W-Lisp Sprachbeschreibung

0. Einleitung

W-Lisp [Wippermann 91] ist eine Sprache, die im Bereich der Implementierung höherer Programmiersprachen verwendet wird. Ihre Anwendung ist nicht auf diesen Bereich beschränkt. Gute Lesbarkeit der W-Lisp-Notation wird durch zahlreiche Anleihen aus dem Bereich der bekannten imperativen Sprachen erzielt.

W-Lisp-Programme können im Rahmen eines Common Lisp-Systems ausgeführt werden. In der W-Lisp Notation können alle Lisp-Funktionen (inkl. MCS) verwendet werden, so daß die Mächtigkeit von Common-Lisp [Steele 90] in dieser Hinsicht auch in W-Lisp verfügbar ist.

1. Elemente zur Programmnotation

Die Elemente (Token), mit denen Programme notiert werden können, lassen sich in eine Anzahl von Klassen aufteilen. Die Beschreibung dieser Token durch reguläre Ausdrücke findet sich im Anhang B.

A) Namen (`ID`) dienen zur Bezeichnung von Objekten verschiedenster Art. Namen können gebildet werden aus Buchstabenzeichen, Ziffernzeichen und Zeichen aus {+ - * / % @ ^ & _}. Namen beginnen mit einem Buchstabenzeichen oder einem Zeichen aus {# & % @}. Bei der Transformation von W-Lisp nach Common Lisp wird ein einleitendes "#"-Zeichen eliminiert, allen mit anderen Zeichen beginnenden Namen wird ein "-"-Zeichen (Bindestrich) angehängt.

   Durch den angehängten Bindestrich werden ungewollte Kollisionen frei gewählter Namen mit Namen, die in Common Lisp eine spezielle Bedeutung haben (z. B. `first` als Funktionsname), verhindert.

```
Beispiele:
  mark   => mark-
  #first => first
  first  => first-
```

   Es wird empfohlen, Namen klein zu schreiben. Zwischen Groß- und Kleinschreibung wird nicht unterschieden.

   In Syntaxregeln werden Namen durch `ID` bezeichnet.

B) Zahlen (`NB`) notieren nichtnegative Zahlenwerte. Sie werden in der durch Common Lisp [Steele 90] definierten Weise gebildet.

```
Beispiele:
  4712
  008
```

C) Zeichen (`CHR`) werden durch Voransetzen eines "$"-Zeichens zu Zeichenwerten erklärt. Nicht abdruckbare Zeichen werden durch die (selbsterklärenden) Namen: space, newline, tab notiert.

```
Beispiele:
  $a      => #\a
  $space  => #\space
  $$      => #\$
```

D) Strings = Zeichenketten (`STRG`) werden in der durch Pascal definierten Weise gebildet. Strings dürfen nicht über das Zeilenende hinausgehen.

```
Beispiele:
  "String"
  ""
  """String"" im String"
  """"
```

E) Kennzeichen (`MARK`) entstehen aus Namen durch voranstellen eines ":"-Zeichens. Kennzeichen sind Literale (sie liefern sich selbst als Wert), können aber auch zur Bezeichnung von Funktionen verwendet werden.

```
Beispiele:
  :list  :test
```

F) Wortsymbole dienen zur Formulierung von Programmkonstrukten. Sie bestehen ausschließlich aus Großbuchstaben und besonderen Zeichen.

```
Beispiele:
  FCT    IF    ELSE    FI    END-FCT
```

G) Logische Werte werden durch die Wortsymbole `TRUE` und `FALSE` notiert. Statt `FALSE` kann auch `NIL` verwendet werden.

H) Trennzeichen sind `space`, `newline`, `tab`, "," und ";". Es kann das für die Lesbarkeit jeweils geeignetste Trennzeichen oder eine Folge von Trennzeichen verwendet werden. Wenn Anfang und Ende zweier aufeinanderfolgender Elemente (Token) nicht zu erkennen sind, ist ein Trennzeichen zu setzen. Es empfiehlt sich, in Ausdrücken grundsätzlich zwischen alle Operanden und Operationszeichen Trennzeichen einzufügen. Trennzeichen dürfen optional zwischen allen Token stehen mit folgender Ausnahmen:

Funktions- bzw. Methodenname und öffnende Argumentklammer einer Funktions- bzw. Methodenanwendung dürfen nicht getrennt werden.

Klassenname und öffnende Konstruktorklammer bei der Objektinstanziierung dürfen nicht getrennt werden.

```
Beispiele:
  a-1 + 1
  a - 1
  abs(x)
  abs (x)
  x <- a; y <- b
  x <- a y <- b
  Empty{}
  String{ repr = "123" }
```

Ein Preis für die freizügige Verwendbarkeit von Trennzeichen muß entrichtet werden: wo in klassischen Programmiersprachen (wie Pascal) Ausdrücke geschrieben werden können, dürfen in W-Lisp (mit wenigen Ausnahmen) nur Operanden stehen.

```
Beispiele:
  W-Lisp:                 Pascal:
  max((x + 1) y)          max(x+1, y)
  n <- (n + 1)            n := n+1
```

