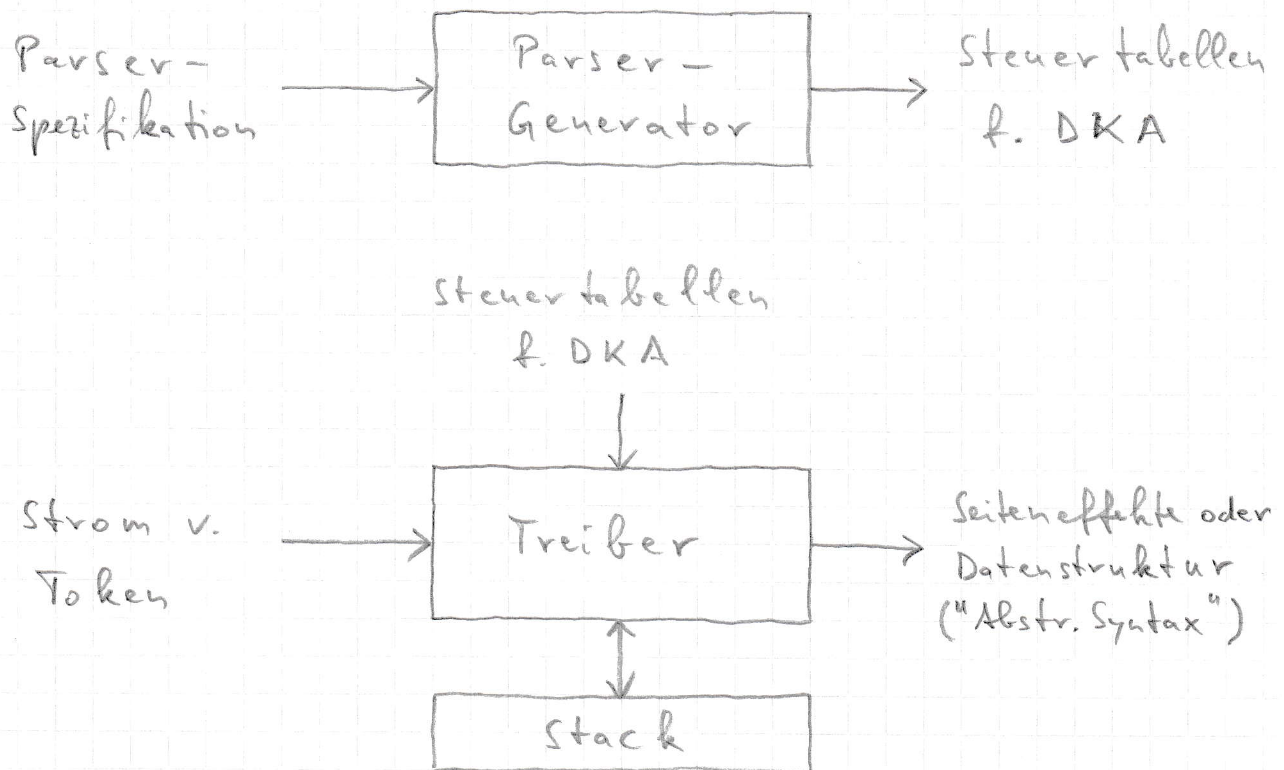
4. Parsergenerator

4.1. Funktionsweise



Die Steuertabellen bestimmen die erkannte Sprache, aber auch den Parsing-Algorithmus (LL(1), LR(0), SLR, LR(1), LALR(1)). Für den Fall LL(1) gibt es eine gute Alternative, wenn der Parser von Hand konstruiert werden soll: "Recursive Descent". Dabei wird für jedes Nichtterminalsymbol eine (rekursive) Prozedur geschrieben, die die rechten Seiten der Produktionen für das Nichtterminal akzeptiert. Der Stack tritt dabei explizit gar nicht in Erscheinung; er wird durch die Laufzeitumgebung der Implementierungssprache zur Verfügung gestellt und durch die rekursiven Prozeduraufrufe implizit genutzt.

