# Die Spezifizierungssprache GIPSY/L

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A. Helbling, T. Murer

# Die Spezifizierungssprache GIPSY/L

TIK-Report
Nr. 13, Mai 1995

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

A. Helbling, T. Murer
Die Spezifizierungssprache GIPSY/L
Mai 1995
Version 1
TIK-Report Nr. 13

---

Computer Engineering and Networks Laboratory,
Swiss Federal Institute of Technology (ETH) Zurich

Institut für Technische Informatik und Kommunikationsnetze,
Eidgenössische Technische Hochschule Zürich

Gloriastrasse 35, ETH-Zentrum, CH-8092 Zürich, Switzerland

---

# 1 Inhaltsverzeichnis

# 2 Einleitung

Mit der Spezifizierungssprache GIPSY/L können formale Sprachen mit Hilfe Erweiterbarer Attributierter Grammatiken [MM92] definiert werden. In GIPSY/L verfasste Dokumente werden vom Werkzeug GIPSY [Hel95] als Eingabe für die Erzeugung von integrierten Umgebungen [Hel95] benutzt.

GIPSY/L basiert auf Cocol [Möss90], der Beschreibungssprache des Compiler-Generators Coco/R [Möss90]. Da von GIPSY als Zielsprache Oberon verwendet wird, ist die Beschreibung des Attributierungssystems praktisch identisch mit den jeweils äquivalenten Konstrukten der Sprache Oberon.

Scanner und Parser werden mit GIPSY/L in verschiedenen Dokumenten spezifiziert und von GIPSY getrennt erzeugt. Zusammen mit dem umgebungsunabhängigen GIPSY-Kern [Hel95] entsteht daraus eine integrierte Umgebung für die spezifizierte Sprache.

In der Scanner-Spezifikation wird die lexikalische Ebene einer formalen Sprache definiert. Die syntaktische Ebene baut auf der Scanner-Definition auf und wird in Erweiterbaren Attributierten Grammatiken (Parser-Spezifikation) angegeben. Mit der Möglichkeit, zu den Nonterminalsymbolen Attribute und Attributierungsprozeduren zu definieren, kann die Sprach-Definition um Kontextbedingungen erweitert werden, welche die Sprache weiter einschränken. Die Attributierungsprozeduren können auch als semantische Aktionen eingesetzt werden, die den Eingabestrom verarbeiten.

Abb. 1 veranschaulicht die einzelnen Ebenen einer Sprachdefinition anhand einer Sprache für einfache mathematische Ausdrücke.

Am Anfang steht der Eingabestrom als Zeichenfolge, wie sie vom Benutzer eingeben worden ist. Der Scanner überliest alle als überflüssig definierten Zeichen (z.B. Leerzeichen), überprüft den Eingabestrom auf lexikalische Fehler und bildet eine Token-Liste.

Der Parser überprüft die Syntax und erzeugt aus der Token-Liste einen Struktur-Baum, falls keine syntaktischen Fehler aufgetreten sind.

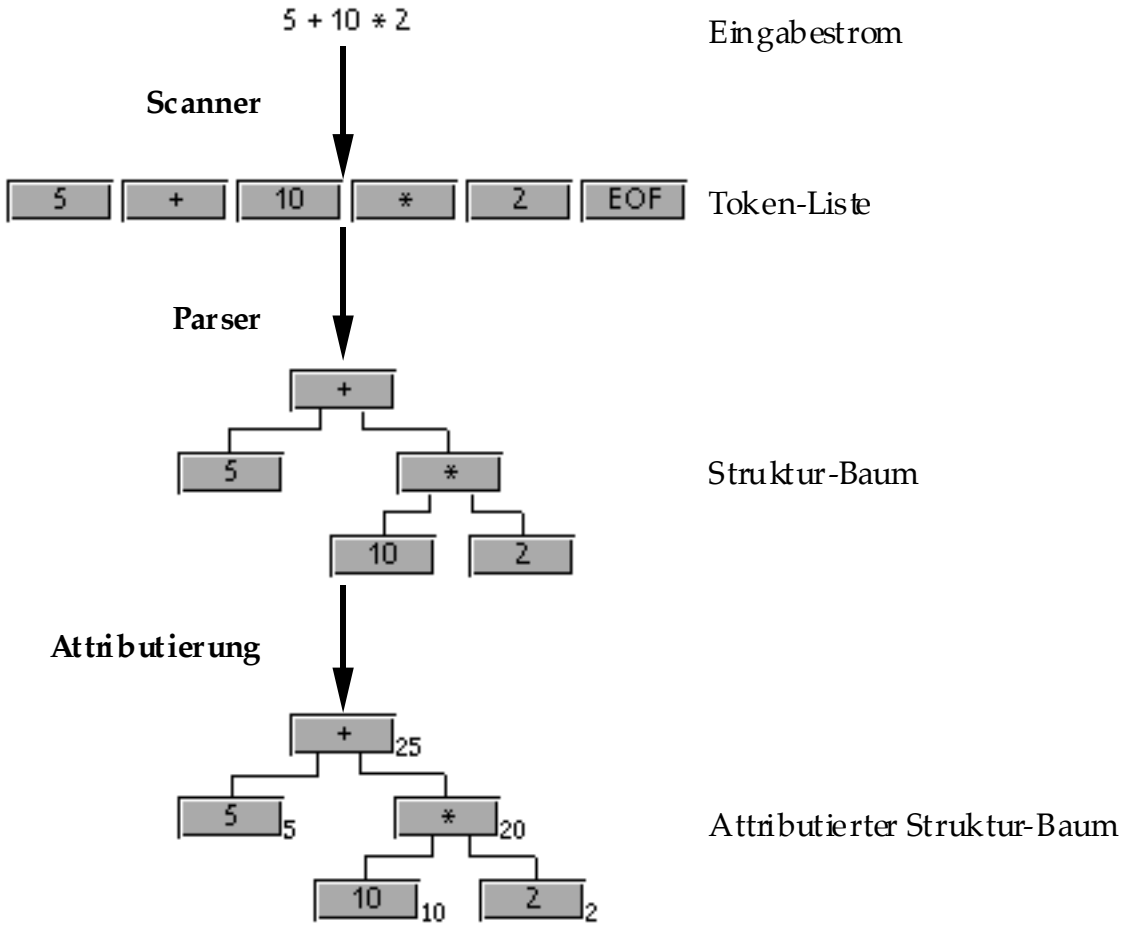


Abb. 1: Ebenen einer Sprachdefinition

Durch die Attributierung werden schliesslich die Attribute jedes Knotens des Strukturbaumes berechnet. Im Beispiel aus Abb. 1 wird der Ausdruck ausgewertet.

Mit Hilfe der Attributierungsprozeduren eines Nonterminalsymbols wäre es hier durchaus auch möglich, den Eingabestrom noch weiter einzuschränken (Kontextbedingungen). Es könnten z.B. nur Ausdrücke zugelassen werden, die zu einem positiven Resultat führen.

# 3 Scanner-Spezifikation

Die Aufgabe eines Scanners besteht im Aufspalten des Eingabestromes (Zeichenfolge) in eine Folge von Tokens (Terminalsymbole) und im Erkennen und Anzeigen von lexikalischen Fehlern.