I) Sonderzeichen sind mit einer speziellen, eventuell von der Umgebung abhängigen Bedeutung versehen, die durch die Syntax der Sprachkonstrukte gegeben ist.

```
Beispiele:
  (  )  <-  <=  =
```

J) Kommentare sind bedeutungslos. Sie werden durch zwei aufeinanderfolgende Bindestriche eingeleitet und reichen bis zum Zeilenende.

```
Beispiele:
  -- dieser Ausdruck: x -- y ist Kommentar,
  -- in W-Lisp müßte er lauten: x - -y
  -- oder: x -(-y), besser noch: x + y!
```

2. Funktionsdefinition

Eine Funktionsdefinition wird durch folgende Regel syntaktisch beschrieben:

function-definition	=	**FCT** fct-id bound-vars [externals] **IS** clause-sequ **END-FCT**.
fct-id	=	ID | SP-ID | MARK.
bound-vars	=	"(" id-list ")".
externals	=	**EXT** id-list.
id-list	=	ID*.

`clause-sequ` siehe Abschnitt 5.

FCT, **IS** und **END-FCT** sind die Wortsymbole, die für eine Funktionsdefinition kennzeichnend sind. Die Nonterminale `fct-id`, `bound-vars` usw. sind durch die weiteren Regeln definiert.

Die Notation der Syntaxregeln entspricht der von Pascal her bekannten erweiterten Backus-Naur-Form (EBNF). Nonterminale werden durch kleingeschriebene Namen notiert, Terminale werden in Apostroph eingeschlossen oder durch großgeschriebene Namen notiert.

Durch eine Funktionsdefinition wird eine Funktion an einen Namen (`fct-id`) gebunden, unter dem sie zur Anwendung zur Verfügung steht. Eine Funktion besteht aus einer Folge von Klauseln (`clause-sequ`). Anwendung einer Funktion (auf Argumentwerte) bedeutet Ausführung dieser Klauselfolge, wobei zu den Klauseln die aktuellen Argumente unter den Namen der mit ihnen korrespondierenden (auf gleicher Position stehenden) formalen Parameter (`bound-vars`) verfügbar sind. Sollen mehrere Funktionen auf gemeinsame Objekte zugreifen können, müssen diese als externe Variable ausgewiesen werden. Treten in den Klauseln Assoziationen mit externen Variablen auf, sind die Namen dieser Variablen in der Liste der Externvariablen (`externals`) aufzuführen. Variable, die weder extern noch gebunden (formale Parameter) auftreten, sind lokal.

Die Ausführung einer Klauselfolge liefert einen Wert (siehe Abschnitt 5), der als Ergebniswert der Funktionsanwendung verfügbar ist. Außer einem Funktionsergebnis können durch Ausführung der enthaltenen Klauselfolge (Seiten-) Effekte auftreten (z. B. durch eine Assoziation mit einer externen Variablen).

```
Beispiel:
  FCT max(a b) IS
    IF a < b THEN
      b
    ELSE
      a   FI
  END-FCT
```

3. Klassendefinition

class-definition	=	base-class-definition | mixin-class-definition.
base-class-definition	=	**CLASS** ID [**INHERIT** ID] [**MIXINS** id-list] [externals] **IS** attrib* **END-CLASS**.
mixin-class-definition	=	**MIXIN** ID [**MIXINS** id-list] [externals] **IS** attrib* **END-CLASS**.

attrib = ID ["=" opd].

Eine Klasse definiert die Struktur ihrer Instanzen, d. h. sie definiert, welche Attribute jede Instanz enthält, und wie die Attribute bei der Instanziierung initialisiert werden. Die Struktur und das Verhalten kann durch Ableiten von Subklassen (Unterklassen) spezialisiert werden.

Eine Klasse ist eindeutig gekennzeichnet durch ihren Namen (`ID`). In einer Klassendefinition werden die Namen ihrer Superklassen (Oberklassen, `id-list`) und eine Liste von Attributen (`attrib`) angegeben. Bei der Definition einer Klasse müssen alle ihre Superklassen bereits definiert sein.

Basisklassen, gekennzeichnet durch das Schlüsselwort **CLASS**, repräsentieren (im Unterschied zu Mixin-Klassen) die essentiellen Eigenschaften ihrer Objekte. Mixin-Klassen, gekennzeichnet durch das Schlüsselwort **MIXIN**, enthalten zusätzliche Eigenschaften von Objekten, die sich meistens auf einen einzigen speziellen Aspekt beziehen. Mixin-Klassen können nicht instanziiert werden, d. h. es können nur Objekte von Basisklassen gebildet werden.

Eine Basisklasse kann höchstens eine andere Basisklasse spezialisieren, d. h. von ihr erben. Sie kann jedoch beliebig viele Mixinklassen beerben. Falls eine Basisklasse beerbt werden soll, wird sie nach dem Schlüsselwort **INHERIT** angegeben; Mixin-Klassen werden nach dem Schlüsselwort **MIXINS** notiert. Mixin-Klassen können nur von anderen Mixin-Klassen erben. Werden mehrere Klassen beerbt, so dürfen diese noch nicht in einer Beziehung zueinander stehen, d. h. es darf keine Klasse Superklasse einer anderen sein und es darf keine gemeinsame Superklasse existieren.

