Universität – Gesamthochschule – Essen

Fachbereich Mathematik

Informatik-Berichte

SETL/E Sprachbeschreibung
Version 0.1

Ernst-Erich Doberkat
Ulrich Gutenbeil
Wilhelm Hasselbring

Report 01-90

SETL/E Sprachbeschreibung
Version 0.1

Ernst-Erich Doberkat
Ulrich Gutenbeil
Wilhelm Hasselbring

26. März 1990

Abstract

Wir geben eine Einführung in SETL/E[1], eine Erweiterung der Programmiersprache SETL. Dieses Dokument ist als Entwurfsdokument gedacht. Wir diskutieren die Eigenschaften von SETL/E und weisen auf Änderungen gegenüber SETL hin. In diesem ersten Dokument wird zunächst nur der Kern der Sprache beschrieben, weitere Dokumente werden Erweiterungen wie etwa Moduln, abstrakte Datentypen und persistente Strukturen beschreiben.

[1] Set Theoretic Language/Essen

Inhaltsverzeichnis

Tabellenverzeichnis

1 Einleitung

Wir geben eine Einführung in SETL/E, ein Dialekt und eine Erweiterung der Programmiersprache SETL. Dieses Dokument ist als Entwurfsdokument gedacht. In einem ersten Schritt wird zunächst nur der Kern der Sprache beschrieben, weitere Schritte werden Spracherweiterungen wie etwa Moduln, abstrakte Datentypen und dgl. beschreiben.
SETL ist in den folgenden Büchern beschrieben:

- *Schwartz, J. T. et al.*: Programming With Sets – An Introduction to SETL. Graduate Texts in Computer Science, Springer-Verlag, Berlin etc., 1986
- *Doberkat, E.-E., Fox, D.*: Software Prototyping mit SETL. Leitfäden und Monographien der Informatik. Teubner-Verlag, Stuttgart, 1989

Wann immer wir uns hier auf SETL beziehen, verweisen wir implizit auf die obigen Texte, die der Leser konsultieren möge, falls er SETLs unkundig ist.

1.1 Zur Notation in dieser Sprachbeschreibung

Die Notation folgt einer erweiterten Backus-Naur-Form (BNF). *Nicht-Terminalzeichen* werden in spitzen Klammern eingeschlossen und in Schrägschrift gedruckt:

$$<\textit{nonterminal}>$$

Terminale Symbole werden im Text so gedruckt:

`terminal`

Durch '|' können auf der rechten Seite Alternativen angegeben werden. Um die Darstellungen übersichtlicher gestalten zu können, werden noch einige zusätzliche Erweiterungen benutzt: Alternativen können in speziellen Klammern auch übereinander geschrieben werden. Die Klammern haben die folgenden Bedeutungen:

1. [...] schließen optionale Alternativen ein (das leere Wort kann statt dessen genommen werden)
2. {...} schließen obligate Alternativen ein (eine muß genommen werden)
3. $(\ldots)^*$ beliebige Wiederholung des Klammerinhalts oder das leere Wort
4. $(\ldots)^+$ beliebige Wiederholung des Klammerinhalts. Das leere Wort ist nicht erlaubt

Die grammatischen Einsprengsel sind nicht dazu gedacht, sich zu einer vollständigen Grammatik zusammensetzen zu lassen, sondern dienen lediglich der Demonstration der besprochenen Spracheigenschaften.

2 Lexikalische Konventionen

Es gibt die folgenden Klassen von lexikalischen Token: Bezeichner, Operator-Bezeichner, numerische Literale, Zeichenketten, vordefinierte Operatoren und anderen Token. Blanks und andere nicht sichtbare Zeichen, die verschieden sind vom EOF-Zeichen, dienen der Trennung von Token und werden andernfalls ignoriert.

2.1 Kommentare

Kommentare beginnen wie in Ada mit einem doppelten Minuszeichen (ohne Blank dazwischen) und erstrecken sich bis zum Ende der Zeile. Sie werden vom Compiler überlesen.

2.2 White Space

Lexikalische Einheiten werden durch *white space* voneinander getrennt. Dies ist zunächst einmal der Zwischenraum, dann Tabulatoren ("\t") und das *newline*-Symbol "\n".

2.3 Bezeichner

Bezeichner werden als Namen und als reservierte Wörter benutzt; sie müssen wie in SETL oder Pascal mit einem Buchstaben beginnen, darauf kann eine Folge von Buchstaben, Ziffern oder Unterstrichen folgen. Es werden Groß- und Kleinbuchstaben zugelassen, aber nicht unterschieden. Die reservierten Bezeichner sind in Abschnitt 10 (p. 40) zu finden.

2.4 Operator-Bezeichner

Zur Bezeichnung von benutzer-definierten Operatoren ist es sinnvoll, die Klasse der hierfür möglichen Bezeichner zu erweitern, um flexibler sein zu können. Ein Bezeichner für einen Operator muß der regulären Menge

$$\mathcal{B}(\mathcal{B} \cup \mathcal{N} \cup \mathcal{S})^*$$

entnommen sein, wobei

$$\begin{aligned} \mathcal{B} &:= \{\texttt{'a'}, \ldots, \texttt{'z'}, \texttt{'A'}, \ldots, \texttt{'Z'}\} \\ \mathcal{N} &:= \{\texttt{'0'}, \ldots, \texttt{'9'}\} \cup \{\_\} \\ \mathcal{S} &:= \{?, @, \%, !, \backslash, \$, \&\} \end{aligned}$$

2.5 Numerische Literale

Alle Zahlen werden zur Basis 10 genommen, so daß sich die Angabe einer Basis erübrigt. Ganze Zahlen werden notiert durch eine Folge von Ziffern. Die Größe der kleinsten und der größten darstellbaren ganzen Zahl hängt von der Maschine ab, auf der das Programm läuft.
Gleitkommazahlen werden notiert durch eine nicht-leere Folge von Ziffern, gefolgt von einem Dezimalpunkt und einer nicht-leeren Folge von Ziffern. Optional kann ein Exponent angehängt werden, der notiert wird durch Angabe eines (großen oder kleinen) E, gefolgt von einer ganzen Zahl. Die Darstellung von reellen Zahlen ist maschinenabhängig. Es sollte jedoch durch die Implementierung gesichert sein, daß die arithmetische Genauigkeit der doppelten Genauigkeit von C auf der gleichen Maschine entspricht.

2.6 Zeichenketten

Zeichenketten sind eine Folge von Zeichen, die begrenzt werden durch ". Eingebettete Anführungszeichen sind nicht zulässig, ebensowenig wie *white space*, vgl. 2.2 (p. 5). Als Konstanten sind definiert: Tabulatoren, notiert als "\t" und das *newline*-Symbol, das als "\n" notiert wird.
Die Gesamtlänge einer Zeichenkette ist durch die vorgegebene Maschine beschränkt, sie kann durch die Implementierung z.B. auf 256 Zeichen eingeschränkt werden.

2.7 Anmerkung zu Operatoren

Einige Schlüsselwörter sind vordefinierte Operatoren, einige Operatoren werden durch Zeichen dargestellt, die weder Buchstabe noch Ziffer sind. Wir werden diese Operatoren diskutieren, wenn wir die entsprechenden Konstrukte besprechen.
Binäre Operatoren können wie in SETL zur Abkürzung verwendet werden. Dies betrifft ebenfalls — wie in SETL — benutzerdefinierte binäre Operatoren. Sei X%@_ ein benutzerdefinierter Operator, so darf

$$AberWerWirdDennGleichSoLang := AberWerWirdDennGleichSoLang\ X\%@\_\ 1$$

notiert werden als

$$AberWerWirdDennGleichSoLang\ X\%@\_ := 1.$$

2.8 Makros

SETL/E bietet die Möglichkeit, Token-Folgen parametrisiert abzukürzen. In einer Phase zwischen der lexikalischen und der syntaktischen Analyse werden diese Abkürzungen aufgelöst. Eine solche Abkürzung wird vereinbart durch:

```
macro Bezeichner (Argumentliste);
Lokale Vereinbarungen
Folge von Token
endm Bezeichner;
```

Die in Klammern eingeschlossene Liste der Parameter kann auch fehlen; falls sie vorhanden ist, besteht sie aus einer durch Kommata voneinander getrennten Liste von Bezeichnern. Makros können lokale Parameter haben. Sie werden im Text des Makros vereinbart durch Angabe des Schlüsselworts `local` und eine wieder durch Kommata getrennte und durch Semikolon abgeschlossene Liste von Bezeichnern.
Die Gültigkeit von Makros ist lexikalisch bestimmt: ein Bezeichner ist an einen Makronamen gebunden von der Definition des Makros an bis zum Ende der Datei, es sei denn, die Bindung wird durch `drop` explizit aufgelöst. `drop` ist eine Direktive an den Makro-Prozessor, allgemein sieht diese Direktive so aus: `drop`, gefolgt von einer durch Kommata voneinander getrennten Liste von Bezeichnern, abgeschlossen durch ein Semikolon. Makro-Definitionen dürfen nicht überschrieben werden: ist ein Bezeichner an ein Makro gebunden, so muß erst die `drop`-Direktive ausgeführt werden, bevor derselbe Bezeichner neu gebunden werden kann.
Der Makro-Prozessor arbeitet stackorientiert: wird in der von der lexikalischen Analyse erzeugten Token-Folge ein Bezeichner gefunden, der an eine Makro-Definition gebunden ist, so wird dieser Bezeichner ersetzt durch die entsprechende Token-Folge, wobei ggf. formale Parameter durch aktuelle ersetzt werden. Diese neu entstandene Token-Folge wird dann von der Position an, bei der die Makro-Definition angetroffen wurde, linear durchsucht, ob weitere an Makros gebundene Bezeichner anzutreffen sind; wenn ja, wiederholt sich dieser Prozeß so lange, bis kein Makro-Aufruf mehr vorhanden ist. Es sollte beachtet werden, daß für Makros eine Art lazy evaluation vereinbart ist, die anders arbeitet als der Makro-Prozessor etwa in der Programmiersprache C oder in TEX. Insbesondere kann der Makro-Prozessor durch eine rekusive Makro-Definition in eine Endlosschleife geschickt werden.
Die in einem Makro als lokal definierten Namen werden beim Antreffen der Makro-Definition vom Makro-Prozessor durch eindeutig definierte, dem Programmierer nicht zugängliche Namen ersetzt. Makros können geschachtelt werden, also selbst wieder Makro-Definitionen enthalten:

```
macro DefMacro(x, y);
macro x; y endm x;
endm DefMacro;
```

hat den Effekt, daß nach den Aufrufen

```
DefMacro(eins, 1);
DefMacro(zwei, 2);
```

die Makros *eins* und *zwei* definiert und an die Werte 1 bzw. 2 gebunden sind.

3 Programm-Struktur

Ein (einfaches) SETL/E-Programm sieht grammatisch wie folgt aus:

$$
\begin{aligned}
&\langle ProgDefn\rangle \longrightarrow \langle ProgHeader\rangle\langle ProgBody\rangle\langle ProgTrailer\rangle\\
&\langle ProgHeader\rangle \longrightarrow \texttt{program}\langle ProgName\rangle\boxed{;}\\
&\langle ProgName\rangle \longrightarrow \langle Bezeichner\rangle\\
&\langle ProgParamName\rangle \longrightarrow \langle Bezeichner\rangle\\
&\langle ProgBody\rangle \longrightarrow [\ \langle Decls\rangle\ ]^{*}\langle Stmts\rangle[\ \langle ProcDefns\rangle\ ]\\
&\langle Decls\rangle \longrightarrow \begin{pmatrix}\langle Vars\rangle\\ \langle Consts\rangle\end{pmatrix}\\
&\langle Stmts\rangle \longrightarrow \langle Stmt\rangle^{+}\\
&\langle ProcDefns\rangle \longrightarrow (\ \langle ProcDefn\rangle\ )^{*}\\
&\langle ProgTrailer\rangle \longrightarrow \texttt{end}\langle ProgName\rangle\boxed{;}
\end{aligned}
$$

Eine Programm-Definition besteht also aus dem Schlüsselwort **program**, auf das der Name des Programms folgt. Im darauf folgenden optionalen Deklarations-Abschnitt werden Konstanten und Variablen definiert, die für das gesamte Programm sichtbar sind. Der Programmtext selbst folgt. Er ist eine Liste von Anweisungen, die die Hauptprozedur des Programms ausmachen. Die Definition wird fortgesetzt vom Prozedur-Abschnitt, der aus einer Liste von Deklarationen für solche Prozeduren, die vom Hauptprogramm aus sichtbar sind, besteht. Abschließend folgt der Schlüsselwort **end** mit dem Namen des Programms.