In einer Scanner-Spezifikation werden die folgenden lexikalischen Elemente einer formalen Sprache deklariert:

- Zeichenmengen
- Menge der zu überlesenden Zeichen
- Klassen von Terminalsymbolen
- Kommentar
- Terminalsymbole (Literale und Schlüsselworte)

Als Grund-Alphabet aller in einer Spezifikation vorkommenden Elemente dient der ASCII-Zeichensatz. Neben Zeichen werden auch beliebige Objekte (Gadgets im Oberon System3) zugelassen, die durch Angabe ihres Typs identifiziert werden können.

Die Terminalsymbole, die in den Produktionen einer Attributierten Grammatik als Literale oder Schlüsselworte vorkommen können, müssen bereits in der Scanner-Spezifikation aufgeführt werden. So können Scanner und Parser unabhängig voneinander generiert werden, da aus Erfahrung der Parser viel häufiger erzeugt werden muss. Zudem kann der gleiche Scanner für mehrere (erweiterte) Attributierte Grammatiken wiederverwendet werden.

Der zugehörige Ausschnitt aus der EBNF-Syntax zeigt den Aufbau einer Scanner-Beschreibung.

```
Scanner      ::= 'SCANNER' ident ';' Declaration 'END' ident '.'.
Declaration  ::= 'CHARACTERS' SetDecl {SetDecl}
                 ['IGNORE' Set ]
                 'TOKENS' TokenDecl {TokenDecl}
                 ['COMMENT' 'FROM' string 'TO' string ['NESTED']]
                 [LITERALS Literal {',' Literal}].
```

Eine Scanner-Spezifikation beginnt mit dem Schlüsselwort SCANNER und einem Bezeichner zur Identifikation des Scanners. Darauf folgen die einzelnen Bestandteile einer Scanner-Beschreibung (Zeichenmengen, Ignore-Menge, Klassen

von Terminalsymbolen, Terminalsymbole, Kommentar), die in den nachfolgenden Abschnitten näher erklärt werden.

## 3.1 Zeichenmengen

Die Definition von Zeichenmengen dient als Grundlage für die Beschreibung von Terminal-Klassen. Anstatt wiederholt die einzelnen Zeichen und Zeichenbereiche aufführen zu müssen, kann dafür ein aussagekräftiger Bezeichner definiert werden.

```
Declaration   ::= 'CHARACTERS' SetDecl {SetDecl} …
SetDecl       ::= ident '=' Set '.'.
Set           ::= BasicSet {('+' | '-') BasicSet}.
BasicSet      ::= ident | '{' [Range {',' Range}] '}'.
Range         ::= (char | charConst) ['..' (char | charConst)] | string.
```

Die Zeichenmengen können durch Vereinigung und Differenz zweier Mengen zusammengesetzt werden. Mit dem Symbol '..' können ganze Zeichenbereiche in eine Zeichenmenge aufgenommen werden.

Beispiele von Zeichenmengen:

```
CHARACTERS
   digit       = {'0'..'9'}.
   smallLetter = {'a'..'z'}.
   capLetter   = {'A'..'Z'}.
   letter      = smallLetter + capLetter.
   blank       = {' '}.
   button      = {BasicGadgets.Button}.
```

Als Zeichenmenge können auch Gadgets deklariert werden. In diesem Falls wird die Menge durch Angabe von „Modulname.NewProzedur" spezifiziert.

Das Zeichen für Datei-Ende (Eof) besitzt im Oberon System den festen Wert 0X. In GIPSY/L muss daher Wert des Eof-Zeichens ebenfalls mit 0X definiert werden.

## 3.2 Menge der zu überlesenden Zeichen

In fast jeder Sprache befinden sich Zeichen, welche die einzelnen Tokens abschliessen bzw. voneinander trennen und so die Lesbarkeit verbessern, sonst aber keinen Einfluss auf den Inhalt ausüben. Meistens wird dafür das Leerzeichen *(blank)* verwendet.

```
Declaration   ::= … ['IGNORE' Set ] …
```

Als Elemente der IGNORE-Menge können vorher deklarierte Zeichenmengen angegeben werden.

Beispiel:

```
IGNORE
   {blank}
```

## 3.3 Klassen von Terminalsymbolen (Terminal-Classes)

In GIPSY/L wird die Menge der gültigen Zeichenfolgen durch Terminalklassen und Literale (Terminalsymbole) angegeben. Zu einer Terminalklasse (z.B. *ident* für Bezeichner) können mehrere Wörter, die alle als das gleiche Terminalsymbol erkannt werden, gehören. Typischerweise können diese Mengen unendlich viele Elemente enthalten.

```
Declaration   ::= … 'TOKENS' TokenDecl {TokenDecl} …
TokenDecl     ::= ident '=' TokenExpr | 'GADGET' ident '.' ident | anyGadget | '@gadget' '.'.
TokenExpr     ::= TokenTerm {'|'TokenTerm}.
TokenTerm     ::= TokenFactor {TokenFactor} [Context].
TokenFactor   ::= ident | char | string | '(' TokenExpr ')' | '[' TokenExpr ']' | '{' TokenExpr '}'.
Context       ::= 'CONTEXT' '(' TokenExpr ')'.
```

Die rechte Seite der Definition einer Terminalklasse enthält die Struktur der zu definierenden Wörter in einem regulären Ausdruck in EBNF-Notation.

Beispiel für Terminalklassen:

```
TOKENS
   ident      = letter {letter | digit}.
   integer    = digit {digit}.
```

Für gewisse Terminalklassen reicht es nicht, bei der Wortanalyse nur ein Zeichen vorauszuschauen. So erkennt ein Scanner z. B. die Zeichenfolge '0..10' als eine Fliesspunkt-Zahl, gefolgt von einem Punkt und einer Ganzzahl, obwohl dieser Ausdruck auch als ein Zahlenbereich angesehen werden kann. Diese Unterscheidung wird möglich, wenn die Definition der Terminalklasse *integer* um eine Kontextdeklaration erweitert wird.

```
integer    = digit {digit} | digit {digit} CONTEXT('..').
```

## 3.4 Terminalsymbole (Literale)

Ein Terminalsymbol (Literal) definiert ein gültiges Wort, das in den Attributierten Grammatiken, die diesen Scanner benutzen, vorkommen kann.

```
Declaration   ::= … LITERALS Literal {',' Literal}.
Literal       ::= char | string.
```

Einzelne Zeichen oder Schlüsselworte werden zwischen Hochkommata oder Apostrophen angegeben und durch Kommata voneinander getrennt.

Beispiele einer Terminalsymbol-Deklaration:

```
LITERALS
   'BEGIN', 'END', '+', '-'
```

Oft sind Terminalsymbole auch Elemente einer Tokenklasse (z.B. *BEGIN* passt in die Tokenklasse *ident*), ohne jedoch ihre Eigenständigkeit zu verlieren.

## 3.5 Kommentare

In GIPSY/L können beliebige, bis zu acht Zeichen lange Kommentar-Begrenzer gewählt werden. Dabei kann auch angegeben werden, ob verschachtelte Kommentare erlaubt sind oder nicht.

```
['COMMENT' 'FROM' string 'TO' string ['NESTED']]
```

Die Kommentar-Möglichkeit ist ausgeschaltet, wenn in der Spezifikation keine Kommentar-Deklaration aufgeführt wird.

