausdrücken besondere Bedeutung und müssen mit \ maskiert werden
  `\ ^ $ . [ ] | ( ) * + ? { } " % < > /`
- Steuerzeichen aus ANSI-C sind ebenfalls gültig, etwa
  - \n Line Feed
  - \r Carriage Return
  - \t Horizontal Tab
- `<<EOF>>` matcht nur am Ende des Zeichenstromes

Abschnitt 1: Deklaration

- Makros zur Vereinfachung von Nutzercode
- Optionen für den erzeugten Flex-Scanner sind u.a.

`noyywrap`	nimmt an, dass keine weiteren Eingabedateien vorliegen
`yylineno`	aktiviert die automatische Aktualisierung der globalen Variablen `yylineno` mit der aktuellen Zeilennummer
`nodefault`	erzeugt einen Fehler, falls keine Regel matcht, statt das Symbol auf der Standardausgabe auszugeben
`nounput`	deaktiviert die Erzeugung der Funktion `unput()`, die ein Zeichen in den Eingabestrom zurücklegt
`noinput`	deaktiviert die Erzeugung der Funktion `input()`, die ein Zeichen aus dem Eingabestrom zurückgibt
`main`	erzeugt eine Standard `int main()`, die so lange `yylex()` ruft, bis der Eingabestrom `stdin` aufgebraucht ist

Abschnitt 2: Regeln

- Suchmuster mit Aktionscode beim erfolgreichen Match
- Muster als regulärer Ausdruck am Anfang einer Zeile, ohne Einrückung
- Aktionscode wird nach Match ausgeführt und darf die Funktion mit `return` verlassen
- mehrere alternative Muster dürfen den gleichen Aktionscode ausführen
- vordefinierte Zeichenklassen:
  - `[:alpha:]`, `[:alnum:]`
  - `[:digit:]`, `[:xdigit:]`
  - `[:blank:]`, `[:cntrl:]`, `[:space:]`
  - `[:graph:]`, `[:print:]`, `[:punct:]`
  - `[:lower:]`, `[:upper:]`
  - alle negierten Varianten davon, z. B. `[:^alpha:]`

Abschnitt 3: Implementation

- beinhaltet etwa die Implementation der im ersten Abschnitt für dieses Modul deklarierten Funktionen
- hat Zugriff auf alle internen Funktionen des generierten Scanners
- (hier könnte z.B. die `int main()` stehen; da wir sie im Rahmen der Aufgabe vorgeben, muss hier erstmal kein Code erscheinen)

Matching

- DFA kann mit zusätzlichen Zuständen versehen werden
  - `%x STATE` Zustände matchen unabhängig von unmarkierten Mustern (exklusiv)
  - `%s STATE` Zustände matchen die eigenen und die unmarkierten Muster (inklusiv)
  - Zustandsübergang im Aktionscode mittels `BEGIN(STATE)`
  - initialer Zustand ist `INITIAL`
- `Flex` nutzt die POSIX-Konvention, falls mehrere Muster gleichzeitig matchen

  POSIX-Konvention ausgewählt wird das erste im Quelltext erscheinende Muster mit dem längsten Match

  Perl-Konvention ausgewählt wird die erste matchende Regel, unabhängig von der Länge des Matchings

Flex-Bezeichner

- `int yylex()` ist die generierte Funktion zum Aufruf des Scanners
  - Token-ID ist Rückgabewert
  - beim nächsten Aufruf wird an der nächsten Stelle weitergescannt
  - gibt `0` zurück, wenn Scannen erfolgreich beendet
- `FILE *yyin, *yyout`: globale Variablen für den Ein- und Ausgabestrom
  - diese kann man vor dem ersten Ruf von `yylex()` setzen, um die Ein- und Ausgabe umzuleiten
  - wenn man sie nicht setzt, sind sie standardmäßig auf `stdin` und `stdout` gesetzt
- es gibt vorbereitete Variablen, auf die der Nutzer im Aktionscode zugreifen kann:

  yytext Zeiger auf das erste Zeichen des aktuellen Matches
  yyleng Länge des Matches

Komplexeres Beispiel

```
%option main noinput nounput noyywrap
IDENT [[:alpha:]_][[:alnum:]_]*
%x COMMENT
%%

"/*"           { BEGIN(COMMENT); }
<COMMENT>"*/" { BEGIN(INITIAL); }
<COMMENT>.|\n

while     |
int       |
return    { printf("<kw:%s>", yytext); }

{IDENT}     printf("<id:%s>", yytext);
(0(x[[:xdigit:]]+|[0-7]*))|[1-9][[:digit:]]* {
    printf("<num:%s>", yytext);
}
\"[^"]*\" { printf("<str:%s>", yytext); }

%%
```