Treten bei mehreren Superklassen gleichnamige Attribute auf, so wird das Attribut der zuletzt notierten (Mixin-) Klasse geerbt.

Beispiel:

```
MIXIN M IS a END-CLASS

CLASS C IS a, b END-CLASS

CLASS S INHERIT C MIXINS M IS
  -- es wird das Attribut a der Mixin-Klasse M und
  -- das Attribut b der BasisKlasse C ererbt
END-CLASS
```

Attribute können beim Instanziieren von Objekten mit Default-Werten belegt werden. Diese werden als Operand (opd) hinter dem jeweiligen Attribut in der Klassendefinition angegeben. Zur Berechnung dieser Werte können globale Bezeichner benutzt werden; diese müssen dann in der Extern-Liste (`external`) aufgeführt werden. Bei gleichnamigen Attributen aus mehreren Superklassen wird der Initialisierungsausdruck der zuletzt notierten (Mixin-) Klasse geerbt.

Wird bei der Instanziierung (siehe Abschnitt 8) ein Initialisierungsausdruck angegeben, so wird dessen Wert für die Initialisierung verwendet, sonst der Wert des Ausdrucks in der Klassendefinition. Fehlen beide, so bleibt der Wert des Attributs undefiniert.

Beispiel:

```
CLASS circle IS
  x-offset = 0          -- Sofern nichts anderes
  y-offset = 0          -- spezifiziert wird.
  radius                -- Bleibt undefiniert, bis
                        -- eine Zuweisung erfolgt.
END-CLASS

CLASS colorcircle INHERIT circle IS
  color
END-CLASS
```

```
CLASS vector IS
  x, y
END-CLASS
```

Für jedes Attribut werden automatisch Zugriffsmethoden (Accessor-Methoden) zum Lesen und Schreiben definiert. Der Name der Zugriffsmethode entspricht dem Attributnamen.

Klassen können nicht redefiniert werden. Existiert bereits eine Klasse gleichen Namens, erfolgt eine Warnung und die neue Definition wird ignoriert.

4. Methodendefinition

Eine Methodendefinition wird durch folgende Regel syntaktisch beschrieben:

method-definition	=	**METHOD** method-id "(" param+ ")" [externals] **IS** clause-sequ **END-METHOD**.
method-id	=	ID | SP-ID | MARK.
param	=	ID [**OF** ID].

Durch eine Methodendefinition wird eine Funktion an einen Namen (`ID`) gebunden, unter dem sie zur Anwendung zur Verfügung steht. Eine Methode besteht aus einer Folge von Klauseln (`clause-sequ`). Anwendung einer Methode (auf Argumentwerte) bedeutet Ausführung dieser Klauselfolge, wobei zu den Klauseln die aktuellen Argumente unter den Namen der mit ihnen korrespondierenden (auf gleicher Position stehenden) formalen Parameter (`params`) verfügbar sind. Bei der Definition von Methoden muß mindestens ein Parameter angegeben werden. Sollen mehrere Methoden auf gemeinsame Objekte zugreifen können, müssen letztere als externe Variable ausgewiesen werden. Treten in den Klauseln Assoziationen mit externen Variablen auf, sind die Namen dieser Variablen in der Liste der Externvariablen (`externals`) aufzuführen. Variable, die weder extern noch gebunden (formale Parameter) auftreten, sind lokal.

```
Beispiel:
  METHOD area ( obj OF circle ) EXT PI IS
    (obj.radius * obj.radius * PI)
  END-METHOD
```

4.1 Multimethoden

Eine Methode ist eindeutig bestimmt durch ihren Namen und die Klassen ihrer Argumente. In W-Lisp werden Multimethoden unterstützt, d. h. eine Methode kann auf mehrere Klassen spezifiziert werden. Sie kann nur dann aktiviert werden, wenn alle Argumente von den korrespondierenden formalen Parametern akzeptiert werden. Ein Argument wird von dem zugehörigen formalen Parameter akzeptiert, wenn es von der spezifizierten Klasse (oder einer Subklasse) ist. Nicht spezifizierte Parameter, d. h. Parameter ohne nachfolgende Klassenangabe, akzeptieren Argumente jeder Klasse.

```
Beispiel:
  METHOD move ( obj OF circle; offset OF vector ) IS
    obj.x-offset <- (obj.x-offset + offset.x);
    obj.y-offset <- (obj.y-offset + offset.y);
  END-METHOD
```

4.2 Methodenauswahl

Eine Methode ist Kandidat für die Aktivierung, wenn alle Argumente mit den formalen Parametern korrespondieren. Ein Argument korrespondiert mit einem formalen Parameter, wenn es der gleichen Klasse oder einer Subklasse angehört, mit der der formale Parameter spezifiziert ist und an gleicher

Stelle steht.

Existieren mehrere Kandidaten für die Aktivierung, wird der speziellste Kandidat verwendet. Ein Kandidat ist spezieller als ein zweiter, wenn die Spezifikationen der ersten i-1 Parameter übereinstimmen und der i-te Parameter des ersten Kandidaten mit einer (echten) Subklasse des i-ten Parameters des zweiten Kandidaten spezifiziert ist.