Beispiel (Oberon-Kommentar)

```
COMMENT FROM '(*' TO '*)' NESTED
```

## 3.6 Beispiel einer Scanner-Spezifikation

Als Beispiel für eine GIPSY/L-Spezifikation ist hier eine Scanner-Beschreibung für die Behandlung von Zahlen und Operationen angegeben. Die Spezifikation enthält folgende Konstrukte:

- Bezeichner (ident), Ganzzahlen (integer) und Fliesspunkt-Zahlen (real)
- Literale (+,-,*,/), Schlüsselworte (DIV, MOD)
- Kommentar ist eingeschaltet: jeder Textausschnitt zwischen (* und *) wird vom Scanner überlesen.

```
SCANNER CalcScanner;

  CHARACTERS
    letter  = {'A'..'Z','a'..'z'}.
    digit   = {'0'..'9'}.

  IGNORE
    {blank}

  TOKENS
    ident   = letter {letter | digit}.
    integer = digit {digit}.
    real    = digit {digit} ['.' digit {digit}].

  COMMENT FROM '(*' TO '*)' NESTED

  LITERALS
    '+', '-', '*', '/', 'DIV', 'MOD'

END CalcScanner.
```

Aufgrund dieser Spezifikation werden für die einzelnen Terminalklassen folgende Wörter als gültig bzw. ungültig erkannt.

| Terminalklasse | Gültige Wörter | Ungültige Wörter |
|---|---|---|
| Bezeichner (ident) | name, Name, a1, ab103 | 1name, name_1 |
| Ganzzahl (integer) | 12, 15, 0 | -10, 1..10 |
| Fliesspunktzahl (real) | 3.147, 0.002 | -0.3, 3.3E10 |

Tab. 1: Beispiele von gültigen und ungültigen Symbolen aus den Terminalklassen der Grammatik *CalcScanner*

# 4 Attributierte Grammatiken (Parser-Spezifikation)

In GIPSY/L wird die Syntax und Semantik einer Sprache durch Attributierte Grammatiken definiert. Eine Attributierte Grammatik ist eine kontextfreie Grammatik, deren Nonterminalsymbole erweitert sind um

- **Attribute** und
- **Attributierungsprozeduren** zum Berechnen der Attribute (semantische Aktionen).

Eine Attributierte Grammatik ist in GIPSY/L ähnlich aufgebaut wie ein Modul in Oberon. Eine Import-Deklaration ermöglicht die Verwendung von externen Prozeduren in den Attributierungsprozeduren. Daneben können auch globale Konstanten, Typen, Variablen und Prozeduren deklariert und in den Attributierungsprozeduren benutzt werden. Zur Initialisierung einer Grammatik kann am Ende eine Anweisungsfolge angegeben werden.

```
Gram        ::= 'GRAMMAR' ident ('(' ident ')'] | ';''SCANNER' ident) ';'
                ['IMPORT' Import {',' Import} ';']
                DeclSeq
                [Skip]
                Prod {Prod}
                ['BEGIN' StatSeq] 'END' ident '.'.
```

Zusätzlich enthält eine Attributierte Grammatik auch Konstrukte, die zur Definition der Syntax und Semantik einer Sprache notwendig sind. Mit der Anweisung

```
SCANNER = ident;
```

wird der Scanner festgelegt, der für die Sprache verwendet werden soll, die in dieser Grammatik definiert wird.

Danach folgt schliesslich die Definition der Nonterminalsymbole mit den Attributen und Attributierungsprozeduren. Das an erster Stelle aufgeführte Nonterminalsymbol gilt als **Start-Nonterminalsymbol**.

```
Prod        ::= 'NONTERMINAL' NTObj ('::=' Expr'.' |'(' ident'.' ident')')
                [Attributes] {'PROCEDURE' ProcDecl}.
NTObj       ::= ident.
Attributes  ::= 'ATTR' VarDecl ';' {VarDecl ';'}.
```

```
Expr        ::= Term {' | ' Term}.
Term        ::= Factor {Factor}.
Factor      ::= Symbol | '(' Expr ')' | '[' Expr ']' | '{' Expr '}'.
Symbol      ::= ident | char | keyword.
```

## 4.1 Skip-Menge

In der Skip-Menge können Tokens angegeben werden, die zwar vom Scanner erkannt werden, aber für die Semantik keine Bedeutung haben sollen und daher vom Parser überlesen werden. Die Formatier-Zeichen für Zeilenende (*eol*) und Tabulator (*tab*) befinden sich meistens in dieser Menge.

```
Skip        ::= 'SKIP' ident {ident} ';'.
```

Als Elemente der Skip-Menge können vorher deklarierte Zeichenmengen angegeben werden.

Beispiel:

```
SKIP
   tab, eol;
```

## 4.2 Deklaration von Nonterminalsymbolen

Eine Nonterminal-Deklaration beginnt mit dem Schlüsselwort NONTERMINAL. Zum nachfolgenden Namen des Nonterminalsymbols können in Klammern ein Basis-Nonterminalsymbol (vgl. Erweiterbare Attributierte Grammatiken) und die persistenten Attribute deklariert werden. Die Produktionsregel wird durch einen kontextfreien Ausdruck in EBNF-Notation festgelegt. Jede Nonterminaldeklaration besitzt einen eigenen Gültigkeitsbereich, in dem ihre lokalen Objekte (Attribute und Attributierungsprozeduren) sichtbar sind.

Da jedes Nonterminalsymbol bei der Übersetzung von GIPSY in einen Oberon-Typ abgebildet wird, können in den Attributierungsprozeduren zu allen Nonterminalsymbolen Objekte deklariert und verwendet werden. Die Attribute eines Nonterminalsymbols verhalten sich dann wie Datenfelder eines Objektes und die Attributierungsprozeduren können wie Methoden aufgerufen werden.

Beispiel:

```
NONTERMINAL AddOp ::= '+' | '-'.
```

```
PROCEDURE Eval(SELF: AddOp; x,y: LONGINT): LONGINT;
   VAR term: GIPSYSpecs.TERM; str: ARRAY 32 OF CHAR;
BEGIN
   term:=GIPSYSpecs.FIRSTTERM(SELF);
   GIPSYSpecs.GETSTRING(term,str);
   IF str[0]='+' THEN
      RETURN x+y
   ELSIF str[0]='-' THEN
      RETURN x-y
   ELSE
      RETURN 0
   END
END Eval;
```

## 4.3 Attribute und Attributierungsprozeduren

### 4.3.1 Handler-Konzept

Die Unterstützung von Command-Dispatch und die Verwaltung von Attributen wird nicht automatisch von GIPSY generiert, sondern muss explizit im Handler eines Nonterminalsymbols programmiert werden. Dadurch vereinfacht sich nicht nur die Generierung für GIPSY, sondern dem Entwickler werden ein offenes, unbeschränktes Konzept und viele Freiheiten in der Geststaltung der Schnittstelle zum Anwender angeboten.

**Handler**

Die Handler-Prozedur eines Nonterminalsymbols muss den Namen *handle* besitzen, damit auch der vom Oberon-System vordefinierte Handler des Objektes überschrieben wird. Zum Überschreiben der handle-Prozeduren muss das Zeichen '*' zwischen dem Schlüsselwort PROCEDURE und dem Prozedurnamen eingefügt werden. Der Aufruf des geerbten Handlers erfolgt nicht automatisch, sondern muss aus dem Grammatiknamen der Basisgrammatik und dem Namen des Nonterminalsymbols zusammengesetzt werden (vgl. 4.3.4).