4.2. Parser-Spezifikation

4.2.1. Elemente d. Spezifikationssprache

Zweck der Spez.-Sprache ist die Zuordnung von Aktionen (z.B. "$$ = $1 + $3;") zu Reduktionen beim LR-Parsen.
Eine "Reduktion" bedeutet, daß die vollständige rechte Seite einer Produktion der vorliegenden Grammatik im Strom der Tokens erkannt wurde (und evtl. k Tokens mehr, "LR(k)") und durch das Nichtterminal der linken Seite ersetzt wird.
Also müssen folgende Elemente in der Spezifikationssprache enthalten sein:

- Definition derjenigen Symbole, die Tokens repräsentieren
- Definition des Start-Nichtterminals
- Produktionen
- Aktionen in Form von Programmfragmenten

Sehr nützlich sind weiterhin:

- Kommentare
- Alternativen in den rechten Seiten von Produktionen
- evtl. reguläre Ausdrücke über TUN in den rechten Seiten von Produktionen
- Spezifikation des Datentyps für "semantische Werte"
- Spezifikation der Berechnung von "semantischen Werten" in den Aktionen (Zugriff auf den zum Parse-Stack parallelen "Semantik-Stack")

4.2.2. Parser-Spezifikations-Sprache

< siehe folgende zwei Seiten >

2.4. Syntax of the Specification Language (LR Parsing)

```
specification     :  definitions SDEL rules E_O_F
                  |  definitions SDEL rules SDEL utilities E_O_F
                  ;

utilities         :  /* epsilon */
                  |  utilities PROG
                  ;

definitions       :  /* epsilon */
                  | definitions definition
                  ;

definition        :  TOKEN tokenlist
                  |  START IDENT
                  |  PROG
                  ;

tokenlist         :  IDENT
                  |  tokenlist IDENT
                  ;

rules             :  /* epsilon */
                  |  rules rule
                  ;

rule              :  IDENT COLON alternatives SEMIC
                  ;

alternatives      :  alternative
                  |  alternatives BAR alternative
                  ;

alternative       :  EMPTY
                  |  EMPTY PROG
                  |  sequence
                  |  sequence PROG
                  ;

sequence          :  IDENT
                  |  sequence IDENT
                  ;
```

2.3. Lexical Analysis

Comments are enclosed in /* and */ and may span several lines. Comments do not nest. Blanks, tabs, returns, newlines, and comments are skipped during lexical analysis.

Identifiers (terminals as well as nonterminals) must begin with a letter, and consist of letters and digits. The underscore counts as a letter.

Program fragments are enclosed in %{ and %}. They do not nest. Furthermore, they are immediately copied to the generated parser during lexical analysis of the specification (so that there is no danger of overflowing the generator's matching buffer).

For the tokens and attributes returned by the scanner of the parser generator see the following table:

Lexeme	Token	Attribute
blank	-	-
tab	-	-
return	-	-
newline	-	-
/* comment */	-	-
EOF	E_O_F	-
%%	SDEL	-
%{ program text %}	PROG	-
%token	TOKEN	-
%start	START	-
%empty	EMPTY	-
letter, followed by letters or digits	IDENT	pointer to name
:	COLON	-
;	SEMIC	-
		BAR	-

4.3. Ad-hoc Parser für den Bootstrap

Solche Parser lassen sich am einfachsten als Top-Down-Recursive-Descent-Parser realisieren. Notwendige Schritte:

- Grammatik in LL(1)-Form bringen
- für jedes Nichtterminalsymbol: rechte Seiten zusammenfassen
- genau eine Funktion für jedes Nichtterminalsymbol schreiben (die Alternativen auf der rechten Seite sind Verzweigungen im Kontrollfluß)
- evtl. mehrere Funktionen zu einer zusammenfassen
- jede Funktion berechnet ihren semantischen Wert aus den semantischen Werten der von ihr gerufenen Funktionen

Bem.: Die hauptsächlich benötigten Grammatiktransformationen sind "Linksfaktorisierung" und "Elimination von Linksrekursion":

a) $A \to \beta \gamma_1 \mid \beta \gamma_2 \quad \rightsquigarrow \quad A \to \beta A'$

