4.4. Abstrakte Syntax und Namenstabelle

4.4.1. Prinzip

Die abstrakte Syntax stellt genau zwei Knotentypen zur Verfügung: alternative rechte Seiten von Produktionen und Listen von Grammatiksymbolen für eine einzelne rechte Seite. Die Namenstabelle enthält zwei Sorten Einträge:

- für Token: die Token-Nummer
- für Nichtterminalsymbole: die Liste der rechten Seiten der zu diesem Symbol gehörenden Produktionen

4.4.2. Konkrete Implementierung

Siehe Seiten P11 – P12

4.4.3. "Demo"-Beispiel

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


#ifndef _ABSYN_H_
#define _ABSYN_H_


#define NODE_PROD       0       /* production */
#define NODE_SYM        1       /* grammar symbol */


typedef struct node {
  int type;                     /* what type of node it is */
  int line;                     /* in which source line it is found */
  union {
    struct {
      int prodNum;              /* production number */
      Name *lhs;                /* nonterminal on left-hand side */
      struct node *syms;        /* sequence of grammar symbols */
      struct node *next;        /* next production for same nonterminal */
    } prodNode;
    struct {
      Name *name;               /* name of grammar symbol */
      struct node *next;        /* next grammar symbol */
    } symNode;
  } u;
} Node;


Node *newProd(int line, int prodNum, Name *lhs, Node *syms);
Node *newSym(int line, Name *name);

void showAbsyn(Node *node);


#endif /* _ABSYN_H_ */
```

```
/*
 * names.h -- handle names for tokens and nonterminal symbols
 */


#ifndef _NAMES_H_
#define _NAMES_H_


typedef struct {
  char *name;                   /* name of symbol */
  Bool isToken;                 /* true iff symbol is a token */
  union {
    struct {                    /* valid only if isToken == true */
      int tokenNumber;          /* each token has a unique number */
    } tokenName;
    struct {                    /* valid only if isToken == false */
      struct node *firstProd;   /* NULL if symbol is not (yet) defined */
      struct node *lastProd;    /* used for appending productions */
    } nontermName;
  } u;
  Bool mark;                    /* used in semantic checks only */
} Name;


Name *defToken(char *name, int tokenNumber);
Name *refSymbol(char *name);

void showTokens(void);
void showNonterms(void);

Bool undefinedNonterms(void);
void showUndefinedNonterms(void);

void unmarkSymbols(void);
Bool unmarkedTokens(void);
void showUnmarkedTokens(void);
Bool unmarkedNonterms(void);
void showUnmarkedNonterms(void);

void markDeriving(void);


#endif /* _NAMES_H_ */
```

```
%{

/*
 * demo.pg -- demonstration program
 */

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

%}

%token          NUMBER
%token          VAR
%token          NEWLINE
%token          PLUS MINUS STAR SLASH
%token          LPAREN RPAREN
%token          STORE

%start          list


%%


list            : %empty
                  %{
                    /* nothing to do here */
                  %}
                | list NEWLINE
                  %{
                    /* nothing to do here */
                  %}
                | list assign NEWLINE
                  %{
                    printf("\t%.8g\n", $2.value);
                  %}
                ;

assign          : expr
                  %{
                    $$.value = $1.value;
                  %}
                | assign STORE VAR
                  %{
                    memory[$3.index] = $1.value;
                    $$.value = $1.value;
                  %}
                ;

expr            : term
                  %{
                    $$.value = $1.value;
                  %}
                | expr PLUS term
                  %{
                    $$.value = $1.value + $3.value;
                  %}
                | expr MINUS term
                  %{
                    $$.value = $1.value - $3.value;
                  %}
                ;

term            : factor
                  %{
                    $$.value = $1.value;
                  %}
```

```
                | term STAR factor
                  %{
                    $$.value = $1.value * $3.value;
                  %}
                | term SLASH factor
                  %{
                    if ($3.value == 0.0) {
                      printf("division by zero\n");
                      exit(99);
                    }
                    $$.value = $1.value / $3.value;
                  %}
                ;

factor          : primary
                  %{
                    $$.value = $1.value;
                  %}
                | PLUS factor
                  %{
                    $$.value = $2.value;
                  %}
                | MINUS factor
                  %{
                    $$.value = -$2.value;
                  %}
                ;

primary         : NUMBER
                  %{
                    $$.value = $1.value;
                  %}
                | VAR
                  %{
                    $$.value = memory[$1.index];
                  %}
                | LPAREN expr RPAREN
                  %{
                    $$.value = $2.value;
                  %}
                ;