**Command-Message**

Die *Structures.CommandMsg* ist nur für das Startnonterminalsymbol einer Grammatik von Interesse. Sie dient zur Verwaltung von Commands (Attributierungsprozedur des Startnonterminalsymbols) und als Schnittstelle zum GIPSY-Kern und zum Anwender.

```
CommandMsg* = RECORD (Objects.ObjMsg)
    id*: INTEGER; (* do, enum, requires *)
    cmd*: Objects.Name;
    res*: INTEGER;
    Enum*: PROCEDURE (obj: Objects.Object; cmd: ARRAY OF CHAR);
    obj*: Objects.Object          (* Parameter für Enum-Prozedur *)
END;
```

*Msg.id=Structures.do*

Ausführen des Commands Msg.cmd

M.res zeigt an, ob der Command ausgeführt wurde

*Msg.id=Structures.enum*

Führt für jeden Command die Prozedur Msg.Enum aus. Msg.Enum besitzt im Gegensatz zu Objects.AttrMsg.Enum-Prozedur einen Parameter obj vom Typ Objects.Object, da meistens der Command-Name allein nicht viel nützt.

*Msg.id=Structures.requires*

Führt für alle von einem Command vorausgesetzten Command-Namen die Enum-Prozedur aus.

**Attribut-Message**

Die *Structures.AttrMsg* ist eine Erweiterung von *Objects.AttrMsg*. Sie besitzt zusätzlich ist das Feld obj (Objects.Object, mit dem beliebige Objekte (Msg.class=Structures.object) als Attribute verwaltet werden können

```
AttrMsg* = RECORD (Objects.AttrMsg)
  (* id Objects.enum, Objects.get, Objects.set *)
    obj*: Objects.Object    (* Feld für beliebige Objekte (AttrMsg.class=object) *)
END;
```

*Msg.id=Objects.get/ Msg.id=Objects.set*

Das Schreiben und Lesen eines Attributes mit dem Namen Msg.name verhält sich gleich bei der Oberon-AttrMsg.

Mit M.class=Structures.object können auch Attribute mit nicht-Standard-Typen bei einer Rekompilation gerettet werden; natürlich nur, falls sie nicht Verweise auf Knoten des ungültig gewordenen Baumes enthalten.

*Msg.id=Objects.enum*

Die Aufzählung der Attribut-Namen folgt einem anderen Verhalten als die Standard-Enum-Meldungen im Oberon-System.

Am Anfang enthält das Feld 'name' der Meldung den leeren String. Bei jedem Aufruf wird dann 'name' mit dem nächsten Attributnamen und 'class' mit der zugehörigen Typen-Klasse aufgefüllt. 'res' zeigt an, wann die Aufzählung beendet ist.

Beispiel:

```
Msg.name:=''; Msg.id=Objects.enum;
syn.handle(syn,Msg);
WHILE Msg.res=0 DO
    ... Msg.name / Msg.class ...
    syn.handle(syn,Msg);
END
```

**Beispiel: Handler für das Startnonterminalsymbol der Grammatik Calc**

Die Grammatik Calc bietet die zwei Commands *CalcRes* und *AddRes* an. Das Startnonterminalsymbol enthält das Attribut *res*, in dem das Resultat der Berechnungen festgehalten ist. Dieses Attribut kann vom Anwender mit dem Attribut-Inspektor eingesehen und verändert werden.

```
NONTERMINAL Calc ::= Expression.
    ATTR res: LONGINT;

    PROCEDURE handle*(obj: Objects.Object; VAR M: Objects.ObjMsg);
        VAR SELF: Calc;
    BEGIN
        SELF:=obj(Calc);
        WITH M: Structures.CommandMsg DO
            M.res:=0;
            IF M.id=Structures.do THEN
                IF M.cmd='CalcRes' THEN SELF.CalcRes(SELF)
                ELSIF M.cmd='AddRes' THEN SELF.AddRes(SELF)
                ELSE Structures.SynNodeHandle(obj,M)
                END
            ELSIF M.id=Structures.enum THEN
                Structures.SynNodeHandle(obj,M);
                M.Enum(M.obj,'CalcRes'); M.Enum(M.obj,'AddRes')
            ELSIF M.id=Structures.requires THEN
                M.Enum(M.obj,Structures.CheckSyntaxCmd)
            ELSE Structures.SynNodeHandle(obj,M)
            END
        |M: Structures.AttrMsg DO
            IF M.id=Objects.enum THEN
                IF M.name='' THEN M.name:='res'; M.class:=Objects.Int; M.res:=0
                ELSE Structures.SynNodeHandle(obj,M)
```

```
            END
         ELSIF M.id=Objects.get THEN
            IF M.name='res' THEN M.class:=Objects.Int; M.i:=SELF.res; M.res:=0
            ELSE Structures.SynNodeHandle(obj,M)
            END
         ELSIF M.id=Objects.set THEN
            IF M.name='res' THEN SELF.res:=M.i; M.res:=0
            ELSE Structures.SynNodeHandle(obj,M)
            END
         ELSE Structures.SynNodeHandle(obj,M)
         END
      ELSE Structures.SynNodeHandle(obj,M)
      END
   END handle;
```

### 4.3.2 Attribute

Die Deklaration der Attribute eines Nonterminalsymbols entspricht einer Variablen-Deklaration in Oberon. Eingeleitet durch das Schlüsselwort ATTR können als Attribute Bezeichner mit allen gültigen Datentypen verwendet werden. Auf die Attribute selber kann wie auf ein Record-Feld zugegriffen werden.

Zur Initialisierung der Attribute kann dem Nonterminalsymbol eine *Structures.AttrMsg* mit der Bezeichung *Structures.init* geschickt werden. Falls das Nonterminalsymbol Attribute besitzt, die initialisiert werden können, so wird in seinem Handler diese Meldung abgefangen und die Initialisierung ausgelöst.

Beispiel:

```
NONTERMINAL Gram ::= 'GRAMMAR' ident ';' Prod {Prod} 'END' ident '.'.
   ATTR objTree: Trees.Tree; symTab: SymbolTable.SymbolTable;

   PROCEDURE InitAttrs(SELF: Gram);
   BEGIN
      SELF(Gram).objTree:=NIL;
      NEW(SELF(Gram).symTab);
      SymbolTable.InitSymbolTable(SELF(Gram).symTab);
      SynSpecs.SetSymTab(SELF(Gram).symTab)
   END InitAttrs;

   PROCEDURE handle*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   BEGIN
      WITH …
      |M: Structures.AttrMsg DO
         IF M.id=Structures.init THEN obj(Gram).InitAttrs(obj(Gram))
         ELSIF …
      |M: … DO
      …
      …
   END handle;
```

Für das Abspeichern und das Laden der einzelnen Attribute ist die im Oberon-System vordefinierte *FileMsg* aus dem Modul Objects zuständig. Durch die Interpretierung der *FileMsg* im Handler eines Nonterminalsymbols werden die gewünschten Attribute beim Speichern des Textes mitgeschrieben und sind nach einem späteren Laden wieder im gleichen Zustand vorhanden.