3.1 Programm-Parameter

Programm-Parameter sind Zeichenketten, die dem Programm über die Kommando-Zeile beim Aufruf mitgegeben werden können; sie werden im vordefinierten Tupel **argv** abgelegt. Die Parameter werden durch *white space* voneinander getrennt, beides ganz analog zum Vorgehen bei C. Diese Konstruktion ersetzt die **getipp/getspp**-Konstruktion in SETL.

3.2 Scope und Kontur

Wie in SETL sind Objekte *by default* lokal in dem Scope, in dem sie definiert sind. Zunächst verstehen wir hier unter einem *Scope* den Text des Hauptprogramms, einer Prozedur, eines Operators oder einer Ausnahme. Weitere Scopes kommen später in Form von Moduln oder Paketen hinzu. Scopes sind hierarchisch angeordnet. An der Spitze der Hierarchie steht als Wurzel des Scope-Baums das Hauptprogramm, und ein Scope $\mathcal{A}$ hat einen anderen Scope $\mathcal{B}$ als Sohn, falls $\mathcal{B}$ als lokales Objekt in $\mathcal{A}$ definiert ist.

Die Deklaration **visible** im Scope $\mathcal{A}$ macht ein Objekt in allen untergeordneten Scopes bekannt, also in allen Scopes, die im Unterbaum von $\mathcal{A}$ zu finden sind. Solcherart ausgezeichnete Namen können verschattet werden: ist der Scope $\mathcal{B}$ ein Abkömmling des Scope $\mathcal{A}$, so verschattet die Deklaration **visible** x in $\mathcal{B}$ einen in $\mathcal{A}$ als **visible** deklarierten Namen x. Diese Vereinbarungen über die Sichtbarkeit betreffen Variablen und Konstanten gleichermaßen und regeln insbesondere die Sichtbarkeit lokal vereinbarter Prozeduren.
Eine Kontur ist die Visualisierung eines Scope zur Laufzeit. Jede Kontur entspricht einem Scope und enthält die lokal definierten Objekte. Enthält eine Kontur γ eine andere Kontur δ, so sind in δ lediglich die hier definierten Objekte sichtbar, und diejenigen Objekte, die für δ durch **visible** in den statischen Vorgängern sichtbar gemacht werden, nicht jedoch andere Objekte aus γ oder umgebenden Konturen. **visible** in γ ist nicht sichtbar in δ, falls γ nicht auch ein statischer Vorgänger von δ ist. Nur der statische Scope-Baum bestimmt die Sichtbarkeit!
Konturen werden in *LIFO*-Disziplin verwaltet: die jeweils zuletzt geschaffene Kontur ist die aktuelle. Jede Kontur hat einen *statischen* und einen *dynamischen* Vorgänger: der statische Vorgänger ist durch den Programm-Text gegeben als die textuell umgebende Prozedur, der dynamische Vorgänger ist die Prozedur, die die in Rede stehende aufgerufen hat. Das folgende Beispiel möge dies erläutern:

```
procedure outer;
visible n := 0;
p(empty);
 procedure empty;
  pass;
 end empty;
 procedure p(f);
 visible i := n;
 if n = 0 then n := 1; p(q); end if;
 put("i:1 =%d", i);
 q();
 put("i:2 =%d", i);
  procedure q;
  if n = 1 then n := n + 1; f(); end if;
  i := i + 1;
  end q;
 end p;
end outer;
```

Der Aufruf

outer()

hat zur Folge, daß ausgedruckt wird:

```
i:1 = 1
i:2 = 2
i:1 = 1
i:2 = 2
```

3.3 Zur Deklaration von Konstanten und Variablen

Parameter ebenso wie der Name einer Prozedur werden als durch **visible** deklarierte lokale Variablen behandelt. Variable können bei ihrer Deklaration als **visible** auch initialisiert werden, indem auf ihren Namen ein Zuweisungszeichen (:=) mit dem initialisierenden Ausdruck folgt. Der Wert des Ausdrucks muß zu dem Zeitpunkt, in dem diese Sichtbarkeitsdeklaration elaboriert wird, bekannt sein. Syntaktisch sieht eine Variablendeklaration wie folgt aus:

<*Vars*> ⟶ `visible`<*VarsList*>[`;`]
<*VarsList*> ⟶ (<*Bezeichner*>[`:=`<*Expr*>][`,`])*<*Bezeichner*>[`:=`<*Expr*>][`;`]

Neben Variablen können auch lokale Konstanten deklariert werden: dies geschieht mit Hilfe des Schlüsselworts `constant`. Während bei Variablen-Deklarationen die Initialisierung optional ist, ist dies bei Konstanten nicht der Fall: Konstanten müssen initialisiert sein, dies geschieht durch Angabe ihres Werts nach dem Zuweisungszeichen. Konstanten können lokal im gegenwärtig gültigen Scope sein, analog zu Variablen macht die Deklaration `visible` ein konstantes Objekt in allen untergeordneten Scopes bekannt. Dies ist in der Syntax reflektiert:

<*Consts*> ⟶ [`visible`] `constant`<*ConstList*>[`;`]
<*ConstList*> ⟶ (<*Bezeichner*>[`:=`]<*Expr*>[`,`])*<*Bezeichner*>[`:=`]<*Expr*>[`;`]

3.4 Die Konstanten sind dynamisch

Im Gegensatz zu SETL sind die Konstanten in SETL/E *dynamische* Konstanten (und nicht *manifesten* Konstanten, deren Wert fixiert ist und zur Übersetzungszeit festgestellt werden kann), die allerdings den undefinierten Wert om nicht annehmen dürfen. Diese dynamischen Konstanten können durch den Compiler ähnlich wie Variable behandelt werden: bei ihrer Deklaration wird ihnen ein Wert zugeordnet, aber weitere Zuweisungen innerhalb ihres Gültigkeitsbereichs sind nicht möglich. Konstantendeklarationen werden dynamisch ausgewertet, d.h. die Konstante erhält in der Regel erst dann ihren Wert zugewiesen, wenn der Kontrollfluß den zugehörigen Scope erreicht; der Wert kann dann innerhalb des Scopes nicht geändert werden. Dies hat zur Folge, daß (lokale) Konstanten während eines Programmdurchlaufes bei jedem Eintritt in den zugehörigen Scope einen neuen Wert oder auch Typ annehmen können — andererseits sind dadurch aber im Vergleich mit SETL keinerlei Einschränkungen bezüglich des Ausdrucks auf der rechten Seite der Konstantendeklaration mehr nötig. Darüberhinaus kann jeder Ausdruck, der zu einem definierten Wert ausgewertet werden kann, auf der rechten Seite einer Konstanten-Deklaration stehen.

3.4.1 Elaboration von Konstanten-Vereinbarungen

Eine Konstante wird *elaboriert*, indem der Wert der rechten Seite an den Namen der Konstanten gebunden wird. Bevor das geschieht, muß der Wert der rechten Seite bekannt sein.

3.5 Programm-Text

Eine Anweisungsliste ist eine Folge von Anweisungen, in der jede Anweisung dieser Liste durch ein Semikolon vom nächsten abgetrennt ist. Ein Semikolon wird hier als Terminator benutzt, nicht als Separator wie in Pascal.

3.6 Prozeduren

Eine Prozedur ist ein benannter Bestandteil eines Programms, der lokale Daten und Anweisungen enthält, die durch den Aufruf der Prozedur ausgeführt werden. Eine Prozedur kann einen Wert zurückgeben (dies wird durch die `return`-Anweisung geregelt); als *default* wird der Wert om zurückgegeben.

Grammatisch sieht die Definition einer Prozedur so aus:

$$
\begin{aligned}
&\langle ProcDefn\rangle \longrightarrow \langle ProcHeader\rangle\langle ProcBody\rangle\langle ProcTrailer\rangle\\
&\langle ProcHeader\rangle \longrightarrow \texttt{procedure}\langle ProcName\rangle\langle ParamList\rangle\boxed{;}\\
&\langle ProcName\rangle \longrightarrow \langle ProcBezeichner\rangle\\
&\langle ParamList\rangle \longrightarrow \left[\ \boxed{(}\ (\ \langle ParamMode\rangle\langle ParamName\rangle\boxed{,}\ )^{*}\langle ParamMode\rangle\langle ParamName\rangle\boxed{)}\ \right]\\
&\langle ParaMode\rangle \longrightarrow \left[\begin{array}{l}\texttt{rw}\\ \texttt{wr}\\ \texttt{rd}\end{array}\right]\\
&\langle ParamName\rangle \longrightarrow \langle Bezeichner\rangle\\
&\langle ProcBody\rangle \longrightarrow [\ \langle LocalDecls\rangle\ ]^{*}\langle Stmts\rangle[\ \langle ProcDefn\rangle\ ]\\
&\langle LocalDecls\rangle \longrightarrow \left(\begin{array}{c}\langle Vars\rangle\\ \langle Consts\rangle\end{array}\right)^{*}\\
&\langle ProcTrailer\rangle \longrightarrow \texttt{end}\langle ProcName\rangle\boxed{;}
\end{aligned}
$$

Zunächst geben wir also das Schlüsselwort **procedure** an, dann den Namen der Prozedur, optional gefolgt von einer Liste formaler Parameter in Klammern; ein Semikolon schließt diese Kopfzeile ab. Der Prozedur-Rumpf besteht aus einem optionalen Abschnitt, in dem Konstante und Variable deklariert werden, dann folgt eine Liste von Anweisungen, die den Text der Prozedur selbst ausmachen, und schließlich ein optionaler Abschnitt, in dem lokale Prozeduren definiert werden. Die Prozedur wird beendet durch Angabe des Schlüsselworts **end**, auf das der Name der Prozedur folgt, und ein Semikolon.

Die Liste der formalen Parameter ist eine Liste von Namen; jeder Name kann optional mit einer Zugangsspezifikation verbunden sein. Diese Spezifikation gibt an, wie der entsprechende Parameter übergeben wird, hier haben wir die folgenden drei Möglichkeiten: `rd` als *default*, dies sind Parameter, die nur gelesen werden, deren Werte-Änderungen aber nicht an die aufrufende Prozedur weitergeben werden, `rw` als solche Parameter, die gelesen und geschrieben werden können, die also von der aufrufenden Prozedur initialisiert sind und von der aufrufenden Prozedur geändert werden können, so daß Änderungen an den Aufrufer zurückgegeben werden. Parameter können mit `wr` als nur geschriebene Parameter ausgezeichnet werden; diese mit `wr` gekennzeichneten Parameter werden beim Eintritt in die Prozedur zu `om` initialisiert und geben ihren Wert an die aufrufende Prozedur zurück. Die Parameter-Übergabe bei `rd`-Parametern erfolgt durch call by value, es wird also eine Kopie des entsprechenden aktuellen Parameters angefertigt, mit der die Prozedur arbeitet. `rw`-Parameter werden durch *call by value/result* übergeben: für einen solchen Parameter legt der Compiler eine temporäre Variable an, in die der Anfangs-Wert kopiert wird. Die Prozedur selbst arbeitet mit dieser temporären Variablen, nach Rückkehr vom Prozedur-Aufruf erhält die als aktueller Parameter fungierende Variable den Wert der temporären Variablen. Die Bindung lokaler Objekte ist im Abschnitt 4.2.4 (p. 20) diskutiert.

Die Anzahl der formalen muß mit der Anzahl der aktuellen Parameter beim Prozedur-Aufruf übereinstimmen; wir lassen also keine Prozeduren mit einer variablen Anzahl von Parametern zu. Parameterlose Prozeduren werden beim Aufruf mit der leeren Parameter-Liste () versehen, bei ihrer Vereinbarung darf keine Parameter-Liste angegeben zu werden. Gibt eine Prozedur einen Wert w zurück, so muß dies im Text der Prozedur durch **return** w angedeutet sein; Werte eines Prozedur-Aufrufs müssen nicht notwendig benutzt werden, so daß ein Prozedur-Aufruf wie eine Anweisung behandelt werden kann. Formale wie aktuelle Parameter und Werte einer Prozedur können selbst wieder Prozeduren sein.

