## 3.4. Abstrakte Syntax und Namenstabelle

### 3.4.1. Prinzip

Die abstrakte Syntax stellt für jedes richtige Konstrukt der Spezifikationssprache einen Knotentyp zur Verfügung.

Bsp.: "Cat" – Knoten für die Konkatenation mit den beiden Komponenten "Vorgänger" und "Nachfolger". Die Namenstabelle speichert die Abbildung Name ↦ AST für alle regulären Abkürzungen.

### 3.4.2. Konkrete Implementierung

Siehe Seiten S9 – S11

### 3.4.3. "Demo" – Beispiel

```
/*
 * absyn.h -- abstract syntax for scangen's input
 */


#ifndef _ABSYN_H_
#define _ABSYN_H_


#define NODE_RULES      0       /* sequence of rules */
#define NODE_RULE       1       /* single rule */
#define NODE_ALT        2       /* alternative */
#define NODE_CAT        3       /* concatenation */
#define NODE_CLOSURE    4       /* closure */
#define NODE_PCLOSURE   5       /* positive closure */
#define NODE_OPTIONAL   6       /* expression or epsilon */
#define NODE_NAME       7       /* name used for expression */
#define NODE_CHAR       8       /* single character */
#define NODE_CLASS      9       /* character class */
#define NODE_COMPONENT  10      /* character class component */


typedef struct node {
  int type;                     /* what type of node it is */
  int line;                     /* in which source line it is found */
  union {
    struct {
      struct node *rules;       /* list of rules */
    } rulesNode;
    struct {
      int num;                  /* rule number */
      struct node *exp;         /* regular expression */
      struct node *next;        /* link to next rule */
    } ruleNode;
    struct {
      struct node *left;        /* left alternative */
      struct node *right;       /* right alternative */
    } altNode;
    struct {
      struct node *left;        /* predecessor */
      struct node *right;       /* successor */
    } catNode;
    struct {
      struct node *exp;         /* regular expression */
    } closureNode;
    struct {
      struct node *exp;         /* regular expression */
    } pclosureNode;
    struct {
      struct node *exp;         /* regular expression */
    } optionalNode;
    struct {
      Name *name;               /* name of abbreviation */
    } nameNode;
    struct {
      int code;                 /* ASCII code */
    } charNode;
    struct {
      Bool compl;               /* take complement of components */
      struct node *components;  /* list of class components */
    } classNode;
    struct {
      int bgnCode;              /* start of ASCII code range */
      int endCode;              /* end of ASCII code range */
      struct node *next;        /* link to next component */
    } componentNode;
  } u;
} Node;
```

```
Node *newRules(int line, Node *rules);
Node *newRule(int line, int num, Node *exp);
Node *newAlt(int line, Node *left, Node *right);
Node *newCat(int line, Node *left, Node *right);
Node *newClosure(int line, Node *exp);
Node *newPclosure(int line, Node *exp);
Node *newOptional(int line, Node *exp);
Node *newName(int line, Name *name);
Node *newChar(int line, int code);
Node *newClass(int line, Bool compl, Node *components);
Node *newComponent(int line, int bgnCode, int endCode);

void showAbsyn(Node *node);


#endif /* _ABSYN_H_ */
```

```
/*
 * names.h -- handle names as abbreviations for regular expressions
 */


#ifndef _NAMES_H_
#define _NAMES_H_


typedef struct name {
  char *name;                   /* name of abbreviation */
  struct node *exp;             /* regular expression */
  struct name *next;            /* link to next pair */
} Name;


Name *lookup(char *name);
Name *enter(char *name, struct node *exp);

void showNames(void);


#endif /* _NAMES_H_ */
```

```
%{

/*
 * scan.sg -- scanner for demonstration program
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "tokens.h"

%}

:{D}            [0-9]
:{INT}          {D}+
:{EXP}          e(\+|\-)?{INT}
:{NUM}          ({INT}\.?|{INT}?\.{INT}){EXP}?
:{VAR}          [a-z]


%%


[\ \t]          %{
                  /* nothing to do here */
                %}

{NUM}           %{
                  scanVal.value = strtod(scanText, NULL);