Beispiel:

```
NONTERMINAL Expression ::= Term {AddOp Term}.
   ATTR res: INTEGER;

   PROCEDURE handle*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   BEGIN
      WITH …
      |M: Objects.FileMsg DO
         IF M.id=Objects.load THEN
            Files.ReadInt(M.R,obj(Expression).res)
         ELSE
            Files.WriteInt(M.R,obj(Expression).res)
         END
      |M: … DO
      …
      …
   END handle;
```

### 4.3.3 Attribute, die den Syntaxbaum überleben

Durch jede Änderung im Text (Editieroperation) oder einen Syntax-Check wird der zugehörige Syntax-Baum ungültig und während dem nächsten Parsen wird ein neuer Baum aufgebaut. Da keine inkrementelle Compilation stattfindet und die Knoten nicht wiederverwendet werden, gehen mit dem Syntax-Baum auch alle Attribute verloren, deren Werte in den einzelnen Knoten gespeichert sind. Um diese Unstimmigkeit zu beheben, sehen GIPSY-Dokumente die Möglichkeit vor, im Text Attribut-Gadgets [Hel95] einfliessen zu lassen. Diese tragen die Information über den Inhalt der Attribute, auch wenn der Strukturbaum nicht vorhanden ist. Beim Neuaufbau des Baumes werden sie gemäss ihrer Position im Text wieder dem nächsten passenden Knoten zugeweisen. Somit helfen diese Attribute, die 'baumlose' Zeit zu überbrücken.

Zusätzlich besteht noch die Möglichkeit, mit dem Attribut-Inspektor [Hel95] diese Attribute einzusehen oder zu verändern.

Diese Attribut-Gadgets werden aufgrund der Information aus dem Handler eines Nonterminalsymbols automatisch erzeugt und sind nur für Attribute mit Standardtypen erhältlich. Dazu gehören die Typen BOOLEAN, INTEGER, LONGINT, REAL und LONGREAL. Für die Darstellung von Strings kann der Typ *GIPSYSpecs.String* aus einem Modul des GIPSY-Kerns verwendet werden. Im Handler kann auch bestimmt werden, welche Attribute überhaupt vom Anwender verändert werden können und welche 'nur' ihre Werte retten.

### 4.3.4 Attributierungsprozeduren

In den Attributierungsprozeduren der Nonterminalsymbole wird die Semantik einer Sprache festgelegt.

Für die Implementation der Attributierungsprozeduren gilt die Oberon-Syntax. Zur Programmierung des Attributierungssystem stehen das Modul *GIPSYSpecs* und einige Standardfunktionen zur Verfügung.

Bei der Übersetzung der Attributierungsprozeduren wird die objektorientierte Programmierung mit Prozedurvariablen (Standard-Oberon; nicht Oberon-2) unterstützt, was folgende Konsequenzen mit sich bringt:

- Das Objekt, dessen Methode aufgerufen wird, muss in der Parameterliste mitgegeben werden, wenn man innerhalb der Attributierungsprozedur darauf zugreifen will.
- Beim Überschreiben von Attributierungsprozeduren in erweiterten Grammatiken darf die Definition der Prozedur (Parameterliste) nicht verändert werden und das Zeichen '*' muss zwischen dem Schlüsselwort PROCEDURE und dem Prozedurnamen eingefügt werden. Innerhalb der Attributierungsprozedur kann dann auf das erweiterte Objekt nur mit einem Type-Guard zugegriffen werden.
- Der direkte Aufruf einer überschriebenen Attributierungsprozedur ist wegen der Verwendung von Prozedurvariablen nicht möglich. Die überschriebene Attributierungsprozedur kann aber trotzdem erreicht werden, falls die Prozedur nach folgendem Schema aufgerufen wird:

**Gram**Strucs.**ProcNonterm**(params);

Entschlüsselung der variablen Bestandsteile des Aufrufes:
**Gram**: Grammatikname, in der das Nonterminalsymbol definiert ist
**Proc**: Name der Attributierungsprozedur
**Nonterm**: Name des Nonterminalsymbols

Beispiel:

Deklaration eines Nonterminalsymbols *Object* mit einer Attributierungsprozedur in einer Basis-Grammatik G:

```
NONTERMINAL Object::= ident;
    ATTR x: INTEGER;

    PROCEDURE Do(Self: Object; a,b: INTEGER);
    BEGIN
        Self.x:=a+b
    END Do;
```

Der Aufruf für diese Attributierungsprozedur lautet: obj.Do(obj,params)

Erweiterung des Nonterminalsymbols *Object* um ein Attribut und Überschreiben einer Attributierungsprozedur in einer von G erweiterten Grammatik G':

```
NONTERMINAL Object (G.Object);
    ATTR y: INTEGER;

    PROCEDURE * Do(Self: G.Object; a,b: INTEGER);
    BEGIN
        GStrucs.DoObject(Self,a,b);
        IF Self IS Object THEN
            Self(Object).y:=Self.x+a*b
        END
    END Do;
```

### 4.3.5 Attributierungsprozeduren des Start-Nonterminalsymbols

Die Attributierungsprozeduren des Start-Nonterminalsymbols werden in den erzeugten Dokumenten als Einstieg für die verschiedenen Attributierungsprozesse verwendet. Mit der Structures.CommandMsg kann gesteuert werden, welche Attributierungsprozeduren unter welchem Namen in einem GIPSY-Dokument als Eintrag in der Menu-Liste erscheinen und durch Auswählen ausgeführt werden können.

## 4.4 Erweiterbare Attributierte Grammatiken

Der Ausgangspunkt einer Grammatik-Hierarchie bildet eine Basis-Grammatik, die zusammen mit dem Scanner die Syntax einer Sprache definiert. Erweiterte Grammatiken enthalten keine neuen Syntaxregeln (Produktionen). Es können hingegen neue Attribute und Attributierungsprozeduren zu bestehenden Nonterminalsymbolen hinzugefügt werden.

Anwendung: Verarbeitung eines strukturierten Eingabe-Datenstroms, dessen Syntax in der Basisgrammatik und im Scanner festgelegt wird.

- Automatische Überprüfung der Syntax durch Regeln (Produktionen)
- Semantische Aktionen (Attributierungsprozeduren) zur Festlegung der Kontextbedingungen

Die Erweiterbarkeit von Attributierten Grammatiken ermöglicht:

- Aufteilung in zusammengehörende Teilaufgaben (Modularisierung)
- Integration (verschiedene Erweiterungen können auf ihre gleiche Basisgrammatik Bezug nehmen)

## 4.5 Kopplung zwischen Scanner und Parser

- Ein Scanner darf nur in nicht-erweiterten Grammatiken anschliessend an das Schlüsselwort SCANNER angegeben werden. In erweiterten Grammatiken kann kein neuer Scanner gewählt werden, sondern es wird der Scanner der Basisgrammatik übernommen.

```
SCANNER CalcScanner;
```

- Nach der Definition des Scanners können alle im Scanner definierten Terminalsymbole und Literale in Produktionen verwendet werden. In der Deklarationsanalyse der Parser-Generierung wird schliesslich überprüft, ob die verwendeten Terminalsymbole und Literale im angegeben Scanner spezifiziert worden sind.

## 4.6 Traversieren des Syntaxbaumes

Das Modul *GIPSYSpecs* dient als Schnittstelle zwischen GIPSY-Kern und den Attributierungsprozeduren in Attributierten Grammatiken und bietet vordefinierte Typen und Prozeduren zum Traversieren des Syntaxbaumes an. Das Modul muss in der Grammatik nicht importiert werden, aber die Typen und Prozeduren müssen qualifiziert angewandt werden.