$\qquad A' \to \gamma_1 \mid \gamma_2$

b) $A \to A\beta \mid \gamma \quad \rightsquigarrow \quad A \to \gamma A'$

$\qquad A' \to \beta A' \mid \varepsilon$

Bsp.: Additive Ausdrücke im Demo-Programm

```
expr : term
     | expr PLUS term
     | expr MINUS term
     ;
```

Erinnerung:

- Operatorprioritäten realisiert durch Grammatik-"Stufen"
- Operatorassoziativitäten realisiert durch Links/Rechts-Rekursion

Transformation:

```
expr        : term expr-cont
            ;

expr-cont   : PLUS term expr-cont
            | MINUS term expr-cont
            | /* empty */
            ;
```

Diese zwei Produktionen werden als zwei Funktionen programmiert, die man zusammenfassen kann:

```
expr() {
    term();
    while (token == PLUS || token == MINUS) {
        token = nextToken();
        term();
    }
}
```

```
/* parser */


void parseList(void);
SemanticValue parseAssign(void);
SemanticValue parseExpr(void);
SemanticValue parseTerm(void);
SemanticValue parseFactor(void);
SemanticValue parsePrimary(void);


int lookahead;


void syntaxError(char *msg) {
  printf("syntax error: %s\n", msg);
  exit(99);
}


void parseList(void) {
  SemanticValue value;

  lookahead = nextToken();
  while (lookahead != END_OF_INPUT) {
    if (lookahead == NEWLINE) {
      lookahead = nextToken();
    } else {
      value = parseAssign();
      if (lookahead != NEWLINE) {
        syntaxError("newline expected");
      }
      printf("\t%.8g\n", value.value);
    }
  }
}


SemanticValue parseAssign(void) {
  SemanticValue value;

  value = parseExpr();
  while (lookahead == STORE) {
    lookahead = nextToken();
    if (lookahead != VAR) {
      syntaxError("variable expected");
    }
    memory[tokenValue.index] = value.value;
    lookahead = nextToken();
  }
  return value;
}


SemanticValue parseExpr(void) {
  SemanticValue value;
  SemanticValue aux;

  value = parseTerm();
  while (lookahead == PLUS || lookahead == MINUS) {
    if (lookahead == PLUS) {
      lookahead = nextToken();
      aux = parseTerm();
      value.value += aux.value;
    } else
    if (lookahead == MINUS) {
      lookahead = nextToken();
      aux = parseTerm();
```

```
      value.value -= aux.value;
    }
  }
  return value;
}


SemanticValue parseTerm(void) {
  SemanticValue value;
  SemanticValue aux;

  value = parseFactor();
  while (lookahead == STAR || lookahead == SLASH) {
    if (lookahead == STAR) {
      lookahead = nextToken();
      aux = parseFactor();
      value.value *= aux.value;
    } else
    if (lookahead == SLASH) {
      lookahead = nextToken();
      aux = parseFactor();
      if (aux.value == 0.0) {
        printf("division by zero\n");
        exit(99);
      }
      value.value /= aux.value;
    }
  }
  return value;
}


SemanticValue parseFactor(void) {
  SemanticValue value;

  if (lookahead == PLUS) {
    lookahead = nextToken();
    value = parseFactor();
    return value;
  }
  if (lookahead == MINUS) {
    lookahead = nextToken();
    value = parseFactor();
    value.value = -value.value;
    return value;
  }
  value = parsePrimary();
  return value;
}


SemanticValue parsePrimary(void) {
  SemanticValue value;

  if (lookahead == NUMBER) {
    value.value = tokenValue.value;
    lookahead = nextToken();
    return value;
  }
  if (lookahead == VAR) {
    value.value = memory[tokenValue.index];
    lookahead = nextToken();
    return value;
  }
  if (lookahead == LPAREN) {
    lookahead = nextToken();
    value = parseExpr();
    if (lookahead != RPAREN) {
```

```
      syntaxError("right parenthesis expected");
    }
    lookahead = nextToken();
    return value;
  }
  syntaxError("number, variable, or left parenthesis expected");
  /* never reached */
  return value;
}
```

Pg