Eine Methode kann die nächst allgemeinere Methode durch einen Aufruf der CLOS-Funktion `call-next-method` aktivieren. Die nächst allgemeinere Methode wird dann mit exakt den gleichen Parametern aufgerufen.

```
Beispiel:
  METHOD move ( obj OF colorcircle; offset OF vector ) IS
    -- Body
  END-METHOD

  move( colorcircle{radius=3; color=blue}, vector{x=3; y=5} );
```

Bei diesem Aufruf sind die Methoden aus Abschnitt 4.1 und Abschnitt 4.2 Kandidaten für die Aktivierung. Die Methode aus Abschnitt 4.2 ist jedoch die speziellere und wird deshalb aktiviert.

Innerhalb einer Methode sind keine Zuweisungen an die Parameter erlaubt.

5. Klauseln

Klauseln (`clause`) treten in mehreren Konstrukten von W-Lisp als Klauselfolgen (`clause-sequ`) auf:

clause-sequ = clause* [**RESULT** clause*].

Klauseln sind Assoziationen (`assoc`) oder Werte-Teile, wozu Schleifen (`loop`) und Operanden (`opd`) zählen. In Operanden sind Bedingungen (`cond`) enthalten, daher werden diese hier mit aufgeführt:

clause = assoc | loop | opd.
assoc = dest-part "<-" opd
| opd "->" dest-part.
dest-part = ID | list-descr | obj-descr.

`list-descr` siehe Abschnitt 7.

`opd` siehe Abschnitt 6.

loop = **WHILE** gcd **OD**.
cond = **IF** gcd (**ELSIF** gcd)*
[**ELSE** clause-sequ] **FI**.
gcd = expr (**THEN** | **DO**) clause-sequ.

`expr` siehe Abschnitt 6.

Die Klauseln einer Klauselfolge (`clause-sequ`) werden sequentiell ausgeführt. Der Wert einer Klauselfolge ist durch die letzte Klausel, oder - falls vorhanden - durch die auf **RESULT** folgende Klausel bestimmt.

Durch eine Assoziation (`assoc`) werden im Zielteil (`dest-part`) stehende Variable mit Werten assoziiert, die sich bei Ausführung des Operanden (`opd`) ergeben. Der Wert des Operanden ist auch der Wert der Assoziation. Ein außerhalb eines Zielteils auftretender Variablenname steht für den mit diesem Namen assoziierten Wert.

In Bedingungen und Schleifen treten "bewachte" Klauselfolgen (`gcd`) auf. Dieses Konzept stammt von Dijkstra [Dijkstra 75]. Dijkstra bezeichnet Folgen von Kommandos (commands), denen eine

Bedingung als "Wächter" (guard) vorangestellt ist, als **g**uarded **c**omman**d**s (= `gcd`). Die Ausführung einer `gcd` beginnt mit der Ausführung der Bedingung (`expr`). Der als Bedingung stehende Ausdruck liefert `TRUE` oder `FALSE` ≡ `NIL`, wobei jeder Wert ≠ `NIL` als `TRUE` gilt. Für `NIL` wird die Ausführung der `gcd` abgebrochen, sonst wird die anschließende Klauselfolge ausgeführt. Der Wert der ausgeführten Klauselfolge wird zum Wert der `gcd`. In diesem Fall ist die Ausführung erfolgreich.

Die `gcd`s einer Bedingung (`cond`) werden sequentiell ausgeführt, solange, bis eine `gcd` erfolgreich ausgeführt wurde. Ihr Wert ist der Wert der Bedingung. Falls keine `gcd` erfolgreich ausgeführt wurde, ist der Wert der Bedingung durch die hinter **ELSE** stehende Klauselfolge oder - falls diese nicht vorhanden ist - durch `NIL` gegeben.

Die `gcd` einer Schleife (`loop`) wird wiederholt ausgeführt. Ist eine Ausführung nicht erfolgreich, ist die Ausführung der Schleife beendet. Ihr Wert ist dann durch den Wert der letztmalig mit Erfolg ausgeführten `gcd` gegeben. Falls keine erfolgreiche Ausführung der `gcd` zustandekommt (abweisender Schleifencharakter !) ist der Wert der Schleife `NIL`.

```
Beispiele:
  res <-  IF a < b THEN
            b
          ELSE
            a  FI

  old-list -> <first ..old-list>

  WHILE first DO
    RESULT
    new-list <- {first ..new-list}
    old-list -> <first ..old-list> OD

  -- Die Bedeutung wird in Abschnitt 7 erklärt.
```

6. Ausdrücke und Funktionsanwendungen

Die Syntax von Ausdrücken und Funktionsanwendungen stimmt weitgehend überein mit den Regelungen in den verbreiteten imperativen Sprachen. Durch die Regeln für Konjunktion (`conj`), Relation (`relation`), Summe (`sum`), Term (`term`) und Primärausdruck (`primary`) ist die Priorität der Operatoren geregelt. Boole'sche Operatoren (**AND**, **OR**) haben geringere Priorität als arithmetische und vergleichende. Für sie gilt verkürzte Auswertung. Alle Operatoren sind linksassoziativ.

expr	=	conj (**OR** conj)*.
conj	=	relation (**AND** relation)*.
relation	=	sum [relop sum].
relop	=	"=" | "/=" | "<" | "<=" | ">" | ">=" | **IN**.
sum	=	term (addop term)*.
addop	=	"+" | "-".
term	=	primary (mulop primary)*.
mulop	=	"*" | "/".
primary	=	[monop] opd.
monop	=	"+" | "-" | **NOT**.
opd	=	fct-or-method-appl | ID [index] | obj-descr [index]
		| cond | list-constr | obj-constr | "(" expr ")" | lambda | literal | lisp-form.
fct-or-method-appl	=	SP-ID "(" opd* ")".