Der Syntaxbaum, der vom GIPSY-Kern beim Parsen eines Dokumentinhaltes aufgebaut wird, bleibt immer solange erhalten, bis das Dokument verändert wird. Daher kann er mit Hilfe der Prozeduren aus dem Modul *GIPSYSpecs* und den Standardprozeduren der Syntaxbaum jederzeit traversiert werden. Dabei können die Attributierungsprozeduren eines beliebigen Knotens (Nonterminalsymbols) aufgerufen werden oder es können mit dessen Attributen Berechnungen durchgeführt werden.

```
DEFINITION GIPSYSpecs;

    IMPORT Obejcts, Structures, Tokens;

    CONST
        Undef=0; Before=1; AtStartOf=2; Inside=3; AtEndOf=4; Behind=5; Same=6;
        (* RelPosNT:  Undef, Before, AtStartOf, Inside, AtEndOf, Behind
           RelPosTerm:    Undef, Before, Same, Behind
        *)

    TYPE
        NONTERM = Structures.SynNode;
        TERM = Tokens.Token;
        TermClass = Objects.Name;
        Action = POINTER TO ActionDesc;
        ActionDesc = RECORD
            Open: PROCEDURE (context: NONTERM);
            DoToNode: PROCEDURE (obj: Objects.Object);
            Close: PROCEDURE (context: NONTERM)
        END;

    (* Nonterminal-Symbole *)
    PROCEDURE FIRSTNT(N: NONTERM): NONTERM;
    PROCEDURE LASTNT(N: NONTERM): NONTERM;
    PROCEDURE NEXTNT(N: NONTERM): NONTERM;
    PROCEDURE PREVNT(N: NONTERM): NONTERM;
    PROCEDURE EACH (N: NONTERM; action: Action);

    (* Terminal-Symbole *)
    PROCEDURE FIRSTTERM(N: NONTERM): TERM;
    PROCEDURE LASTTERM(N: NONTERM): TERM;
```

```
    PROCEDURE NEXTTERM(N: NONTERM; term: TERM): TERM;
    PROCEDURE GETSTRING(term: TERM; VAR string: ARRAY OF CHAR);

    PROCEDURE RELPOSNT(N: NONTERM; term: TERM): INTEGER;
    PROCEDURE RELPOSTERM(anchor,term: TERM): INTEGER;

END GIPSYSpecs.
```

### 4.6.1 NONTERM

Der Typ NONTERM stellt den Typ des abstrakten Nonterminalsymbols dar, von dem alle weiteren Nonterminalsymbole einer Grammatik implizit abgeleitet werden.

Die Operationen *FIRSTNT, LASTNT, NEXTNT* und *PREVNT* liefern zu einem Nonterminalsymbol das entsprechende erste, letzte, nächste bzw. vorherige Nonterminalsymbol oder NIL, falls kein solches Symbol existiert.

Mit *EACH* werden alle Nonterminalsymbole im Strukturbaum traversiert und das im Typ *Action* angegebene Verhalten wird auf jeden Knoten im Baum ausgeführt.

### 4.6.2 TERM

Variablen des Typs TERM repräsentieren Referenzen auf Terminalsymbole einer Grammatik. Zum Typ TERM gehören folgende Operationen

- *FIRSTTERM* liefert das erste Terminalsymbol eines Nonterminalsymbols.
- *LASTTERM* liefert das letzte Terminalsymbol eines Nonterminalsymbols.
- *NEXTTERM* liefert das auf ein Terminalsymbol nächstfolgende Terminalsymbol.
- *GETSTRING* liefert die Zeichenfolge (String) eines Terminalsymbols.

### 4.6.3 Relative Position von Terminal- und Nonterminalsymbolen

Zur Bestimmung der relativen Position zwischen Terminal- und Nonterminalsymbolen stehen die Funktionen *RELPOSNT* und *RELPOSTERM* zur Verfügung.

- *RELPOSNT* berechnet die Position eines Terminalsymbols bezüglich eines Nonterminalsymbols bzw. umgekehrt. Der Aufruf von

  ```
  GIPSYSpecs.RELPOSNT(N,term)
  ```

liefert dabei als Resultate folgende Konstanten aus dem Modul *GIPSYSpecs*:

| Resultat | Bedeutung |
|---|---|
| Undef | undefiniert (N=NIL oder term=NIL) |
| Before | term gehört zu einem Nonterminalsymbol, das vor N liegt |
| Behind | term gehört zu einem Nonterminalsymbol, das nach N folgt |
| AtStartOf | term entspricht FIRSTTERM(N) |
| Inside | term liegt zwischen FIRSTNT(N) und LASTNT(N) |
| AtEndOf | term entspricht LASTNT(N) |

Tab. 2: Rückgabewerte von *RELPOSNT*

- *RELPOSTERM* berechnet die relative Position zwischen zwei Terminalsymbolen. Der Aufruf von

  GIPSYSpecs.RELPOSTERM(anchor,term)

  liefert dabei folgende Resultate relativ zum Terminalsymbol *anchor*:

| Resultat | Bedeutung |
|---|---|
| Undef | undefiniert (anchor=NIL oder term=NIL) |
| Before | term liegt vor anchor |
| Same | term und anchor sind dieselben Terminalsymbole |
| Behind | term folgt nach anchor |

Tab. 3: Rückgabewerte von *RELPOSTERM*

## 4.7 Standardfunktionen in GIPSY/L

### 4.7.1 IsTerm

Die Standardfunktion*IsTerm* entscheidet, ob ein Terminalsymbol einer bestimmten Terminalklasse angehört.

```
PROCEDURE IsTerm(term: TERM; name: ARRAY OF CHAR): BOOLEAN;
```

### 4.7.2 ErrorMsg

*ErrorMsg* dient zum Einfügen von semantischen Fehlern.

```
PROCEDURE ErrorMsg(msg: ARRAY OF CHAR);
```

Falls *ErrorMsg* zur Ausführung kommt, wird im Dokument ein Fehlerelement mit der Meldung *msg* eingefügt.

Beispiel:

```
NONTERMINAL Decl ::= ('VAR' VarDecl | 'TYPE' TypeDecl ) ';';

   PROCEDURE CheckDecl(SELF: ProcDecl; VAR error: BOOLEAN);
      VAR nt: GIPSYSpecs.NONTERM; varDecl: VarDecl; typeDecl: TypeDecl;
   BEGIN
      nt:=GIPSYSpecs.FIRSTNT(SELF);
      IF (nt#NIL) & (nt IS VarDecl) THEN
         varDecl:=nt(VarDecl);
         varDecl.CheckDecl(varDecl,error)
      ELSIF (nt#NIL) & (nt IS TypeDecl)
         typeDecl:=nt(TypeDecl);
         typeDecl.CheckDecl(typeDecl,error)
      END;
      IF error THEN ErrorMsg('Fehler in Declaration') END
   END CheckDecl;
```

### 4.7.3 Ablaufsteuerung von Commands

Zur Steuerung des Ablaufs von Commands (Attributierungsprozeduren des Start-Nonterminalsymbols), die aufeinander aufbauen oder einander voraussetzen, wird die Anweisung

```
GIPSYSpecs.CmdDone(cmd: ARRAY OF CHAR);
```

angeboten.