3.6.1 Anonyme Prozeduren

Anonyme Prozeduren haben sich selbst als Wert und sind entfernt mit λ-Ausdrücken in LISP vergleichbar. Syntaktisch werden diese anonymen Prozeduren wie folgt vereinbart:

<LambdaDefn> ⟶ *<LambdaHeader>* [:] *<LambdaBody>* *<LambdaTrailer>*
<LambdaHeader> ⟶ `lambda` *<ParamList>*
<LambdaBody> ⟶ [*<LocalDecls>*] *<Stmts>* [*<ProcDefn>*]
<LambdaTrailer> ⟶ `end lambda`

auf das Schlüsselwort `lambda` folgt wie bei der Vereinbarung *normaler* Prozeduren eine Liste formaler Parameter mit Zugangsspezifikation, die in Klammern eingeschlossen ist; die Parameter sind durch Kommata getrennt. Auf die Liste der formalen Parameter folgt — durch einen Doppelpunkt voneinander getrennt — der Text so, wie er auch bei benannten Prozeduren üblich ist. Diese Vereinbarung wird durch `end lambda` abgeschlossen. Der Typ anonymer Prozeduren ist `proctype`. Anonyme Prozeduren können überall dort auftreten, wo Prozedur-Namen verwendet werden dürfen; ist eine anonyme Prozedur rekursiv, so kann sie durch das Schlüsselwort `self` auf sich selbst Bezug nehmen. Der Bezeichner `self` wird behandelt wie der Name einer Prozedur — er ist überall im Text der anonymen Prozedur sichtbar, und kann auf der rechten Seite von Zuweisungen stehen, also z.B.

```
lambda (x, y, n):
  g := self;
  h(n);
  return if n = 0 then x
         elseif n > 0 then g(y, x + y, n - 1)
         end if;

  procedure h(k); -- lokale Prozedur
    g := y; -- g ist hier lokal
    print("Aufruf der Funktion: %d\n:", k);
  end h;
end lambda;
```

berechnet für $n \geq 0$

$$(x, y, n) \mapsto F_{n-1}x + F_n y$$

wobei F_n die n^{te} Fibonacci-Zahl ist. Im Text der lokalen Prozedur h ist zwar `self` bekannt, aber nicht die Variable g (die dazu hätte als `visible` deklariert werden müssen).
Anonyme Prozeduren können eingelesen werden, vgl. 8.3 (p. 34). Sie können *on the fly* definiert und ausgeführt werden, wie das folgende Beispiel zeigt:

```
lambda(x): return x + 1; end lambda(4)
```

ergibt den Wert 5.
Die Bindung lokal nicht gebundener Werte erfolgt wie bei benannten Prozeduren, vgl. 4.2.4 (p. 20); ein Beispiel möge das erläutern:

```
procedure f(t);
  if t ≤ 1 then return lambda:
                         put("t = %d\n", t);
                       end lambda;
  else t := t + 1;
       return lambda:
                put("t = %d\n", t);
              end lambda;
  end if;
end f;
```

Ruft man nun auf

```
f(0)(); f(2)();
```

so erhält man als Ergebnis

```
t = 0
t = 3
```

Prozeduren und Operatoren können *anonymisiert* werden. Dies geschieht bei Prozeduren, indem im Prozedur-Text jedes Auftauchen des Namens der Prozedur durch eine generierte Variable ersetzt wird, die als `visible` deklariert und mit `self` initialisiert ist. Der generierte Variablenname kann lexikalisch nicht durch den Benutzer vereinbart werden. Kopf- und Fußzeilen der Definition werden entsprechend justiert. Operatoren werden als Spezialfälle von Prozeduren betrachtet. Ist p eine Prozedur, so hat `str` p die Zeichenkette zum Wert, die durch die Anonymisierung entsteht. Analog druckt etwa put("%p", p) diese anonymisierte Version.
Die Anonymisierung der Prozedur

```
procedure fib (x, y, n):
  h(n);
  return if n = 0 then x
         elseif n > 0 then fib(y, x + y, n - 1)
         end if;

  procedure h(k); -- lokale Prozedur
    print("Aufruf der Funktion: %d\n:", k);
  end h;
end fib;
```

ist

```
lambda (x, y, n):
  g.0003421 := self;
  h(n);
  return if n = 0 then x
         elseif n > 0 then g.0003421(y, x + y, n - 1)
         end if;

  procedure h(k); -- lokale Prozedur
    print("Aufruf der Funktion: %d\n:", k);
  end h;
end lambda;
```

Die entsprechende Zeichenkette ist dann auch der Wert von `str` *fib* (ohne Einrückungen oder Zeilenwechsel).

3.6.2 Benutzer-definierte Operatoren

Wie in SETL hat der Benutzer in SETL/E die Möglichkeit, Operatoren selbst zu definieren, wobei Operatoren unär oder binär sein können; unäre Operatoren werden als Prä- oder Postfix-Operatoren benutzt, binäre als Infix-Operatoren. Parameter werden an Operatoren stets als `rd`-Parameter übergeben. Syntaktisch sieht eine Operator-Deklaration wie folgt aus:

$$
\begin{aligned}
&\langle \textit{OperatorDefn} \rangle \longrightarrow \langle \textit{OperatorHeader} \rangle \langle \textit{ProcBody} \rangle \langle \textit{OperatorTrailer} \rangle \\
&\langle \textit{OperatorHeader} \rangle \longrightarrow \texttt{operator} \langle \textit{ProcName} \rangle \langle \textit{OpParams} \rangle \boxed{;} \left[\ \langle \textit{OperatorAusz} \rangle \ \right] \\
&\langle \textit{OpParams} \rangle \longrightarrow \langle \textit{Bezeichner} \rangle \left[\ \boxed{,} \langle \textit{Bezeichner} \rangle \ \right] \\
&\langle \textit{OperatorAusz} \rangle \longrightarrow \texttt{use as} \left[\ \langle \textit{Prop} \rangle \ \right] \texttt{operator} \boxed{;} \\
&\langle \textit{Prop} \rangle \longrightarrow \left\{ \begin{array}{c} \texttt{left\_associative} \\ \texttt{right\_associative} \\ \texttt{transitive} \\ \texttt{prefix} \\ \texttt{postfix} \end{array} \right\}
\end{aligned}
$$

Die Spezifikation der Benutzung mittels `use` sieht folgendermaßen aus:

```
use as ⟨Prädikat;⟩
```

hierbei kann als Prädikat das folgende verwendet werden:

`left_associative` der Operator ist links-assoziativ (ein Operator op heißt *links-assoziativ*, wenn a op b op c gedeutet wird als (a op b) op c

`right_associative` der Operator ist rechts-assoziativ

`transitive` der Operator ist transitiv, a_1 op a_2 ... op a_n wird gedeutet als a_1 op a_2 and ... and a_{n-1} op a_n. Dies ist ganz offensichtlich nur sinnvoll für binäre Operatoren mit Booleschen Werten.

`prefix` der Operator wird als Präfix-Operator gebraucht, d.h. er ist von der Form op a — dies ist offensichtlich nur sinnvoll für unäre Operatoren.

`postfix` der Operator wird als Postfix-Operator gebraucht, er ist als von der Form a op; auch dies ist nur sinnvoll für unäre Operatoren.

Die Definition

```
operator with(x, s);
use as left_associative operator;
x into s; return s;
end with;
```

führt den aus SETL bekannten `with`-Operator wieder ein. Bezeichner für Operator-Namen können abweichend von SETL eine geringfügig erweiterte lexikalische Struktur haben, vgl. Abschnitt 2.4 (p. 5).
Der Benutzer darf vordefinierte Operatoren nicht selbst überladen.

3.6.3 Ausnahmen

Die Ausnahmebehandlung in SETL/E dient dazu, ungewöhnliche Situationen aufzufangen. Jede Ausnahme wird mit einem Namen versehen, dieser Name ist lexikalisch ein Bezeichner. An diesen Namen ist eine Folge von Anweisungen gebunden, ganz ähnlich zur Folge von Anweisungen in einer Prozedur. Wir nennen diese Anweisungsfolge den *Handler* der Ausnahme. Ausnahmen werden im gleichen Teil einer Prozedur definiert wie lokale Prozeduren.

Die Grundidee ist wie folgt. Wenn in einer Prozedur P eine Ausnahme X aktiviert wird, so terminiert diese Prozedur und erhält als Rückgabewert den undefinierten Wert `om`. Nun wird der Handler für X gesucht. Ist der Handler in P vereinbart, so werden seine Anweisungen ausgeführt. Nach ihrer Beendigung wird die Kontrolle an die P aufrufende Prozedur zurückgegeben. Ist der Handler dagegen nicht in P vereinbart, X aber sichtbar, so wird die Kontrolle an die P aufrufende Prozedur zurückgegeben, `om` als Rückgabewert verwendet und der Handler für X wird in dieser aufrufenden Prozedur gesucht. Hier wiederholt sich das gleiche Vorgehen, so daß eine aktivierte Ausnahme jeweils die Kette der gerade aktiven Prozedur-Aufrufe (Konturen) in umgekehrter Reihenfolge durchläuft. Sei diese Kette durch $P = Q_n, \ldots, Q_0$ gegeben, wobei Q_0 das Hauptprogramm ist; Q_i ruft also Q_{i+1} auf. Die aktuellen Parameter der Aktivierung brauchen nicht in $Q_{n-1}, \ldots, Q_0$ sichtbar zu sein.
Der Durchlauf durch $Q_n, \ldots, Q_0$ kann durch die folgenden Ergeignisse angehalten werden:

- der gesuchte Handler wurde in der Kontur Q_j gefunden, wo $j \leq n$. Die Anweisungsfolge wird ausgeführt, danach wird die Kontrolle an Q_{j-1} als den Aufrufer von Q_j zurückgegeben,
- der Name X ist in einem Q_j nicht mehr sichtbar, dann ist der Handler nicht mehr zugänglich, die vordefinierte Ausnahme "`unbehandelte Ausnahme`" aktiviert,
- es ist kein Handler für X vorhanden, weil der Name X verschattet wurde. Es wird auch in dieser Situation die Ausnahme "`unbehandelte Ausnahme`" aktiviert.

Eine Ausnahme kann aktiviert und dynamisch deaktiviert werden; eine deaktivierte Ausnahme kann dynamisch wieder aktiviert werden. Durch `raise` X (Liste der aktuellen Parameter) wird die Ausnahme X aktiviert, durch `deactivate` X wird sie deaktiviert. Das Aktivieren einer deaktivierten Ausnahme durch `raise` hat keinen Effekt. Ausnahmen werden ganz ähnlich zu Prozeduren definiert; sie bilden einen Scope, vgl. Abschnitt 3.2 (p. 7).

<AusnahmeDefn> ⟶ *<AusnahmeHeader><ProcBody><AusnahmeTrailer>*
<AusnahmeHeader> ⟶ `exception` *<AusnahmeIdentif><AusnahmeParamList>* [`;`]
<AusnahmeIdentif> ⟶ *<Bezeichner>*
<AusnahmeParamList> ⟶ [[`(`] (*<ParamName>* [`,`])* *<ParamName>* [`)`]]
<ParamName> ⟶ *<Bezeichner>*
<AusnahmeTrailer> ⟶ `end` *<AusnahmeIdentif>* [`;`]

Die Folge von Anweisungen ist analog zur Anweisungsfolge in Prozeduren zu sehen, insbesondere können darin andere Prozeduren definiert werden und andere Ausnahmen definiert oder aktiviert werden. Parameter können an Ausnahmen ausschließlich als `rd`-Parameter übergeben werden. Ausnahmen werden im Hinblick auf ihre Sichtbarkeit wie alle anderen Objekte behandelt: sie sind sichtbar in dem Scope, in dem sie definiert werden, nicht jedoch in untergeordneten lokalen Prozeduren, außer wenn sie durch `visible` explizit sichtbar gemacht werden. Bezeichner können durch Namen von Ausnahmen verschattet werden. Es ist möglich, die Namen von Ausnahmen selbst zu verschatten, insbesondere können in der System-Umgebung definierte Ausnahmen lokal neu definiert werden.
Ausnahmen sind im Gegensatz zu Prozeduren keine Objekte erster Ordnung. Es ist also nicht möglich, Ausnahmen als Parameter zu übergeben oder als Ergebniswert von Prozeduren zu bekommen oder gar als Elemente von Mengen zu haben.
Wenn in einer Prozedur eine Ausnahme aktiviert wird, so hat dies folgenden Effekt auf die Parameter dieser Prozedur: ein `rd`-Parameter behält seinen Wert, den er vor dem Aufruf gehabt

unär	+, -
binär	+, -, *, **, div, mod, max, min
Prädikate	=, /=, >, <, >=, <=
Operationen in der Standardbibliothek	abs, even, odd, float, random

Tabelle 1: Vordefinierte Operationen: `integer`

hat, bei Parametern, die von der Prozedur geschrieben werden können (also `rw` oder `wr`), wird der undefinierte Wert `om` eingetragen.