Funktions- bzw. Methodenname und öffnende Argumentklammer einer Funktions- bzw. Methodenanwendung dürfen nicht getrennt werden. Eine Funktionsanwendung (`fct-appl`)

bedeutet die Anwendung der mit dem Namen assoziierten Funktion auf die Werte der (in Argumentklammer) angegebenen Operanden. Die Operanden werden sequentiell ausgewertet.

index = "[" opd* [".."] "]".

Die Auswertung einer Variablen, eines Objektzugriffs und einer Funktionsanwendung kann eine Liste als Ergebnis liefern, die indiziert werden kann. Das erste Listenelement besitzt den Index 0. Durch Angabe mehrerer Indizes können ineinander verschachtelte Listen indiziert werden. Folgt auf den letzten Index der Operator "..", so bildet die Teilliste, die mit dem indizierten Element beginnt, den Wert der Indizierung.

`cond` siehe Abschnitt 5.

`list-constr` siehe Abschnitt 7.

lambda = "\" ID* "." opd.

Ein Lambda-Ausdruck (`lambda`) bezeichnet eine unbenannte Funktion. Die Bezeichner (`ID*`) sind im Ausdruck gebunden und werden bei Aktivierung der Funktion an Werte gebunden, für alle anderen Bezeichner gilt lexikalische Bindung.

literal = MARK | STRG | NB | CHAR | boolean | quoted-id.
boolean = **TRUE** | **FALSE** | **NIL**.
quoted-id = "'" ID.
lisp-form = "|" CommonLisp-Form "|".
In W-Lisp - Ausdrücken können auch Formen aus Common-Lisp verwendet werden. Zur Definition von Common-Lisp - Formen siehe [Steele 90].

Arithmetische, logische und vergleichende Operatoren haben die in der Mathematik gewohnte Bedeutung.

Der Operator **IN** prüft das Vorhandensein eines Elementes in einer Liste (durch die u. a. eine Menge dargestellt werden kann). Die Vergleichsoperatoren "=" und "/=" können auf Operanden (`sum`) beliebiger Art, die restlichen Vergleichsoperatoren nur auf arithmetische Operanden angewendet werden.

Wird einem Namen ein Apostroph vorangesetzt (`quoted-id`) bezeichnet er sich selbst als Wert.

```
Beispiele:
  a OR b AND NOT c
  a + 1 < b OR c
  a IN old-list
  name = 'alpha
  max((x + 1) y)
```

7. Listenkonstruktoren und -deskriptoren

Listen stellen die mächtigste und universell verwendbarste Datenstruktur dar. Fast alle komplexeren Datenstrukturen, die in der Informatik verwendet werden, lassen sich durch Listen darstellen und in durchweg sehr einfacher Weise handhaben. In W-Lisp sind zwei - durch große Anschaulichkeit ausgezeichnete - Konstrukte für die Listenmanipulation vorhanden:

list-constr = "{" ([".."] opd)* "}".
list-descr = "<" descr* [descr-tail] ">".
descr = ID | obj-descr | MARK | list-descr.
descr-tail = ".." [ID | obj-descr].

Durch einen Listenkonstruktor (`list-constr`) kann eine Folge von Elementen zu einer Liste zusammengefaßt werden. Die Elemente ergeben sich durch Auswertung der Operanden (`opd`) in sequentieller Ordnung. Der ".."-Operator überführt (wenn er in einem Listenkonstruktor steht) eine

Liste in eine Folge. Der Operand, auf den der ".."-Operator angewendet wird, muß bei seiner Auswertung also eine Liste ergeben.

```
Beispiele:
  e1 <- {1 2 3}            -- Wert von e1 = (1 2 3)
  e2 <- {..e1 4 5}         -- Wert von e2 = (1 2 3 4 5)
  e3 <- {e1 {4 5}}         -- Wert von e3 = ((1 2 3) (4 5))
```