*GIPSYSpecs.CmdDone* prüft, ob der Command *cmd* erfolgreich ausgeführt worden ist. Falls der Test negativ ausfällt, wird der Command *cmd* aufgerufen und versucht, die verlangte Bedingung zu erfüllen. Anschliessend wird der Test nochmals durchgeführt ist und das Resultat als Funtionswert zurückgeliefert. Wenn der Command immer noch nicht erfolgreich abgeschlossen werden konnte, kann mit *ErrorMsg* sollte Fehlermeldung ausgegeben und die Ausführung der Attributierungsprozedur abgebrochen.

```
IF ~GIPSYSpecs.CmdDone(Structures.CheckSyntaxCmd) THEN
   ErrorMsg('CheckSyntax missed');
```

```
    RETURN
END
```

## 4.8 Einschränkungen und Empfehlungen

Folgende Einschränkungen, die vom Oberon-Compiler hervorgerufen werden, sollten beim Entwerfen einer Attributierten Grammatik beachtet werden, um eine auf allen Plattformen lauffähige Umgebung zu erhalten:

- In einer Grammatik sind nicht mehr als 100 Nonterminal-Definitionen möglich.
- Möglichst kurze Namen für Nonterminalsymbole und Attributierungsprozeduren wählen:
  - ♦ Länge des Namens von Nonterminalsymbolen < 17 Zeichen
  - ♦ Länge des Namens Nonterminalsymbolen + Länge des Namens einer zugehörigen Attributierungsprozedur < 23 Zeichen
- Die Namen ‘New’, ‘Init’, und ‘Parse’ sind von GIPSY reserviert und dürfen nicht zur Benennung von Attributierungsprozeduren verwendet werden.

# 5 Anhang

## 5.1 Literaturverzeichnis


[ASU86] Aho, A., Sethi, R., Ullman, J.: „Compilers: Principles, Techniques and Tools“. Addison-Wesley, 1986.

[Hel95] Helbling, A., Murer, T.: „Spezifizieren und Erzeugen von integrierten Umgebungen mit GIPSY“. TIK-Report Nr. 12, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, Mai 1995.

[Mar94] Marti, R.: „GIPSY: Ein Ansatz zum Entwurf integrierter Softwareentwicklungssysteme“. ETH Dissertation Nr. 10463. Verlag der Fachvereine, Zürich, 1994.

[MM92] Marti, R., Murer, T.: „Extensible Attribute Grammars“. TIK-Report Nr. 6, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, Dezember 1992.

[Möss86] Mössenböck, H.P., Rechenberg, P.: „Ein Compiler-Generator für Mikrocomputer“. Carl Hanser Verlag München Wien, 1985.

[Möss90] Mössenböck, H. P.: „Coco/R: A Generator for Fast Compiler Front-Ends“. Departement Informatik, Institut für Computersysteme, ETH Zürich, Februar 1990.

## 5.2 GIPSY/L-Syntax für Scanner-Spezifikationen

```
SCANNER GIPSYsL;

    CHARACTERS
        eofC = {0X}.
        tabC = {09X}.
        eolC = {0DX}.
        blankC = {020X}.
        letter = {'A'..'Z','a'..'z'}.
        digit = {'0'..'9'}.
        hexDigit = digit + {'A'..'F'}.
        any = {0X..0FFX} - eofC.
        noQuote1 = any - {'"'}.
        noQuote2 = any - {"'"}.

    IGNORE
        blankC

    TOKENS
        eof = eofC.
        tab = tabC.
        eol = eolC.
        ident = letter {letter | digit}.
        char = '"' any '"' | "'" any "'".
        charConst = '0' [hexDigit] [hexDigit] 'X'.
        string = '"' noQuote1 noQuote1 {noQuote1} '"' | "'" noQuote2 noQuote2 {noQuote2} "'".
        anyGadget = @gadget.

    COMMENT FROM '(*' TO '*) NESTED

    LITERALS
        'SCANNER', ';', 'END', '.', 'CHARACTERS', 'IGNORE', 'TOKENS',
        'COMMENT', 'FROM', 'TO', 'NESTED', LITERALS', ',', '=', 'GADGET',
        '|', '(', ')', '[', ']', '{', '}', 'CONTEXT', '+', '-', '..', '@gadget'

END GIPSYsL.

GRAMMAR GIPSYsL;

    SCANNER = GIPSYsLScanner;

    SKIP
        eol, tab;

    NONTERMINAL Scanner
        ::= 'SCANNER' ident ';' Declaration 'END' ident '.'.

    NONTERMINAL Declaration
    ::= 'CHARACTERS' SetDecl {SetDecl}
        ['IGNORE' Set]
        'TOKENS' TokenDecl {TokenDecl}
        ['COMMENT' 'FROM' string 'TO' string ['NESTED']]
```

```
    [LITERALS Literal {',' Literal}].

  NONTERMINAL TokenDecl
  ::= ident '=' (TokenExpr | 'GADGET' ident '.' ident | anyGadget | '@gadget') '.'.

  NONTERMINAL TokenExpr ::= TokenTerm {'|'TokenTerm}.

  NONTERMINAL TokenTerm ::= TokenFactor {TokenFactor} [Context].

  NONTERMINAL TokenFactor
    ::= ident | char | string
    | '(' TokenExpr ')' | '[' TokenExpr ']' | '{' TokenExpr '}'.

  NONTERMINAL Context ::= 'CONTEXT' '(' TokenExpr ')'.

  NONTERMINAL SetDecl ::= ident '=' Set '.'.

  NONTERMINAL Set ::= BasicSet {('+'|'-') BasicSet}.

  NONTERMINAL BasicSet ::= ident | '{' [Range {',' Range}] '}'.

  NONTERMINAL Range ::= (char | charConst) ['..' (char | charConst)].

  NONTERMINAL Literal ::= char | string.

END GIPSYsL.
```