                  return NUMBER;
                %}

{VAR}           %{
                  scanVal.index = scanText[0] - 'a';
                  return VAR;
                %}

\n              %{
                  return NEWLINE;
                %}

\+              %{
                  return PLUS;
                %}

\-              %{
                  return MINUS;
                %}

\*              %{
                  return STAR;
                %}

\/              %{
                  return SLASH;
                %}

\(              %{
                  return LPAREN;
                %}

\)              %{
                  return RPAREN;
                %}

\-\>            %{
                  return STORE;
```

```
%}

.               %{
                  printf("illegal character '%c' (0x%02X)\n",
                         scanText[0], scanText[0]);
                  exit(99);
                %}


%%
```

```
Names:
D =
[15] Class(
  false,
  [15] Component(0x30..0x39))
INT =
[16] Pclosure(
  [16] Name(D))
EXP =
[17] Cat(
  [17] Cat(
    [17] Char(0x65),
    [17] Optional(
      [17] Alt(
        [17] Char(0x2B),
        [17] Char(0x2D)))),
  [17] Name(INT))
NUM =
[18] Cat(
  [18] Alt(
    [18] Cat(
      [18] Name(INT),
      [18] Optional(
        [18] Char(0x2E))),
    [18] Cat(
      [18] Cat(
        [18] Optional(
          [18] Name(INT)),
        [18] Char(0x2E)),
      [18] Name(INT))),
  [18] Optional(
    [18] Name(EXP)))
VAR =
[19] Class(
  false,
  [19] Component(0x61..0x7A))

Rules:
[25] Rules(
  [25] Rule(
    #0,
    [25] Class(
      false,
      [25] Component(0x20..0x20),
      [25] Component(0x09..0x09))),
  [29] Rule(
    #1,
    [29] Name(NUM)),
  [34] Rule(
    #2,
    [34] Name(VAR)),
  [39] Rule(
    #3,
    [39] Char(0x0A)),
  [43] Rule(
    #4,
    [43] Char(0x2B)),
  [47] Rule(
    #5,
    [47] Char(0x2D)),
  [51] Rule(
    #6,
    [51] Char(0x2A)),
  [55] Rule(
    #7,
    [55] Char(0x2F)),
  [59] Rule(
    #8,
```

```
  [59] Char(0x28)),
[63] Rule(
  #9,
  [63] Char(0x29)),
[67] Rule(
  #10,
  [67] Cat(
    [67] Char(0x2D),
    [67] Char(0x3E))),
[71] Rule(
  #11,
  [71] Class(
    true,
    [71] Component(0x0A..0x0A))))
```

```
/*
 * scan.sg -- scanner for demonstration program
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "tokens.h"

/*
 * scan.skel -- the skeleton of a table-driven scanner
 */


#include <stdio.h>
#include <stdlib.h>


int scan(void) {
  /* THIS WILL BE REPLACED BY SOME CODE (1) */
        case   0:       {
                   /* nothing to do here */
                   break;
                        }
        case   1:       {
                   scanVal.value = strtod(scanText, NULL);
                   return NUMBER;
                   break;
                        }
        case   2:       {
                   scanVal.index = scanText[0] - 'a';
                   return VAR;
                   break;
                        }
        case   3:       {
                   return NEWLINE;
                   break;
                        }
        case   4:       {
                   return PLUS;
                   break;
                        }
        case   5:       {
                   return MINUS;
                   break;
                        }
        case   6:       {
                   return STAR;
                   break;
                        }
        case   7:       {
                   return SLASH;
                   break;
                        }
        case   8:       {
                   return LPAREN;
                   break;
                        }
        case   9:       {
                   return RPAREN;
                   break;
                        }
        case  10:       {
                   return STORE;
                   break;
```

```
                        }
        case  11:       {
                printf("illegal character '%c' (0x%02X)\n",
                        scanText[0], scanText[0]);
                exit(99);
                break;
                        }

  /* THIS WILL BE REPLACED BY SOME CODE (2) */
}
```