%%


%{

int main(int argc, char *argv[]) {
  int i;

  printf("Demo started:\n");
  for (i = 0; i < 26; i++) {
    memory[i] = 0.0;
  }
  parse();
  printf("Demo stopped.\n");
  return 0;
}

%}
```

```
Tokens

LPAREN          (7)
MINUS           (4)
NEWLINE         (2)
NUMBER          (0)
PLUS            (3)
RPAREN          (8)
SLASH           (6)
STAR            (5)
STORE           (9)
VAR             (1)

Productions

assign :
        [40] Alt(
          #3,
          [40] Sym(expr))
    |
        [44] Alt(
          #4,
          [44] Sym(assign),
          [44] Sym(STORE),
          [44] Sym(VAR))
    ;

expr :
        [51] Alt(
          #5,
          [51] Sym(term))
    |
        [55] Alt(
          #6,
          [55] Sym(expr),
          [55] Sym(PLUS),
          [55] Sym(term))
    |
        [59] Alt(
          #7,
          [59] Sym(expr),
          [59] Sym(MINUS),
          [59] Sym(term))
    ;

factor :
        [83] Alt(
          #11,
          [83] Sym(primary))
    |
        [87] Alt(
          #12,
          [87] Sym(PLUS),
          [87] Sym(factor))
    |
        [91] Alt(
          #13,
          [91] Sym(MINUS),
          [91] Sym(factor))
    ;

list :
        [26] Alt(
          #0,
          <empty>)
    |
        [30] Alt(
```

```
          #1,
          [30] Sym(list),
          [30] Sym(NEWLINE))
    |
        [34] Alt(
          #2,
          [34] Sym(list),
          [34] Sym(assign),
          [34] Sym(NEWLINE))
    ;

primary :
        [97] Alt(
          #14,
          [97] Sym(NUMBER))
    |
        [101] Alt(
          #15,
          [101] Sym(VAR))
    |
        [105] Alt(
          #16,
          [105] Sym(LPAREN),
          [105] Sym(expr),
          [105] Sym(RPAREN))
    ;

term :
        [65] Alt(
          #8,
          [65] Sym(factor))
    |
        [69] Alt(
          #9,
          [69] Sym(term),
          [69] Sym(STAR),
          [69] Sym(factor))
    |
        [73] Alt(
          #10,
          [73] Sym(term),
          [73] Sym(SLASH),
          [73] Sym(factor))
  ;

Start Symbol = list
```

```
/*
 * demo.pg -- demonstration program
 */

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

/*
 * parse.skel -- the skeleton of a table-driven parser
 */


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


void parse(void) {
  /* THIS WILL BE REPLACED BY SOME CODE (1) */
        case   0:       {
                        /* nothing to do here */
                        break;
                        }
        case   1:       {
                        /* nothing to do here */
                        break;
                        }
        case   2:       {
                        printf("\t%.8g\n", $2.value);
                        break;
                        }
        case   3:       {
                        $$.value = $1.value;
                        break;
                        }
        case   4:       {
                        memory[$3.index] = $1.value;
                        $$.value = $1.value;
                        break;
                        }
        case   5:       {
                        $$.value = $1.value;
                        break;
                        }
        case   6:       {
                        $$.value = $1.value + $3.value;
                        break;
                        }
        case   7:       {
                        $$.value = $1.value - $3.value;
                        break;
                        }
        case   8:       {
                        $$.value = $1.value;
                        break;
                        }
        case   9:       {
                        $$.value = $1.value * $3.value;
                        break;
                        }
        case  10:       {
                        if ($3.value == 0.0) {
                          printf("division by zero\n");
                          exit(99);
                        }
                        $$.value = $1.value / $3.value;
```

```
                    break;
                        }
        case  11:       {
                    $$.value = $1.value;
                    break;
                        }
        case  12:       {
                    $$.value = $2.value;
                    break;
                        }
        case  13:       {
                    $$.value = -$2.value;
                    break;
                        }
        case  14:       {
                    $$.value = $1.value;
                    break;
                        }
        case  15:       {
                    $$.value = memory[$1.index];
                    break;
                        }
        case  16:       {
                    $$.value = $2.value;
                    break;
                        }

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


int main(int argc, char *argv[]) {
  int i;

  printf("Demo started:\n");
  for (i = 0; i < 26; i++) {
    memory[i] = 0.0;
  }
  parse();
  printf("Demo stopped.\n");
  return 0;
}
```