3.6.4 Anhang: Die system-Funktion

Wir haben oben im Abschnitt 3.1 (p. 7) gesehen, daß Programme zur Laufzeit mit Parametern versehen werden können. Dies ist eher passiv, denn an die Umgebung kann ja nichts zurückgegeben werden. Anders mit dem `system`[2]-Befehl: hier kann ein beliebiger Befehl an das Betriebssystem abgesetzt werden. `system`(s) enthält das Kommando und seine Parameter in der Zeichenkette s. Das Programm wartet, bis der Befehl vollständig ausgeführt ist, und fährt dann mit seiner Arbeit fort. Ein möglicherweise vorhandener Rückgabewert des in s enthaltenen Befehls wird als Wert des Prozedur-Aufrufs `system` an das SETL/E-Programm weitergegeben.

4 Datentypen

Die folgenden Datentypen sind in SETL/E primitiv: ganze Zahlen, reelle Zahlen, Zeichenketten, persistente Atome, Prozeduren und ein undefinierter Wert; Mengen, Abbildungen und Tupel sind zusammengesetzt, Prozeduren nehmen eine Zwischenstellung ein. ?

4.1 Primitive Datentypen

4.1.1 Ganze Zahlen (integer)

SETL/E stellt ganze Zahlen zur Verfügung, dabei ist der Wertebereich ganzer Zahlen durch die Implementation gegeben. Ganze Zahlen können also nicht beliebig lang sein. Ganze Zahlen sind zwar mathematisch reelle Zahlen, aber sie müssen mit `float` explizit in reelle umgewandelt werden. Tabelle 1 (p. 15) gibt alle Operationen auf ganzen Zahlen an.
Die Operationen `mod` und `div` sind dadurch bestimmt, daß für $x, y \in \mathfrak{Z}, y \neq 0$ mit $q := x$ `div` y und $r := x$ `mod` y gilt $x = qy + r$ mit $0 \leq r < |x|$.

4.1.2 Reelle Zahlen (real)

SETL/E bietet reelle Zahlen mit dem üblichen Hinweis auf ungenaue Darstellung an. Die Darstellung reeller Zahlen ist implementationsabhängig, es wird jedoch garantiert, daß die Operationen mit reellen Zahlen der doppelten Genauigkeit von C auf der gleichen Maschine entsprechen. Die Notation für reelle Zahlen ist bereits oben besprochen worden. Tabelle 2 (p. 16) gibt eine Liste der Operationen auf dem Typ `real` an.

[2] vgl. Kernighan/Ritchie: The C Programming Language, p. 157

unär	+, −
binär	+, −, *, /, **, max, min
Prädikate	=, /=, >, <, >=, <=
Operationen in der Standardbibliothek (binär)	atan2
Operationen in der Standardbibliothek (unär)	abs, fix, floor, ceil, exp, log, cos, sin, tan, acos, asin, atan, tanh, sqrt, random, sign

Tabelle 2: Vordefinierte Operationen: `real`

binär	+, *, in, notin
Prädikate	=, /=, >, <, >=, <=
Konstante	"", die leere Zeichenkette
Extraktion	$s(i)$, $s(i..j)$, $s(i..)$
Operationen in der Standardbibliothek(unär)	abs, str, char
Operationen in der Standardbibliothek(String-Scanning)	span, any, break, len, match, notany, lpad, rspan, rany, rbreak, rlen, rmatch, rnotany, rpad

Tabelle 3: Vordefinierte Operationen: Zeichenketten

4.1.3 Zeichenketten

Eine Zeichenkette ist eine Folge von Zeichen, wobei die mögliche Länge dieser Zeichenkette durch die Implementation beschränkt ist. Eine Zeichenkette wird durch " begrenzt, in der Zeichenkette vorkommende Gänsefüßchen müssen durch \ gekennzeichnet sein. In Konstanten oder Variablen des Typs Zeichenkette dürfen keine nicht sichtbaren Zeichen vorkommen. Die Tabelle 3 (p. 16) gibt eine Liste der Operationen auf Zeichenketten an.

unär	not
binär	and, or
Konstante	true, false

Tabelle 4: Vordefinierte Operationen: `boolean`

4.1.4 Boolesche Werte (`boolean`)

Die Tabelle 4 (p. 17) gibt alle Operationen auf diesem Datentyp an; `impl` ist im Gegensatz zu SETL nicht definiert. Wie in SETL werden die Operationen **and** und **or** als *short circuit*-Operationen ausgeführt, so daß ein Boolescher Ausdruck nur partiell soweit ausgewertet wird, bis sein Ergebnis feststeht. Es gilt

a **and** *b* = **if** *a* **then** *b* **else false end if**

a **or** *b* = **if** *a* **then true else** *b* **end if**

4.1.5 Persistente Atome

Atome sind wie in SETL oder in LISP Objekte, die nicht ein zweites Mal existieren. Während in SETL diese Eindeutigkeit relativ zur Ausführung eines Programms garantiert wird, was dazu führt, daß Atome sinvollerweise nicht eingelesen werden können, wollen wir garantieren, daß Atome in einem absoluten Sinn einmalig sind. Daher können solche Atome auf externe Dateien geschrieben und von diesen Dateien gelesen werden. Ein Atom wird erzeugt durch Aufruf der Standardfunktion `newat()`, auf Atomen sind lediglich Tests auf Gleichheit bzw. Ungleichheit definiert.
Ist *a* ein Atom, so ergibt sich für **str** *a* das folgende Druckbild

$$\#lfdNr.Datum.SystemZeit.MaschinenId$$

Dabei wird dargestellt

- *Datum* in der Form $DD : MM : YY : hh : ss : \mu$
- *SystemZeit* als sechsstellige Zahl — die μ-Sekunden, die seit dem letzten Systemstart vergangen sind
- *MaschinenId* als die Identifikation der Maschine, auf der der Compiler läuft.

Atome können geschrieben und gelesen werden, vgl. 8.3.3 (p. 36), 8.3.2 (p. 35).

4.1.6 Der undefinierte Wert

Ähnlich wie SETL hat SETL/E eine Konstante `om`, die für den undefinierten Wert Ω steht. Im allgemeinen bedeutet das Auftauchen von `om`, daß ein undefinierter Wert benutzt wurde. Der undefinierte Wert darf nicht als Element einer Menge oder als Element des Definitionsbereichs oder Wertebereichs einer Abbildung vorkommen.

4.2 Zusammengesetzte Datentypen

SETL/E bietet wie SETL Mengen, Abbildungen und Tupel als zusammengesetzte Datentypen an, daneben haben Prozeduren Bürgerrechte erster Klasse. Mengen, Abbildungen und Tuple sind nicht notwendig homogen, sondern können aus beliebigen anderen Typen zusammengesetzt sein; insbesondere können auch Prozeduren z.B. in Mengen als Elemente vorhanden sein. Der undefinierte Wert `om` darf als Element eines Tupels vorkommen, nicht jedoch als Element einer Menge oder im Definitionsbereich einer Abbildung.
Anders als SETL bietet SETL/E keine *Data Representation Sublanguage (DRSL)*. Eine geeignete Speicherdarstellung der strukturierten Objekte muß der Compiler/Optimizer gewährleisten können.

4.2.1 Mengen

Mengen haben ihre mathematische Semantik als Kollektion von Objekten, die entweder in der Kollektion sind oder nicht; ein Element kann nicht mehrfach in einer Menge vorkommen, die Reihenfolge des Auftretens ist nicht relevant. Mengen müssen endlich sein, denn sie werden effektiv im Speicher abgebildet.
Auf Mengen sind die üblichen Operationen (Vereinigung, Durchschnitt, Differenz, Potenzmenge, Mächtigkeit) definiert. Mengen können wie folgt notiert werden.

aufzählend: Die Menge aus den Objekten $x_1, \ldots, x_n$ wird wie üblich notiert als

$$\{x_1, \ldots, x_n\}$$

einfaches Intervall: Die einem Intervall $\{j \in \mathcal{Z} : i \leq j \leq k\}$ (mit $i, k \in \mathcal{Z}$[3]) entsprechende Menge wird notiert als

$$\{i \,..k\}$$

Intervall mit Schrittweite: Ein Intervall $\{i + t * m : t \geq 0, i + t * m \leq k\}$ (mit $i, k, m \in \mathcal{Z}$) wird notiert als

$$\{i, i + m \,..k\}$$

deskriptiv: Die Menge

$$\{e : x_1 \texttt{ in } s_1,\ x_2 \texttt{ in } s_2,\ \ldots, x_n \texttt{ in } s_n |\ C\}$$

beschreibt die Menge alle Werte, die der Ausdruck e annimmt, wenn x_1 das Datenobjekt s_1, x_2 das Datenobjekt s_2, ..., x_n das Objekt s_n durchläuft, wobei die Bedingung C erfüllt sein muß. Syntaktisch kann der Teil

$$| C$$

weggelassen werden, dann wird als Bedingung `true` angenommen, ebenfalls kann bei $n = 1$ der Ausdruck

$$e :$$

weggelassen werden, dann wird e als Identität angenommen.

Tabelle 5 (p. 19) beschreibt die Mengenoperationen; dabei ist `pow`(A) die Potenzmenge der Menge A, `arb` und `select` dienen der (nicht-)deterministischen Auswahl, vgl. 6.10 (p. 31).

[3] $\mathcal{Z}$ ist die Menge der ganzen Zahlen

unär	#, pow, arb,select
binär	+, −, *, mod
Prädikate	=, /=, in, notin, subset
Zuweisungen	from, into
Bildung von Mengen	s. 4.2.1 (p. 18)
Konstante	{ }, die leere Menge
Operationen in der Standard-bibliothek	npow

Tabelle 5: Vordefinierte Operationen: Mengen

4.2.2 Tupel

Tupel haben ihre mathematische Semantik als geordnete Kollektion von Objekten; ein Element kann mehrfach in einem Tupel vorkommen, die Reihenfolge des Auftretens ist relevant. Tupel müssen endlich sein, denn sie werden effektiv im Speicher abgebildet. Konzeptionell wird ein Tupel angenommen als ein unendlich langer Vektor, der fast überall den Wert om enthält.
Auf Tupeln sind die üblichen Operationen (Konkatenation, Extraktion, Einfügen, *slicing*, Länge) definiert. Die Komponenten eines Tupels werden stets mit 1 beginnend gezählt, seine Länge ist die Position der letzten Komponente, die von om verschieden ist. Analog zu Mengen können Tupel wie folgt notiert werden:

aufzählend: Das Tupel aus den Objekten $x_1, \ldots, x_n$ wird wie üblich notiert als

$$[x_1, \ldots, x_n]$$

einfaches Intervall: Das einem Intervall $i \ldots k$ (mit $i, k \in \mathcal{Z}$) entsprechende Tupel wird notiert als

$$[i \,..\, k]$$

Intervall mit Schrittweite: Ein Intervall $\{i + t * m : t \geq 0, i + t * m \leq k\}$ (mit $i, k, m \in \mathcal{Z}$) wird als Tupel so notiert:

$$[i, i + m \,..\, k]$$

beschreibend: Das Tupel

$$[e : x_1 \texttt{ in } s_1,\ x_2 \texttt{ in } s_2,\ \ldots, x_n \texttt{ in } s_n |\ C]$$

beschreibt den Vektor aller Werte, die der Ausdruck e annimmt, wenn x_1 das Datenobjekt s_1, x_2 das Datenobjekt s_2, ..., x_n das Objekt s_n durchläuft, wobei die Bedingung C erfüllt sein muß. Syntaktisch kann der Teil

$$| C$$

ausgelassen werden, dann wird als Bedingung **true** angenommen, ebenfalls kann bei $n = 1$ der Ausdruck

$$e :$$

weggelassen werden, dann wird e als Identität angenommen.

unär	#, `arb`, `select`
binär	+, −, *,
Prädikate	=, /=, `in`, `notin`
Extraktion	$t(i)$, $t(i..j)$, $t(i..)$
Zuweisungen	`frome`, `fromb`, `into`
Bildung von Tupeln	s. 4.2.2 (p. 19)
Konstante	[ ], das leere Tupel; `argv`, Programm-Parameter

Tabelle 6: Vordefinierte Operationen: Tupel

unär	#, `domain`, `range`
binär	`lessf`
Prädikate	=, /=, `in`, `notin`
Extraktion	$f(i)$, $f\{i\}$, $f(i_1,\ldots,i_n)$, $f\{i_1,\ldots,i_n\}$

Tabelle 7: Zusätzlich definierte Operationen: Abbildungen

Tabelle 6 (p. 20) beschreibt die Tupeloperationen, hierbei sind `frome`, `fromb` und `into` im Abschnitt 6.2 (p. 26) näher beschrieben.
Die Extraktion erfolgt durch $t(i)$, $t(i..j)$ bzw. $t(i..)$ mit $0 < i <= j$.