Der Listendeskriptor (`list-descr`) kann nur als Zielteil (`dest-part`) einer Assoziation auftreten. Die Auswertung des zugehörigen Werteteils muß eine Liste ergeben. Die Elemente dieser Liste werden sequentiell mit den im Listendeskriptor aufgeführten Deskriptoren (`descr`) assoziiert. Enthält die Liste weniger Elemente als Deskriptoren vorliegen, werden die überzähligen Deskriptoren mit `NIL` assoziiert. Mit Kennzeichen (`MARK`) kann nichts assoziiert werden, sie dienen als Platzhalter im Deskriptor, deren korrespondierendes Listenelement nicht interessiert. Tritt ein Listendeskriptor selbst als Deskriptor auf, muß mit ihm eine Liste als Listenelement korrespondieren. Der hier beschriebene Assoziationsprozeß vollzieht sich erneut auf einer tieferen Ebene. Als Deskriptorende (`descr-tail`) kann ein Name angegeben werden, der einem ".."-Zeichen folgt. Der ".."-Operator besagt hier, daß alle nicht mit Deskriptoren korrespondierenden Elemente zu einer Liste zusammengefaßt und mit dem angegebenen Namen assoziiert werden. Fehlt ein Name nach dem ".."-Zeichen, ist dieses wirkungslos.

```
Beispiele (mit den Werten, die mit e1 usw. in obigen Beispielen
assoziiert wurden):
  e1 -> <a b c>          -- Wert von a = 1, b = 2, c = 3
                         -- Wert der Assoziation = (1 2 3)
  e1 -> <a ..tail>       -- Wert von a = 1, tail = (2 3)
                         -- Wert der Assoziation = (1 2 3)
  tail -> <:ignore x y>  -- Wert von x = 3, y = nil
                         -- Wert der Assoziation = (2 3)
  e3 -> <:ignore <y ..>> -- Wert von y = 4
                         -- Wert der Assoziation = ((1 2 3)(4 5))
```

8. Objektkonstruktoren und -deskriptoren

obj-constr	=	OBJ-CONSTR "{" init* "}".
init	=	ID "=" opd.
obj-descr	=	ID ("." ID)+.

Jedes Objekt in W-Lisp ist Instanz einer bestimmten Klasse, die im Objektkonstruktor durch ihren Namen (`ID`) angegeben wird.

Objekte werden instanziiert durch einen Objektkonstruktor (obj-constr). Dabei können Attribute des Objekts mit Initialisierungswerten belegt werden. Eventuell vorhandene Defaultwerte aus der Klassendefinition (siehe Abschnitt 3) werden dabei übergangen.

```
Beispiel:
  obj1 <- circle{ radius = 3 }
  obj2 <- circle{ radius = 3, x-offset = 4, y-offset = 5 }
```

Klassenname und öffnende Konstruktorklammer bei der Objektinstanziierung dürfen nicht getrennt werden.

Entsprechend der Klassendefinition aus Abschnitt 3 ergibt das Beispiel im Common Lisp - System die folgenden Objekte:

```
obj1, an instance of #<Base-Class CIRCLE->, has slot values:
  X-OFFSET-: 0
  Y-OFFSET-: 0
  RADIUS-: 3

obj2, an instance of #<Base-Class CIRCLE->, has slot values:
  X-OFFSET-: 4
  Y-OFFSET-: 5
  RADIUS-: 3
```

Wenn der Klassenname erst zur Laufzeit des Programmes bestimmt werden kann, können Objekte auch mit der CLOS-Funktion `make-instance` erzeugt werden. Der erste Parameter der Funktion bezeichnet den Klassennamen, die nachfolgenden Parameter werden immer paarweise als Attributname und Attributwert aufgefaßt. Die obige Instanziierung kann damit wie folgt aussehen:

```
Beispiel:
  class-name <- 'circle
  obj2 <- #make-instance(  class-name
                           'radius 3 'x-offset 4 'y-offset 5 )
```

Der lesende und schreibende Zugriff auf Attribute von Objekten erfolgt mittels Objektdeskriptor (`obj-descr`). Der erste Bezeichner ist der Name des Objektes, der Bezeichner nach dem Punkt ist der Name des Attributs, auf das zugegriffen werden soll. Wenn das Attribut ein weiteres Objekt bezeichnet, kann dort wiederum ein Attribut angegeben werden usw. Auf Attribute von Objekten kann mittels Objektdeskriptor oder mittels Zugriffsmethode (siehe Abschnitt 3) lesend zugegriffen werden. Die folgenden Beispiele demonstrieren dies:

```
Beispiel:
  obj1.radius <- 17

  #print( obj1.radius )    -- ist gleichbedeutend mit dem
  #print( radius(obj1) )   -- Zugriff mittels Zugriffsmethode
```

Externe Variablen aus Common Lisp können nicht korrekt als Objektdeskriptoren interpretiert werden, da in Common Lisp ein Punkt (".") in Variablennamen erlaubt ist, ohne daß er eine besondere Bedeutung trägt.

9. Ausführung von W-Lisp-Programmen und Klauseln

Ein W-Lisp-Programm besteht aus einer Folge von Funktions-, Klassen- und Methodendefinitionen:

W-Lisp-Progr = { function-definition | class-definition | method-definition
| CommonLisp-Form }.

W-Lisp-Programme werden in Dateien abgelegt. Diese werden beim Laden oder mit Hilfe spezieller Übersetzungsfunktionen in äquivalente Common Lisp-Programme übersetzt. W-Lisp-Klauseln können als äquivalente Common Lisp-Formen interpretiert werden.

Die folgenden Unterkapitel (9.1 bis 9.3) beschreiben die Benutzerschnittstellen und sind daher für den weniger anwendungsorientierten Leser von geringem Interesse.