## 5.3 GIPSY/L-Syntax für Attributierte Grammatiken

```
SCANNER GIPSYpL0;

    CHARACTERS
        eofC = {0X}.
        tabC = {09X}.
        eolC = {0DX}.
        blankC = {020X}.
        any = {0X..0FFX} - tabC - eofC.
        letter = {'A'..'Z','a'..'z'}.
        digit = {'0'..'9'}.
        hexDigit = digit + {'A'..'F'}.
        noQuote1 = any - {'"'} - eolC.
        noQuote2 = any - {'"'} - eolC.

    IGNORE
        blankC

    TOKENS
        eof = eofC.
        tab = tabC.
        eol = eolC.
        ident  = letter {letter | digit}.
        integer = digit {digit} |digit {digit} CONTEXT ('..') |digit {hexDigit}'H'.
        real = digit {digit}'.' {digit} [('E'|'D') ['+'|'-'] digit {digit}].
        char = '"' noQuote1 '"' |'"' noQuote2 '"' | digit {hexDigit}'X'.
        string = '"' [noQuote1 noQuote1 {noQuote1}] '"' |'"' [noQuote2 noQuote2 {noQuote2}]'"'.

    COMMENT FROM'(*' TO'*)' NESTED

    LITERALS
        'GRAMMAR','(',')',';','SCANNER','IMPORT',',','BEGIN','END','.','SKIP',
        'NONTERMINAL','::=','PROCEDURE','|','[',']','{','}','ATTR',':=','CONST',
        'TYPE','VAR','=','ARRAY','OF','RECORD','POINTER','TO',':','*',
        'WHILE','DO','REPEAT','UNTIL','LOOP','EXIT','RETURN',
        'IF','THEN','ELSE','ELSIF','CASE','..','FOR','BY','WITH',
        'IS','#','<','<=','>','>=','IN','+','-','OR','/','DIV','MOD','&','~','NIL','^',
        'ErrorMsg','IsTerm'

END GIPSYpL0.

GRAMMAR GIPSYpL0;

    SCANNER GIPSYpL0Scanner;

    SKIP eol, tab;

    NONTERMINAL Gram
    ::='GRAMMAR' ident ('(' ident')' |';''SCANNER' ident)';'
        ['IMPORT' Import {',' Import}';']
        DeclSeq
        [Skip]
```

```
    Prod {Prod}
    ['BEGIN' StatSeq]'END' ident'.'.

NONTERMINAL Skip
::='SKIP' ident {',' ident}';'.

NONTERMINAL Prod
::='NONTERMINAL' NTObj
    ('::=' Expr'.' |'(' ident'.' ident')')
    [Attributes]
    {'PROCEDURE' ProcDecl}.

NONTERMINAL NTObj
::= ident.

NONTERMINAL Expr
::= Term {'|' Term}.

NONTERMINAL Term
::= Factor {Factor}.

NONTERMINAL Factor
::= Symbol |'(' Expr')' |'[' Expr']' |'{' Expr'}'.

NONTERMINAL Symbol
::= ident | char | string.

NONTERMINAL Attributes
::='ATTR' VarDecl';' {VarDecl';'}.

NONTERMINAL Import
::= ident [':=' ident].

NONTERMINAL DeclSeq
::= {'CONST' {ConstDecl';'}
    |'TYPE' {TypeDecl';'}
    |'VAR' {VarDecl';'}
    }
    {'PROCEDURE' ProcDecl}.

NONTERMINAL ConstDecl
::= IdentDef'=' ConstExpr.

NONTERMINAL ConstExpr
::= Expression.

NONTERMINAL TypeDecl
::= IdentDef'=' Type.

NONTERMINAL Type
::= TypeQualIdent | ArrayType | RecordType | PointerType | ProcType.

NONTERMINAL TypeQualIdent
::= Object ['.' Object].
```

```
NONTERMINAL ArrayType
::='ARRAY' ConstExpr {',' ConstExpr}'OF' Type.

NONTERMINAL RecordType
::='RECORD' ['(' TypeQualIdent')'] [VarDecl] {';' [VarDecl]}'END'.

NONTERMINAL PointerType
::='POINTER''TO' Type.

NONTERMINAL ProcType
::='PROCEDURE' [FormalParameters].

NONTERMINAL VarDecl
::= IdentList':' Type.

NONTERMINAL IdentList
::= IdentDef {',' IdentDef}.

NONTERMINAL ProcDecl
::= ProcHeading';' ProcBody ident';'.

NONTERMINAL ProcHeading
::= ['*'] IdentDef FormalParameters.

NONTERMINAL ProcBody
::= DeclSeq ['BEGIN' StatSeq]'END'.

NONTERMINAL FormalParameters
::= ['(' [FPSection {';' FPSection}]')' [':' TypeQualIdent]].

NONTERMINAL FPSection
::= ['VAR'] ident {',' ident}':' FormalType.

NONTERMINAL FormalType
::='ARRAY''OF' FormalType | TypeQualIdent.

NONTERMINAL StatSeq
::= Statement {';' Statement}.

NONTERMINAL Statement
::= [ AssOrCall
    | IfStatement
    | CaseStatement
    |'WHILE' Expression'DO' StatSeq'END'
    |'REPEAT' StatSeq'UNTIL' Expression
    | ForStatement
    |'LOOP' StatSeq'END'
    |'EXIT'
    | WithStatement
    |'RETURN' [Expression]
    |'ErrorMsg' '(' Designator ',' string')'
    ].

NONTERMINAL AssOrCall
::= Designator [':=' Expression].
```

```
NONTERMINAL IfStatement
::='IF' Expression'THEN' StatSeq
    {'ELSIF' Expression'THEN' StatSeq}
    ['ELSE' StatSeq]
    'END'.

NONTERMINAL CaseStatement
::='CASE' Expression'OF'
     Case {'|' Case}
     ['ELSE' StatSeq]
    'END'.

NONTERMINAL Case
::= [CaseLabelList':' StatSeq].

NONTERMINAL CaseLabelList
::= CaseLabels {',' CaseLabels}.

NONTERMINAL CaseLabels
::= ConstExpr ['..' ConstExpr].

NONTERMINAL ForStatement
::='FOR' ident':=' Expression'TO' Expression ['BY' Expression]'DO' StatSeq'END'.

NONTERMINAL  WithStatement
::='WITH' GuardedVariable':' GuardedTypeId'DO' StatSeq
     {'|' GuardedVariable':' GuardedTypeId'DO' StatSeq}
     ['ELSE' StatSeq]
    'END'.

NONTERMINAL GuardedVariable
::= Designator.

NONTERMINAL GuardedTypeId
::= TypeQualIdent.

NONTERMINAL Expression
::= SimpleExpr [RelOp SimpleExpr |'IS' TypeQualIdent].

NONTERMINAL RelOp
::='=' |'#' |'<' |'<=' |'>' |'>=' |'IN'.

NONTERMINAL SimpleExpr
::= ['+' |'-'] OberonTerm {AddOp OberonTerm}.

NONTERMINAL AddOp
::='+' |'-' |'OR'.

NONTERMINAL OberonTerm
::= OberonFactor {MulOp OberonFactor}.

NONTERMINAL MulOp
::='*' |'/' |'DIV' |'MOD' |'&'.
```

```
    NONTERMINAL OberonFactor
    ::='(' Expression')'
        |'~' OberonFactor
        |'{' [Elements {',' Elements}]'}'
        | Designator
        | integer
        | real
        | char
        | string
        |'NIL'
        |'IsTerm' '(' ident',' ident')'.

    NONTERMINAL Elements
    ::= Expression ['..' Expression].

    NONTERMINAL Designator
    ::= Object {'.' ident
        |'[' Expression {',' Expression}']'
        |'^'
        |'(' [Expression {',' Expression}]')'
       }.

    NONTERMINAL IdentDef
    ::= ident ['*'].

    NONTERMINAL Object
    ::= ident.

END GIPSYpL0.
```

Report 1 - A Relational Data Base Design for an X.500 Directory System Agent
(F. Perruchoud, C. Lanz, B. Plattner, July 1990)
Report 2 - Model and Functionality Definition for the Collaborative Editing Conferencing System Multim ETH.
(H. Lubich, July 1990)
Report 3 - X.400 Security Capabilities: Evaluation and Constructive Criticism
(M. Müller, August 1990)
Report 4 - CPU Evaluation for ADAM
(Schibli, M. Tadjan, February 1992)
Report 5 - Aspekte computergestützter Kooperation - Schriftliches Material eines Seminars an der ETH Zürich
(Hannes Lubich, Januar 1993)
Report 6 - Extensible Attribute Grammars
(R. Marti, T. Murer, December 1992)
Report 8 - Test Case Validation — TTCN Test Case Validation Against SDL Specifications
(F. Kristoffersen, T. Walter, May 1994)
Report 9 - Conformance and Interoperability - A critical assessment
(T. Walter, B. Plattner, September 1994)
Report 10 - OOP-Softwarearchitektur für Multimediakommunikation
(Serge Hoffmann, März 1995)
Report 11 - A Comparison of Selection Schemes used in Genetic Algorithms
(Tobias Blickle, Lothar Thiele, April 1995)
Report 12 - Spezifizieren und Generieren von integrierten Umgebungen mit GIPSY
(A. Helbling, T.Murer, Mai 1995)
Report 13 - Die Spezifizierungssprache GIPSY/L
(A. Helbling, T. Murer, Mai 1995)