4.2.3 Abbildungen

Abbildungen sind als Mengen von Paaren $[x, y]$ mit $x \neq$ `om`, $y \neq$ `om` definiert, es können also alle Mengenoperationen angewandt werden. Zusätzlich können die in der Tabelle 7 (p. 20) aufgeführten Operationen durchgeführt werden. Die Extraktions-Operationen können auf der linken wie auf der rechten Seite stehen, vgl. Abschnitt 5.1 (p. 22). In der Tabelle bezieht sich `domain`(f) auf den *Definitionsbereich* der Abbildung f, d.h. auf die Menge $\{e(1) :$ e `in` $f\}$, `range`(f) auf den *Wertebereich* der Abbildung f, d.h. auf die Menge $\{e(2) :$ e `in` $f\}$. `lessf` ist ein binärer Operator; f `lessf` x entfernt alle Paare mit erster Komponente x aus der Abbildung f, so daß z.B. $f(x) := y$ gleichwertig ist zu f `lessf` x; $[x,\ y]$ `into` f.

4.2.4 Prozeduren

Prozeduren sind in SETL/E selbständige Objekte. Es ist also u.a. erlaubt, Prozeduren als

- Werte an Variablen zuzuweisen,
- Elemente bzw. Komponenten in strukturierte Objekte einzufügen,
- Parameter an andere Prozeduren zu übergeben,
- Werte von Prozeduraufrufen zurückzuerhalten,

- selbständige Objekte in Ausdrücken zu verwenden.

Um bei parameterlosen Prozeduren zwischen Aufruf und Zuweisung unterscheiden zu können, muß der Ausruf stets mit einer leeren Parameter-Liste erfolgen:

> $y := f()$; weist y den Wert des Prozedur-Aufrufs $f()$ zu
> $y := f$; weist y als Wert die anonymisierte Prozedur f zu

Jeder Prozedur ist (implizit) eine anonyme Prozedur zugeordnet, vgl. 3.6.1 (p. 11). Auf Prozeduren sind lediglich drei Standard-Operationen definiert: `str`, # und =. `str` p konstruiert die Darstellung der zur Prozedur p gehörigen anonymen Prozedur, # p gibt die Anzahl der Parameter von p an. $p_1 = p_2$ ist äquivalent zu `str` p_1 = `str` p_2

Bindungen Die Bindung von Werten an Prozedur-Aufrufe ist zu diskutieren. Es muß geklärt werden, was geschieht, wenn Prozeduren auf nicht-lokale Objekte zugreifen, die jedoch in ihrem definierenden Scope sichtbar sind. Beim Aufruf einer Prozedur p wird eine Kontur γ für p innerhalb der Kontur σ des statischen Vorgängers konstruiert; dies muß ggf. wiederholt werden, falls p relativ zur aufrufenden Prozedur tief verschachtelt ist. Alle diese Konturen werden auf den Konturen-Stack gelegt, so daß γ die aktuelle Kontur oben auf dem Stack ist. Die Sichtbarkeit von Bezeichnern für p wird relativ zur Kontur γ mit Berücksichtigung des Scope-Baumes bestimmt. Wird p verlassen, so werden γ und alle umgebenden Konturen vom Konturen-Stack genommen, und die Kontur σ tritt wieder in ihre Rechte ein.
Als Beispiel betrachte man:

```
procedure f;
visible x := 1;
return g;
 --
procedure g;
put("x = %d\n", x);
end g;
 --
end f;
```

Wird unter diesen Voraussetzungen ausgeführt

```
y := f(); y();
```

so greift y auf den in f lokal sichtbaren Wert der Variablen x zu, und es sollte beim Aufruf $y()$ der Wert 1 ausgedruckt werden. Aus dem Konturen-Modell folgt, daß jede Prozedur bei ihrem Aufruf auf die *jeweils gültigen* Werte in ihrem Bindungsbereich zugreift, auch wenn sie als Wert eines anderen Prozeduraufrufs zurückgegeben wird.
Das folgende Beispiel erläutert den Sachverhalt im Zusammenhang mit der Übergabe von `rw`-Parametern durch *call by value/result*:

```
t := 0;
x := f(t);
x();
x();
put("main: t = %d\n", t);
 --
procedure f(rw t);
  t := t + 1;
  return g;
   -- Beginn von g
  procedure g;
    t := t + 1;
    put("g: t = %d\n", t);
  end g;
   -- Ende von g
end f;
```

Man erhält als Ergebnis

```
g:  t = 2
g:  t = 2
main:  t = 1
```

5 Ausdrücke

Wir unterscheiden bei Ausdrücken solche, die einen *l-value* haben, die also z.B. auf der linken Seite einer Zuweisung stehen dürfen, von solchen, die einen *r-value* haben.

5.1 Ausdrücke mit *l-values*

Wir geben hier eine kurze Übersicht über Ausdrücke mit *l-values*, die syntaktische Form dieser Ausdrücke ist in Abschnitt 6.2 (p. 26) beschrieben.

1. Bezeichner
2. aufzählende Tupel, also Tupel der Form

   $$[x, y, \ldots, z]$$

   Hierbei sind x, y und z Ausdrücke mit *l-values*.
3. Auswahl des i^{ten} Buchstabens $s(i)$ einer Zeichenkette s,
4. Unterbereiche bei Zeichenketten, also $s(i..j)$ oder $s(i..)$
5. Auswahl der i^{ten} Komponente $t(i)$ eines Tupels t,
6. Unterbereiche bei Tupeln, also $t(i..j)$ oder $t(i..)$
7. Berechnung des Wertes $f(x)$ einer (eindeutigen) Abbildung f an einer Stelle x
8. Berechnung des Wertes $f\{x\}$ einer (mehrdeutigen) Abbildung f an einer Stelle x

Präzedenz	Operatoren
8	Unäre Operatoren mit Ausnahme von **not** und **type**
7	**
6	*, **mod**, **div**
5	+, −, **max**, **min**
4	alle Operatoren, die hier nicht explizit aufgeführt werden
3	=, /=, <, <=, >, >, =, **in**, **notin**, **subset**, **incs**
2	**not** und **type**
1	**and**
0	**or**

Tabelle 8: Operator-Präzedenzen

9. Berechnung des Wertes

$$f(x_1, \ldots, x_n)(\text{abgekürzt für } f([x_1, \ldots, x_n]))$$

einer (eindeutigen) Abbildung f an einer Stelle $[x_1, \ldots, x_n]$

10. Berechnung des Wertes

$$f\{x_1, \ldots, x_n\}(\text{abgekürzt für } f\{[x_1, \ldots, x_n]\})$$

einer (mehrdeutigen) Abbildung f an einer Stelle $[x_1, \ldots, x_n]$

Die in 5. – 10. angesprochenen Tupel bzw. Abbildungen müssen selbst wieder legale *l-values* sein.

5.2 Andere Ausdrücke

Wir geben eine kurze Übersicht über die weiteren Ausdrücke:

- Literal-Konstanten und vordefinierte Werte wie **true**, **om**, **atom**,
- Unäre und binäre Operationen gemäß der Tabelle 8 (p. 23)
- Prozeduraufrufe $p(x_1, x_2, \ldots, x_n)$ bzw.$p()$ (Die Klammern sind beim Aufruf einer parameterlosen Prozedur notwendig!)
- Bedingte und fallgesteuerte Ausdrücke, vgl. 6.6 (p. 30), 6.7 (p. 30)
- Quantifizierte Ausdrücke: **exists**, **notexists** bzw. **forall**, vgl. 6.8 (p. 31)
- Tupel- und Mengenformer:

$[x \in s | C(x)]$
$[E(x) : x \in s | C(x)]$
$[i..j]$
$[i, j..k]$

und

$$\{x \in s | C(x)\}$$
$$\{E(x) : x \in s | C(x)\}$$
$$\{i..j\}$$
$$\{i, j..k\}$$

- Sonstige: Ergebnis der Anwendung des *compound operators* op/*s*, wenn op ein binärer Operator ist. Hierbei verknüpft op/*s* die Elemente oder Komponenten von *s* mittels op, also ergibt

  t := +/{1, 2, 3}

  für *t* den Wert 6.

5.3 Überblick über die Syntax

<*ExprList*> ⟶ <*Expression*>(' <*Expression*>)*

<*Expression*> ⟶ { <*Constant*> | <*Identifier*>(<*Selector*>)* | <*Identifier*> (<*ExprList*>) | <*Unary*> | <*Binary*> | <*Compound*> | (<*Expression*>) | <*Forall*> | <*Exists*> | <*IfExpr*> | <*CaseExpr*> | <*Tuple*>(<*Selector*>)* | <*Set*>(<*Selector*>)* }

<*Constant*> ⟶ { <*IntTok*> | <*RealTok*> | <*StringTok*> | <*SysVal*> }

<*Unary*> ⟶ <*UnOp*><*Expression*>

<*Binary*> ⟶ <*Expression*><*BinOp*><*Expression*>

<*Compound*> ⟶ <*BinOp*> / <*Expression*>

$$\langle Tuple \rangle \longrightarrow [\ \left\{ \begin{array}{c} \langle ExprList \rangle \\ \langle Former \rangle \end{array} \right\}\ ]$$

$$\langle Set \rangle \longrightarrow \{\ \left\{ \begin{array}{c} \langle ExprList \rangle \\ \langle Former \rangle \end{array} \right\}\ \}$$

$$\langle Former \rangle \longrightarrow \left\{ \begin{array}{c} \langle Expression \rangle\ :\ \langle Iterator \rangle \\ \langle SimpleIt \rangle\ [\ |\ \langle Expression \rangle\ ] \\ \langle Expression \rangle\ [\ ,\ \langle Expression \rangle\ ]\ ..\ \langle Expression \rangle \end{array} \right\}$$

5.4 Änderungen zu SETL

Im Vergleich zu SETL ergeben sich folgende Änderungen

- In SETL/E gibt die Zuweisung keinen Wert mehr zurück, also sind Operationen der Form

  $r := s := t;$
  und
  $r :=$ `s fromb` $t;$

  illegal.

- Die Operatoren der Form `is_xx` zum Typtest fallen weg, der Typoperator `type` liefert als Ergebnis die zu dem Typ gehörige Typkonstante, die ein vordefinierter Werte vom Typ Atom ist.

- Der `?`-Operator fällt weg. Er kann bei Bedarf durch einen `if`-Ausdruck simuliert werden.

- Konstantendeklarationen werden dynamisch ausgewertet, vgl. Abschnitt 3.4 (p. 9).