9.1 Common Lisp-Modus

Nach dem Aufruf von W-Lisp gelangt man in den **Common Lisp-Modus**, worin die Interpretation von Common Lisp-Formen erfolgt. Die auf die Eingabeanforderung (>) angegebene Form wird ausgeführt und der Ergebniswert zurückgeliefert. Um W-Lisp-Definitionen im Common Lisp-Modus auszuführen, müssen diese durch Voranstellen der Zeichenfolge "#!" kenntlich gemacht werden. Ihrer

Ausführung geht die Übersetzung in eine äquivalente (bedeutungsgleiche) Common Lisp-Form voraus.

9.2 W-Lisp-Modus

Der Wechsel vom Common Lisp-Modus in den **W-Lisp-Modus** erfolgt durch Eingabe von "~", in die andere Richtung durch ">q". Eine Beschreibung weiterer Systemkommandos wird durch ">h" angezeigt.

Im W-Lisp-Modus werden Definitionen und Klauseln in W-Lisp-Notation akzeptiert. Für die Ausführung werden sie nach Common Lisp übersetzt. Das Ergebnis der Auswertung wird in der Common Lisp-Form angezeigt.

Die Eingabeaufforderung im W-Lisp-Modus ist "W-Lisp:"

Klauseln müssen vollständig in einer Zeile stehen, während Definitionen mehrere Zeilen lang sein dürfen. Der Text, der auf eine Klausel oder Definition folgt, wird bis zum nächsten Zeilenende ignoriert. Eine Klausel darf nicht über das Zeilenende hinausgehen. Es wird also jeweils nur eine Klausel bzw. eine Definition aus der Eingabe ausgeführt. Die Eingabe

```
W-Lisp: 3 3 + 4
```

liefert deswegen nur das Ergebnis 3 zurück und ignoriert den Ausdruck 3 + 4.

9.3 W-Lisp-Programme

W-Lisp-Programme werden in äquivalente Common Lisp-Programme übersetzt. W-Lisp-Programme können als zusammenhängende Einheit Bestandteil einer Common Lisp-Datei sein. Sie werden dazu zwischen die Zeichen "~" und ">q" gesetzt (">q" unmittelbar vor Dateiende kann entfallen).

Einzelne W-Lisp-Definitionen können in Common Lisp-Dateien eingebettet werden, indem ihnen die Zeichenfolge "#!" vorangestellt wird.

Für Dateien, in denen W-Lisp-Programme oder -Definitionen enthalten sind, stehen im W-Lisp-Modus mehrere Funktionen zum Laden und Übersetzen zur Verfügung. Im Common Lisp-Modus können diese Funktionen durch Lisp-Formen ausgeführt werden, in denen der Name jeweils um "-" ergänzt wird.

W-LISP:LOAD(*file*) lädt die angegebene Datei *file*, wobei alle W-Lisp-Definitionen in äquivalente Common Lisp-Definitionen übersetzt werden.

W-LISP:VLOAD(*file*) lädt wie W-LISP:LOAD und zeigt die geladenen Definitionen an.

W-LISP:SC(*file*) erstellt eine neue Datei *file*.code, in die Common Lisp-Elemente unverändert, W-Lisp-Definitionen in ihrer Übersetzung nach Common Lisp übernommen werden. Mit dieser Funktion kann die wiederholte Transformation bei jedem Ladevorgang auf eine einzige Transformation reduziert werden.

W-LISP:COMPILE-FILE(*file*) übersetzt die Definitionen der angegebenen Datei in eine Binär- (Objekt-) Datei *file*.o, wobei zuvor alle W-Lisp-Definitionen nach Common Lisp übersetzt werden. Bei diesem Übersetzungsvorgang wird ebenfalls die Zwischencode-Datei *file*.code erzeugt.

Die Syntax der Programm-Datei ist unabhängig vom jeweils eingestellten Modus, d. h. daß auch im W-Lisp-Modus W-Lisp-Definitionen gekennzeichnet werden müssen (mit "~" bzw. "#!"), wenn sie von einer Datei geladen werden.

Dateinamenskonvention: Bei Angabe eines Dateinamens *filename* sucht das System zunächst nach einer Datei namens *filename*. Falls diese Datei nicht existiert, wird nach *filename*.lsp gesucht. Bei einem Aufruf einer Übersetzung werden Dateien mit Namen *filename*.code und *filename*.o erzeugt. Falls *filename* bereits den Suffix .lsp enthält, wird dieser **nicht** durch .o bzw. .code ersetzt.

Beispiel einer W-Lisp-Sitzung:

```
> ~
W-Lisp: #use-package('#w-lisp)
W-Lisp: vload("sample")    -- Suffix .lsp kann entfallen
W-Lisp: fac1-
W-Lisp: Ciao
Finished loading sample.lsp
NIL
W-Lisp: fct fac(x) is
          if x < 2 then 1 else (x * fac((x - 1))) fi end-fct
FAC-
W-Lisp: x <- 6
6
W-Lisp: fac(x)
720
W-Lisp: >q
Ciao
>
```

Anhang A: Literatur

[Bretthauer, Kopp] H. Bretthauer, J. Kopp: The Meta Class System MCS: A Portable Object system for Common Lisp, Dokumentation zur Version 1.3, 6/1991

[Dijkstra 75] E.W. Dijkstra: *Guarded Commands, Nondeterminacy and Formal Derivation of Programms*, Comm. ACM, 11, 3 pp. 147-149

[Eppler 91] R. Eppler: *Entwurf und Implementierung einer Semantik-Beschreibungssprache und deren Anwendung auf Occam2*, Diplomarbeit, Universität Kaiserslautern, FB Informatik, 1991.