6 Anweisungen

Anweisungen sind Sprachkonstrukte, die den Daten- und Kontrollfluß steuern.

$$\langle Stmt \rangle \longrightarrow \langle Statement \rangle$$

6.1 Anweisungen: Überblick

Die folgende Produktion gibt die möglichen Anweisungen im Überblick an:

<*Statement*> ⟶ { <*LabelStmt*> | <*NoLabelStmt*> | <*ProcCall*> | **return** <*Expression*> | { **continue** | **quit** } [<*Label*>] | **stop** | **pass** | <*Assignment*> } ;

<*Label*> ⟶ <*Identifier*>

Die **pass**-Anweisung ist die leere Anweisung. Sie hat keinen Wirkungen auf den Kontroll- und Datenfluß. Die **stop**-Anweisung bricht die Ausführung des Programms ab, sorgt dafür, daß aufgeräumt wird (z.B. schließt die noch offenen Dateien) und gibt die Kontrolle an das Betriebssystem zurück. Die **return**-Anweisung ist im Abschnitt 3.6 (p. 9) diskutiert worden. Auf die anderen Anweisungen wird in den nächsten Abschnitten eingegangen.

6.2 Zuweisungen

Die syntaktische Form einer Zuweisung sieht wie folgt aus:

<*Assignment*> ⟶ { <*LValue*>[<*Binop*>] := <*Expression*> | <*LValue*> { **from** | **fromb** | **frome** } <*LValue*> | <*Expression*> **into** <*LValue*> }

Eine Zuweisung hat zur Folge, daß der Wert des rechten Ausdrucks zum Wert der linken Seite wird. Somit dürfen auf der linken Seite nur Ausdrücke stehen, die einen *l-value* haben, vgl. Abschnitt 5.1 (p. 22).

<*LValue*> ⟶ { <*Identifier*>(<*Selector*>)* | [{ <*LValue*> | - } (, { <*LValue*> | - })*] }

<*Selector*> ⟶ { { <*ExprList*> } | ({ <*ExprList*> | <*Expression*> .. [<*Expression*>] }) }

Anmerkungen:

- Die Semantik der Links-Werte ist aus SETL übernommen, es gibt jedoch keine Zuweisungs-Ausdrücke.
- Die Links-Werte in Iteratoren sind im Gegensatz zu SETL lokal zum jeweiligen syntaktischen Konstrukt (siehe z.B. Abschnitt 7 (p. 31)).
- Die `from`-Ausdrücke werden durch `from`-Anweisungen ersetzt. Sie liefern also keine Werte. Ansonsten wird die Semantik aus SETL übernommen: `from` arbeitet auf Mengen, `fromb` und `frome` arbeiten auf Tupeln. Ist s eine Menge, so liefert

  $$x \texttt{ from } s$$

  in x ein beliebiges Element der Menge s, um das s vermindert wird. Ist s die leere Menge, so hat x den Wert `om`. Ist t ein Tupel, so ist

  $$x \texttt{ fromb } t$$

  gleichwertig mit

  $$x := t(1); t := t(2..).$$

  Analog für `frome`:

  $$x \texttt{ frome } t$$

  ist gleichwertig mit

  $$x := t(\#t);\ t := t(1..\#t-1)$$

- Der `into`-Operator erlaubt es, ein Element in eine Menge oder ein Tupel zu geben. Ist s eine Menge, und $x \neq$ `om`, so ist x `into` s gleichwertig zu $s := s + \{x\}$, für ein Tupel t ist x `into` t gleichwertig mit $t := t + [x]$ (Kenner sehen die Abstammung von `into` aus SETL's `with`). `into` liefert keinen Wert, sondern ist auf den Seiteneffekt abgestellt.

6.3 Kontrollstrukturen

Kontrollstrukturen sind statisch im Programmtext eingebundene Konstrukte, die den Kontrollfluß zur Laufzeit eines Programms steuern. Zu ihnen gehören die wiederholten Anweisungen (Schleifen), die bedingten Anweisungen und quantifizierte Ausdrücke (als implizite Schleifen). Für einige dieser Strukturen gibt es in SETL/E benannte und unbenannte Varianten. Auf die Unterschiede wird weiter unten näher eingegangen.
Die Syntax für die Kontrollstrukturen in der erweiterten BNF ist wie folgt:

$$\langle Statement \rangle \longrightarrow \left\{ \begin{array}{c} \langle LabelStmt \rangle \\ \langle NoLabelStmt \rangle \end{array} \right\} \boxed{;}$$

Benannte Strukturen werden einheitlich durch ihren Namen (<*Label*>) abgeschlossen. Dieser Name ist nur innerhalb der Struktur für `quit`- oder `continue`-Anweisungen sichtbar (Abschnitt 6.5 (p. 29)). Nach '`end`<*Label*> ;' ist er nicht mehr sichtbar und kann neu verwendet werden. Umgekehrt dürfen keine aktuell sichtbaren Namen zum Benennen von Blöcken verwendet werden.

$$\langle LabelStmt \rangle \longrightarrow \langle Label \rangle \boxed{:} \left\{ \begin{array}{c} \langle LoopStmt \rangle \\ \langle WhileStmt \rangle \\ \langle UntilStmt \rangle \\ \langle ForStmt \rangle \end{array} \right\} \texttt{end } \langle Label \rangle$$

Diese Blöcke müssen nicht notwendig benannt werden. Unbenannte Blöcke müssen allerdings mit ihrem einleitenden Schlüsselwort abgeschlossen werden.

$$\langle \mathit{NoLabelStmt} \rangle \longrightarrow \left\{ \begin{array}{c} \langle \mathit{LoopStmt} \rangle \texttt{end loop} \\ \langle \mathit{WhileStmt} \rangle \texttt{ end while} \\ \langle \mathit{UntilStmt} \rangle \texttt{ end until} \\ \langle \mathit{ForStmt} \rangle \texttt{ end for} \\ \langle \mathit{IfStmt} \rangle \texttt{ end if} \\ \langle \mathit{CaseStmt} \rangle \texttt{ end case} \end{array} \right\}$$

Wir diskutieren nun die einzelnen Kontrollstrukturen.

6.4 Schleifen

Es gibt benannte und unbenannte Schleifen. Bis auf die Sprungziele für `quit`- und `continue`-Anweisungen (Abschnitt 6.5 (p. 29)) sind diese jedoch semantisch äquivalent. Der syntaktische Kern ist für beide Varianten gleich.

6.4.1 `loop`

Die Endlosschleife hat die folgende Form:

$$\langle \mathit{LoopStmt} \rangle \longrightarrow \texttt{loop } (\langle \mathit{Statement} \rangle)^+$$

Die Anweisungsliste wird wiederholt. Diese Schleife kann nur durch eine `quit`-Anweisung verlassen werden (Abschnitt 6.5 (p. 29)).

6.4.2 `while`

$$\langle \mathit{WhileStmt} \rangle \longrightarrow \texttt{while } \langle \mathit{Expression} \rangle \texttt{do}(\langle \mathit{Statement} \rangle)^+$$

Die übliche Semantik: die Anweisungsliste wird ausgeführt, falls die Bedingung, die jeweils beim Eintritt überprüft wird, erfüllt ist (<*Expression*> liefert `true`). Dies wird solange wiederholt, bis die Bedingung nicht mehr erfüllt ist.

$$\texttt{while } E \texttt{ do } S \texttt{ end while}$$

genügt also der Fixpunktgleichung

$$\texttt{if } E \texttt{ then } S\texttt{; while } E \texttt{ do } S \texttt{ end while; end if}$$

6.4.3 `until`

$$\langle \mathit{UntilStmt} \rangle \longrightarrow \texttt{do}(\langle \mathit{Statement} \rangle)^+ \texttt{ until } \langle \mathit{Expression} \rangle$$

Die übliche Semantik: die Anweisungsliste wird ausgeführt, erst dann wird der bedingte Ausdruck ausgewertet. Dies wird solange wiederholt, bis die Bedingung erfüllt ist (<*Expression*> liefert `true`).

$$\texttt{do } S \texttt{ until } E \texttt{ end until}$$

genügt also der Fixpunktgleichung

$$S\texttt{; if not } E \texttt{ then do } S \texttt{ until } E \texttt{ end until; end if;}$$

6.4.4 Die Zählschleife

Syntaktische Form:

<*ForStmt*> ⟶ `for` <*Iterator*> `do` (<*Statement*>)$^+$ `end of for loop marker?`

Bei der expliziten Iteration nehmen Iterationsvariablen sukzessive die Werte der einzelnen Elemente von Mengen, Tupeln oder Zeichenketten an, über die iteriert werden soll, wobei jeweils über eine Kopie iteriert wird. Die Anweisungsliste wird für jeden Durchlauf ausgeführt. Die Iterationsvariablen sind lokal zu ihrer Schleife (siehe Abschnitt 7 (p. 31)).
Iteratoren haben dabei die folgende Form:

<*Iterator*> ⟶ <*SimpleIt*>(, <*SimpleIt*>)*[| <*Expression*>]

<*SimpleIt*> ⟶ { <*LValue*> `in` <*Expression*> | <*LValue*> = <*MapId*> { (<*LValue*>(, <*LValue*>)*) | { <*LValue*>(, <*LValue*>)* } } }

<*LValue*> sind also die Iterationsvariablen, die grammatisch *l-values* sein müssen, vgl. Abschnitt 6.2 (p. 26). <*MapId*> muß der Name einer ein- oder mehrwertigen Abbildung sein.
Die Angabe von mehreren einfachen Iteratoren in der Liste entspricht der impliziten Schachtelung von entsprechenden `for`-Schleifen. Der am weitesten rechts genannte Iterator entspricht der innersten Schleife. `quit` und `continue` beziehen sich in diesem Fall auf die gesamte Schleife.
Wir verzichten auf `for`-Schleifen nach PASCAL-Art. Diese Schleifenart kann durch die Iteration über konstante Tupel ersetzt werden.

6.5 quit und continue

Syntaktisch:

<*Statement*> ⟶ { `continue` | `quit` } [<*Label*>] ;

Diese Anweisungen sind nur innerhalb von Schleifen sinnvoll und erlaubt. <*Label*> muß ein gültiger Name für einen Schleifen-Block sein, der dann als Referenz benutzt wird. Mit `quit` wird die jeweilige Schleife sofort verlassen und mit der nächsten auf die Schleife folgenden Anweisung fortgefahren. Mit `continue` wird der Rest der Anweisungsliste überschlagen. In einer `loop`-Schleife wird wieder die erste Anweisung der Liste ausgeführt, bei einer `while`- oder `until`-Schleife wird dann erneut die Bedingung ausgewertet, und bei einer `for`-Schleife wird mit dem nächsten Iterationsschritt fortgefahren. Ist die `for`-Schleife implizit geschachtelt, so wird mit dem nächsten *äußeren* Iterationsindex weitergemacht. `quit for`; etc. wie in SETL gibt es in SETL/E nicht: ohne Marke ist immer die jeweils innerste Schleife gemeint.

6.6 Bedingte Anweisungen und Ausdrücke

Bedingte Anweisungen haben syntaktisch die folgende Form:

```
<IfStmt> ⟶ if <Expression> then (<Statement>)+
           (elseif <Expression> then (<Statement>)+ )*
           [else (<Statement>)+]
```

Die Semantik der `if`-Anweisung entspricht der in SETL: die erste Bedingung wird ausgewertet; ist sie wahr, so wird die entsprechende Anweisung ausgeführt und die Anweisung verlassen, ist sie falsch, so werden analog die `elseif`-Zweige nacheinander ausgewertet, ggf. die entsprechenden Anweisungen ausgeführt und die Anweisung verlassen. Ist schließlich keine der Bedingungen wahr, so wird die im `else`-Zweig stehende Anweisung ausgeführt, falls sie vorhanden ist.
`if`-Anweisungen können auch benannt werden. Das hat den Charakter eines Kommentars und wurde eingeführt, um die Syntax der Kontroll-Strukturen einheitlich zu gestalten. Das gleiche gilt für `case`-Anweisungen (Abschnitt 6.7 (p. 30)).
Bedingte Ausdrücke liefern Werte, sie haben die folgende Form:

```
<IfExpr> ⟶ if <Expression> then <Expression>
           (elseif <Expression> then <Expression>)*
           [else <Expression>]
           end if
```

`else` om ist der default, falls der `else`-Zweig fehlt. Das gleiche gilt für `case`-Ausdrücke (Abschnitt 6.7 (p. 30)).

6.7 Die fallgesteuerte Anweisung

Die Semantik der Anweisung ist analog der entsprechenden SETL-Anweisung: <*ConstList*> ist jeweils eine Liste von Konstanten, die zur Zeit der Auswertung der Anweisung elaboriert sein müssen. Die Listen müssen nicht notwendig disjunkt sein. Der Ausdruck wird ausgewertet, und diejenige Folge von Anweisungen wird ausgeführt, deren Liste den Wert enthält; liegt der Ausdruck in mehreren Listen, so wird nicht-deterministisch eine ausgewählt, und die zugehörige Anweisung ausgeführt.

```
<CaseStmt> ⟶ case <Expression> of ([(]<ConstList>[)][:](<Statement>)+ )+
             [else (<Statement>)+]
```

Die Grammatik für fallgesteuerte Ausdrücke ist wie folgt:

```
<CaseExpr> ⟶ case <Expression> of <SingleEx>([,]<SingleEx>)*
             [else <Expression>]
             end case
<SingleEx> ⟶ ([(]<ConstList>[)][:]<Expression>)+
```

In Abschnitt 6.6 (p. 30) Gesagtes gilt hier entsprechend.
Man beachte, daß die `else`-Zweige in SETL obligat und in SETL/E optional sind.

6.8 Quantifizierte Ausdrücke: Allgemein

Quantifizierte Ausdrücke sind implizite Schleifen, die Werte vom Typ `boolean` liefern. Wie bei den `for`-Schleifen (Abschnitt 6.4.4 (p. 29)) sind die Iterationsvariablen lokal zur Struktur.

6.9 `exists,forall`

Der `exists`-Ausdruck liefert lediglich einen Booleschen Wert, der durch eine Bedingung bestimmt wird. Es sei angemerkt, daß diese Anweisung keinen Seiteneffekt in dem Sinne auslöst, daß implizit ein existierender Wert bestimmt wird, weil dies in SETL der Fall ist. Hierzu dienen `arb` und `select`, vgl. 6.10 (p. 31).

$$\langle Exists \rangle \longrightarrow \left\{ \begin{array}{c} \texttt{exists} \\ \texttt{notexists} \end{array} \right\} \langle SimpleIt \rangle (\boxed{,} \langle SimpleIt \rangle)^* \boxed{|} \langle Expression \rangle$$

Im Unterschied zur `for`-Schleife (Abschnitt 6.4.4 (p. 29)) ist hier die Angabe einer Bedingung obligat. Der Ausdruck liefert für `exists true`, falls der Iterator nicht die leere Menge liefert. Entsprechendes gilt für `notexists`.
Der `forall`-Ausdruck liefert ebenfalls lediglich einen Booleschen Wert. Er ist natürlich äquivalent zu `notexists` mit negierter Bedingung.

$$\langle Forall \rangle \longrightarrow \texttt{forall}\ \langle SimpleIt \rangle (\boxed{,} \langle SimpleIt \rangle)^* \boxed{|} \langle Expression \rangle$$

6.10 Auswahlen

Wir stellen zwei Auswahloperatoren zur Verfügung: `arb` für die deterministische und `select` für die nicht-deterministische Auswahl. `arb` ist wie in SETL definiert und wählt ein beliebiges Element aus einer Menge oder einem Tupel aus: es wird garantiert, daß bei gleichem Iterator die Resultate verschiedener Anwendungen von `arb` die gleichen sind. `select` entspricht `arb*` bei Sharir[4] und ist die nicht-deterministische Auswahl, die bei gleichbleibendem Argument bei verschiedenen Aufrufen verschiedene Ergebnisse haben kann, wobei jeder Kandidat die gleiche Chance hat, ausgewählt zu werden.

$$\langle ArbSelect \rangle \longrightarrow \left\{ \begin{array}{c} \texttt{arb} \\ \texttt{select} \end{array} \right\} \langle Expression \rangle$$

7 Blockstruktur

Die Blockstruktur dient im wesentlichen dazu, Objekte implizit als lokal behandeln zu können. Dies hat den Vorzug der besseren Lesbarkeit, aber auch der Eindeutigkeit, da lokale Objekte gleichnamige in einem übergeordneten Scope verschatten können.

[4] Some Observations Concerning Formal Differentiation of Set Theoretic Expressions, ACM TOPLAS 4, 2, 1982, 196 – 225

7.1 Mengen, Tupel, Abbildungen

Mengen werden in der allgemeinen Form

$$\{e:\ x_1\ \texttt{in}\ s_1,\ \ldots\ x_n\ \texttt{in}\ s_n|\ C\}$$

notiert, vgl. 5.3 (p. 24). Dann sollen $x_1, \ldots, x_n$ lokal zu diesem Block sein, alle anderen Objekte ($s_1, \ldots, s_n$ und C) sollen im Block sichtbar sein. Wie üblich kann geschachtelt werden:

$$\{x\ \texttt{in}\ \{1\ \ldots\ 10\}|\ x\ \texttt{notin}\ \{x\ \texttt{in}\ \{1,\ 2,\ \ldots\ 20\}\}\}$$

Analog sollen Tupel und Abbildungen behandelt werden.

7.2 Schleifen und Iteratoren

Allgemein legen wir fest:

1. die Iterationsvariablen sind lokal zu ihrer Schleife
2. Objekte, über die iteriert wird, können in der Schleife geändert werden. Vor dem Eintritt in die Schleife wird eine lokale Kopie angefertigt, über die iteriert wird, und die nach der Iteration verloren ist. Änderungen beziehen sich auf das Objekt selbst, also analog zum *call by value/result*, vgl. Abschnitt 3.6 (p. 9).
3. Nach Verlassen der Schleife ist der Index nicht mehr definiert (auch nach Verlassen mittels `quit`). Geänderte Objekte erhalten ihren temporären Wert erst nach Verlassen der Schleife; dies gilt auch für das Verlassen mittels `quit`.

7.3 Quantoren, Auswahl

Die gebundenen Variablen für den Gültigkeitsbereich eines Quantors sind lokal. Die Auswahl wird wie ein Quantor behandelt. Also wählt z.B.

```
x := 3;
y := select x in S | p(x);
put(x);
```

nicht-deterministisch ein x aus S, für das $p(x)$ gilt, und druckt anschließend den Wert 3.

8 Standard-Bibliothek

Wir haben einige Operationen, die zum Sprachumfang von SETL gehören, aus der Sprache herausgenommen und in die Standard-Bibliothek verlagert. Das hat zur Folge, daß der Sprachumfang geringer geworden ist, aber auch, daß vorher fest vorgegebene Operationen vom Benutzer neu definiert werden können. Dies betrifft insbesondere die Ein- und Ausgabeoperationen, die ebenfalls in diese Bibliothek verlagert wurden. Gegenwärtig ist geplant, daß die Standard-Bibliothek automatisch zum SETL/E-System zugeladen wird; das kann sich ändern.
Wir diskutieren zunächst Operationen, die Objekte (`integer`, `real` und Zeichenketten) manipulieren, dann Ein- und Ausgabeoperationen.

8.1 Manipulation von Standard-Objekten

8.1.1 integer

`abs`	Absoluter Betrag einer ganzen Zahl
`even`	Prädikat: das ganzzahlige Argument ist gerade
`odd`	Prädikat: das ganzzahlige Argument ist ungerade
`float`	Konversion `integer` → `real`
`random`	gleichverteilte, ganzzahlige Zufallszahl zwischen 0 und n

8.1.2 real

`acos`	arcus cosinus
`asin`	arcus sinus
`atan`	arcus tangens
`atan2`	arcus tangens: atan2(x, y) = atan(x/y)
`abs`	absoluter Betrag
`ceil`	ceiling: $x \mapsto \lceil x \rceil$
`cos`	cosinus
`exp`	e^x
`fix`	$x \mapsto$ `if` $x \geq 0$ `then` $floor(x)$ `else` $ceil(x)$ `end if`
`floor`	floor: $x \mapsto \lfloor x \rfloor$
`log`	natürlicher Logarithmus
`random`	Zufallszahl
`sign`	-1, 0, +1, je nach Vorzeichen
`sin`	der andere alte Römer
`sqrt`	Quadratwurzel
`tan`	tangens
`tanh`	tangens hyperbolicus

8.1.3 Zeichenketten

Wir diskutieren zunächst die *string scanning primitives*, die wir aus SETL (und damit aus SNOBOL[5]) übernehmen, und geben dann tabellarisch die anderen vordefinierten Funktionen an.

String scanning primitives Seien s und ss Zeichenketten. Es sei an dieser Stelle darauf hingewiesen, daß die zu diskutierenden Funktionen ihre Parameter (`rw`) ändern können.

`any`: `any`(s, ss) liefert $s(1)$, falls $s(1)$ in ss vorkommt, und modifiziert in diesem Falle s zu $s(2..)$. Falls dagegen $s(1)$ nicht in ss vorkommt, bleibt s unverändert, und `om` wird zurückgegeben

`break`: `break`(s, ss) bricht von s die längste initiale Kette ab, deren Buchstaben nicht in ss sind, und gibt diese Kette zurück; s wird entsprechend modifiziert. Falls jedes Zeichen in s auch in ss enthalten ist, wird `om` zurückgegeben, und s bleibt unverändert

`len`: `len`(s, n) für $n \in \mathbb{Z}$ wird $s(1..n)$ zurückgegeben, und s zu $s(n+1..)$ modifiziert. Falls $n \leq 0$ oder $\#s \leq n$ bleibt s unverändert, und und `om` wird zurückgegeben

[5] R. E. Griswold, J. F. Poage und I. P. Polonsky: The SNOBOL4 Programming Language (second edition), Prentice-Hall, Englewood Cliffs, NJ, 1971

d: `lpad`(s, n) für $n \in \mathbb{Z}$ wird s von links mit Leerzeichen aufgefüllt, so daß eine Zeichenkette der Länge n entsteht. Ist $n < \#s$, wird s unverändert zurückgegeben

ch: `match`(s, ss) gibt ss zurück, falls $\#ss \leq \#s$ und $ss = s(1..\#ss)$; s wird in diesem Falle zu $s(\#ss + 1..)$ modifiziert. Falls die Bedingung nicht erfüllt ist, bleibt s unverändert, und `om` wird zurückgegeben

any: `notany`(s, ss) liefert $s(1)$, falls $s(1)$ in ss nicht vorkommt, und modifiziert in diesem Falle s zu $s(2..)$. Falls dagegen $s(1)$ in ss vorkommt, bleibt s unverändert, und `om` wird zurückgegeben

n: `span`(s, ss) findet das längste Anfangsstück von s, dessen Zeichen alle (in beliebiger Reihenfolge) in ss sind, und gibt dieses Anfangsstück als Wert zurück; s wird um dieses Stück gekürzt. Falls kein Anfangsstück gefunden werden kann, bleibt s unverändert, und `om` wird zurückgegeben

itere vordefinierte Funktionen

	absoluter Betrag, Position eines Zeichens im ASCII-Code
	verwandelt ein beliebiges SETL/E-Objekt in eine Zeichenkette
r	bildet Zahlen in Zeichen ab, invers zu `abs`
an	rechte Variante zu `span`
y	rechte Variante zu `any`
eak	rechte Variante zu `break`
n	rechte Variante zu `len`
tch	rechte Variante zu `match`
tany	rechte Variante zu `notany`
d	rechte Variante zu `lpad`

Mengen

r ist nicht viel zu holen, denn fast alles ist in die Sprache eingebaut.

w	k npow A: die Menge aller k-elementigen Teilmengen der Menge A

Ein- und Ausgabe

r können von der Standard-Eingabe lesen und auf die Standard-Ausgabe schreiben; analog kann Dateien gelesen bzw. auf sie geschrieben werden. Im Gegensatz zu SETL erlauben wir hier ne binären Lese- oder Schreiboperationen; diese Operationen werden später im Zusammenhang dem *persistent store* allgemeiner verfügbar gemacht. Wir stellen die folgenden Ein- bzw. Ausbeoperationen zur Verfügung:

- zeichenorientierte Ein/Ausgabe: `fputcharf` und `putchar`, `fgetcharf` und `getchar` sowie `fungetcharf` und `ungetchar`
- formatierte Ein/Ausgabe: `fputf` und `put` bzw. `fgetf` und `get`

8.3.1 Dateien

Dateien sind konzeptionell extern gespeicherte Tupel; sie werden im Programm durch ihre Namen angesprochen, wobei die üblichen UNIX-Konventionen gelten. Die Namen sind Zeichenketten, die ggf. einen Pfad einschließen. Eine Datei wird mit **fopen** geöffnet; diese Prozedur verlangt zwei Parameter: den Namen der Datei, und den Modus, mit dem die Datei geöffnet wird. Dieser Modus orientiert sich an der Sprache C:

- "r" die Datei wird zum Lesen geöffnet: sie wird auf den Anfang positioniert
- "w" die Datei wird zum Schreiben geöffnet: der frühere Inhalt geht verloren
- "a" die Datei wird zum Schreiben geöffnet, es wird an das Ende der Datei angefügt, so daß der vorherige Inhalt erhalten bleibt.

Eine offene Datei muß mit **fclose** geschlossen werden; es gelten die üblichen Vorsichtsregeln: eine ungeöffnete Datei darf nicht geschlossen werden, eine Datei muß geschlossen oder noch nicht geöffnet sein, wenn sie geöffnet wird, auf eine ungeöffnete Datei darf nicht zugegriffen werden etc. Die Anzahl der gleichzeitig offenen Dateien ist endlich und system-abhängig. Hier greifen geeignete Ausnahmen. Implizit schließt der **stop**-Befehl alle noch offenen Dateien; ebenfalls werden beim Verlassen eines Programms alle noch offenen Dateien geschlossen, es kann aber hier zu undefinierten Situationen kommen.

8.3.2 Schreiben

Ist f eine zum Schreiben geöffnete Datei, so schreibt **fputcharf**(f, c) das Zeichen c auf f; c muß also eine Zeichenkette der Länge 1 sein. Analog schreibt **putchar**(c) das Zeichen c auf die Standard-Ausgabe.
fputf und **put** schreiben formatiert. Die Formatierung erfolgt wie in C durch eine Spezifikation, die in einer Zeichenkette, dem ersten Argument von **put**, zu finden ist. Für jedes zu druckende Objekt muß eine Spezifikation vorhanden sein, eine nicht zutreffende Spezifikation für x hat den gleichen Effekt, als ob **str** x mit der *default*-Spezifikation für Zeichenketten ausgedruckt werden würde. Enthält die Zeichenkette keine Spezifikation, so wird sie allein gedruckt, ggf. vorhandene weitere Argumente werden ignoriert. Analog wird bei **fputf** vorgegangen, wobei das erste Argument ein Datei-Name sein muß, dann folgt die Druckspezifikation, danach kommen ggf. weitere Argumente. Die Spezifikationen sind ebenfalls an C orientiert und sehen so aus:

- %fa: ein Atom wird rechtsbündig mit minimal f Stellen in der Darstellung gemäß 4.1.5 (p. 17) gedruckt; wird f nicht angegeben, so wird $f = 0$ angenommen.
- %fb: ein boolescher Wert wird rechtsbündig mit minimal f Stellen gedruckt, wobei der Wert **true** als #**true** und der Wert **false** als #**false** gedruckt wird; ist f nicht angegeben, so wird $f = 0$ angenommen.
- %fd: eine ganze Zahl mit minimal f Stellen einschließlich des Vorzeichens wird rechtsbündig gedruckt; wird f nicht angegeben, so wird $f = 0$ angenommen.
- %$f.g$f: eine reelle Zahl wird rechtsbündig mit minimal f Stellen einschließlich des Vorzeichens vor dem Dezimalpunkt, und minimal g Stellen nach dem Dezimalpunkt gedruckt; wird $f.g$ nicht angegeben, so wird $f.g = 0.0$ angenommen (beide sollten entweder spezifiziert oder nicht-spezifiziert werden).

- %p: eine vom Benutzer definierte Prozedur wird gedruckt. Dies geschieht, indem das zur Prozedur kanonisch gehörige `lambda` konstruiert und ausgedruckt wird.
- %fs: eine Zeichenkette wird rechtsbündig gedruckt, dazu werden f Stellen benötigt.
- %x: *default*-Darstellung; insbesondere wird der undefinierte Wert `om` als `#*` gedruckt.
- ein zusammengesetztes Objekt (Menge, Abbildung, Tupel) kann gedruckt werden, indem die Spezifikation für ein beliebiges Element in | eingeschlossen wird (also würde eine Menge ganzer Zahlen mittels | %6d | gedruckt werden können.
  Diese Art der Formatierung muß jedoch noch genauer betrachtet werden.

Falls statt %f %$-f$ angegeben wird, so erfolgt die entsprechende Ausgabe linksbündig.
Die Spezifikationen sind wie in C in einer Zeichenkette zusammengefaßt; die Zeichenkette wird gedruckt, und die darin eingeschlossenen Formatangaben werden interpretiert; hierbei haben '\n' (*newline*), '\t' (*tab*-Zeichen), '\b' (*backspace*), '\r' (*carriage return*), '\f' (*form feed*), '\\' (*Fluchtsymbol selbst*) und '\"' die aus C übernommenen Interpretationen. '\%' erlaubt es, das Prozentzeichen % zu drucken.
Beliebige Bit-Muster können mit der Angabe des üblichen Fluchtsymbols \, gefolgt von bis zu drei oktalen Ziffern, die den numerischen Wert des Bit-Musters bezeichnen, erzeugt werden. Dem Null-Zeichen '\0' dürfen dabei keine weiteren Ziffern folgen.
Die Formatierung kann typabhängig gemacht werden, indem man sagt: `put(`*format* x, x`)`, wobei *format* definiert ist durch

```
operator format(t);
use as prefix operator;
if t = om then
 return "%x";
else return
 case type t of
  (atom): "%a"
  (integer): "%d"
  (real): "%f"
  (string): "%s"
  (set, tuple, map): "|" + (+/[format y: y in t]) + "|"
  (optype, proctype): "%p"
  (boolean): "%b"
 end case;
end if;
end format;
```

8.3.3 Lesen

Im Aufruf `fgetf`$(f, \sigma, x_1, \ldots, x_n)$ ist f der Datei-Name, σ die Eingabespezifikation, und $x_1, \ldots, x_n$ sind die Argumente, die gelesen werden. Ist das Ende der Datei erreicht, so erhalten diese Argumente den Wert `om`. Dies ist der einzige vorgesehene Weg, das Ende einer Datei zu erkennen. Analog arbeitet `get`, hier wird als Eingabe allerdings die Standard-Eingabe genommen, so daß `get`$(x_1, \sigma, \ldots, x_n)$ versucht, n Werte zu lesen. `fgetcharf`(f, x) und `getchar`(x) lesen das nächste Zeichen von der Datei f bzw. von der Standard-Ausgabe. In allen diesen Fällen muß die Datei zum Lesen geöffnet sein. `fungetcharf` und `ungetchar` legen ihr Argument in die Eingabedatei bzw. die Standard-Eingabe zurück; pro Datei darf zu jedem Zeitpunkt höchstens ein Zeichen zurückgegeben werden (so daß also für eine spezifische Datei f zwei Aufrufe von `fungetcharf`$(f, \bullet)$ durch mindestens eine Leseoperation auf derselben Datei getrennt sein müssen).

Die Eingabespezifikation ist eine Zeichenkette und orientiert sich an der formatierten Eingabe in C sowie der formatierten Ausgabe gem. Abschnitt 8.3.2 (p. 35). Als *default*-Wert wird die Spezifikation %x angenommen, so daß Werte beliebiger Typen eingelesen werden können. Wird ein Objekt mit einer unpassenden Spezifikation eingelesen, so wird die Ausnahme *Read_Error* (vgl. Tabelle 10 (p. 39)) aktiviert. Im Gegensatz zu den Ausgabespezifikationen wird keine Stellenzahl angegeben (die besteimmt sich aus der externen Darstellung des Werts selbst). Verschiedene Werte werden in der Eingabe durch *white space* voneinander getrennt.

- %a: ein Atom in der Darstellung gemäß 4.1.5 (p. 17) wird gelesen.
- %b: ein boolescher Wert wird gelesen, wobei der Wert `true` als `#true` und der Wert `false` als `#false` in der Eingabe vorhanden sein muß.
- %d: eine ganze Zahl wird gelesen.
- %f: eine reelle Zahl wird gelesen.
- %p: der Text einer anonymen Prozedur wird gelesen.
- %x: *default*-Darstellung. Der undefinierte Wert `om` muß als #* in der Eingabe dargestellt sein, falls er eingelesen werden soll.
- %s: eine Zeichenkette wird eingelesen. Die Zeichenkette sollte weder *white space* noch ein eingebettetes " enthalten. Soll die Zeichenkette das Leerzeichen enthalten, so muß sie in " eingebettet sein (die " gehören nicht zur Zeichenkette); soll sie " enthalten, so muß den Gänsefüßchen ein \ als Fluchtsymbol vorangestellt werden.

Die Fehlerbehandlung bei der Eingabe wäre noch genauer zu betrachten.
Als Beispiel betrachten wir einen primitiven Interpretierer: wir lesen ein Tripel der Form *op* x y ein, wobei $x, y \in \mathcal{Z}$ und *op* einer der arithmetischen Operatoren $+, -, *$ ist. Ausgegeben wird das Resultat.

```
program Interpret;
constant head := "lambda (x, y): return x",
         tail := " y; end lambda",
         Datei := (str newat())(2 ..);
getchar(op); get("%d %d", eins, zwei);
 -- Konvertiere in ein λ
fopen(Datei, "w");
fputf(Datei, "%s", head + op + tail);
fclose(Datei);
fopen(Datei, "r");
fgetf(Datei, "%p", oops); -- oops ist jetzt ein λ
fclose(Datei);
system("rm " + Datei); -- Unix läßt grüßen
put("das war es: %d", oops(eins, zwei));
end Interpret;
```

9 Vordefiniertes

Einige Objekte sind bereits in der Umgebung des Programms definiert, ohne deswegen zum Sprachumfang zu gehören. Es handelt sich hierbei im wesentlichen um Typ-Konstanten, um die Booleschen Werte, und um Ausnahmen. Die Ausnahmen können neu definiert werden, die Typ-Konstanten jedoch nicht.

Name	bezeichneter Typ
`atom`	Atome
`boolean`	Boolesche Werte
`integer`	ganze Zahlen
`map`	Abbildung (ein- und mehrwertig)
`optype`	benutzer-definierter Operator
`proctype`	benutzer-definierte Prozedur
`real`	reelle Zahlen
`set`	Menge
`string`	Zeichenkette (auch einzelne Zeichen)
`tuple`	Tupel

Tabelle 9: Vordefinierte Typ(atome)

9.1 Boolesche Werte

`true` und `false` sind vordefiniert.

9.2 Typ-Konstanten

Der vordefinierte unäre Operator `type` nimmt eine beliebiges Objekt, das von `om` verschieden ist, und produziert seinen Typ. Hierbei können die in Tabelle 9 (p. 38) genannten und nicht weiter überraschenden Werte vorkommen. Diese Typ-Konstanten sind selbst wieder Atome.

9.3 Ausnahmen

Wir geben in Tabelle 10 (p. 39) einige Ausnahmen an, die in der System-Umgebung definiert sind. Lexikalisch sind sie an den beiden Unterstrichen zu erkennen. Falls eine Ausnahme ein Argument hat, ist es angegeben.

Name	Aktion
`Div_Zero`	Division durch Null
`Int_OverFlow`	betragmäßig zu große ganze Zahl
`Real_OverFlow`	betragmäßig zu große reelle Zahl
`Int_UnderFlow`	betragmäßig zu kleine ganze Zahl
`Invalid_Operands`	Nicht erlaubte Operanden
`Om_InSet`(*s*)	Versuch, `om` in die Menge *s* einzufügen
`Om_InMap`(*f*)	Versuch, `om` in eine Abbildung *f* einzufügen
`Op_NotAppl`(*file*)	Operation nicht auf die Argumente anwendbar
`Not_OpenRead`(*file*)	Datei *file* nicht zum Lesen geöffnet
`Not_OpenWrite`(*file*)	Datei *file* nicht zum Schreiben geöffnet
`File_Error`(*file*)	Dateifehler bei Datei *file* (Datei existiert nicht oder ist nicht zugreifbar)
`Read_Error`(*file*)	Lesefehler auf der Datei *file* (Formatangabe ist nicht zutreffend, vgl. 8.3.3 (p. 36))
`TooMany_Files`	Zu viele Dateien gleichzeitig geöffnet
`UnDef_Exception`(*ident*)	die Ausnahme *ident* wird nicht behandelt (vgl. 3.6.3 (p. 13))

Tabelle 10: Vordefinierte Ausnahmen

10 Reservierte Wörter

Index