[Jensen 85] K. Jensen, N. Wirth: *Pascal - User Manual and Report*, Springer 1985

[Knauber, Vorwieger] P. Knauber, S. Vorwieger: *Terminologie des Übersetzerbaus*, Interner Bericht FB Informatik 1994, Universität Kaiserslautern, Nr. 255/94

[Steele 90] Guy L. Steele jr.: *Common Lisp - The Language*, 2nd. edition. Digital-Press 1990.

[Wippermann 91] W. Wippermann: *W-LISP - Definition und Implementierung*, Interner Bericht FB Informatik 1991, Universität Kaiserslautern, Nr. 210/91

Anhang B: Beschreibung der Token[1]

```
                        Zeichenklassen:
letter          =  {"A".."Z" "a".."z"}.
digit           =  "0".."9".
non-number      =  letter | "*" | "/" | "%" | "@" | "$" | "^" | "&" | "_" | "=" | "<" | ">" | "~".
escape-char     =  "(" | ")" | "[" | "]" | "{" | "}" | "!" | "?" | '"' | "'" | "#" | ":".

                        Common Lisp-Symbole dürfen nicht mit "+", "-" oder "." beginnen
                        (wg. integer) und nicht mit ">" enden (wg. list-descr):
symbol-start    =  letter | digit | "*" | "/" | "%" | "@" | "^" | "&" | "_" | "=" | "~"
                |  "\" escape-char | "\\" | "\|" .
symbol-rest     =  symbol-start | "+" | "-" | "." | "," | ";" | "$" | "<" | ">".
symbol-end      =  symbol-start | "+" | "-" | "." | "," | "$" | "<".

                        Zahlen in Common Lisp:
radix           =  digit digit?.                    # 2 <= radix <= 36
binary-digit    =  "0" | "1".
octal-digit     =  "0" .. "7".
hex-digit       =  {"0" .. "9" "A" .. "F"}.
extended-digit  =  {"0" .. "9" "A" .. "Z"}.
integer         =  ( ("+" | "-")? digit+ "."? )
                |  "#b" ("+" | "-")? binary-digit+
                |  "#o" ("+" | "-")? octal-digit+
                |  "#x" ("+" | "-")? hex-digit+
                |  "#" radix ("r" | "R") ("+" | "-")? extended-digit+.
ratio           =  ("+" | "-")? digit+ "/" digit+.
exponent-marker =  "e" | "s" | "d" | "f" | "l" | "E" | "S" | "D" | "F" | "L".
exponent        =  exponent-marker ("+" | "-")? digit+.
floating-point-number = ("+" | "-")? digit* "." digit+ exponent?
                |  ("+" | "-")? digit+ ("." digit*)? exponent.
NB              =  integer | ratio | floating-point-number.

                        Zeichenklassen für Bezeichner:
wlisp-id-start  =  letter | "%" | "&" | "@".
wlisp-id-rest   =  letter | digit | "+" | "-" | "*" | "/" | "%" | "@" | "^" | "&" | "_".

                        Common Lisp-Bezeichner dürfen nicht mit ">" enden (wg. list-descr):
lisp-id-start   =  letter | digit | "*" | "/" | "%" | "@" | "$" | "^" | "&" | "_" | "=" | "~".
lisp-id-rest    =  lisp-id-start | "+" | "-" | "<" | ">".
lisp-id-end     =  lisp-id-start | "+" | "-" | "<".

                        Marken müssen vor Common Lisp-Bezeichnern definiert werden.
MARK            =  ":" wlisp-id-rest+.
sp-mark         =  ":" wlisp-id-rest+ "(".

                        Zeichen müssen vor Common Lisp-Bezeichnern definiert werden.
CHAR            =  "$" ("space" | "newline" | "tab"
                                | {"A".."Z" "a".."z" "$" "%" "-" "+" "*" "/" }).
```

1. Die exakte Beschreibung der verwendeten Konstrukte befindet sich in [Knauber, Vorwieger].

W-Lisp-Bezeichner müssen vor Common Lisp-Bezeichnern definiert werden

```
wlisp-id          = wlisp-id-start wlisp-id-rest*.
sp-wlisp-id       = wlisp-id-start wlisp-id-rest* "(".

lisp-id           = ( ":" ( lisp-id-rest* lisp-id-end )? )
                  | lisp-id-start ( lisp-id-rest* lisp-id-end )?
                                  (":" ( lisp-id-rest* lisp-id-end )? )?
                  | lisp-id-start ( lisp-id-rest* lisp-id-end )?
                                  ("::" ( lisp-id-rest* lisp-id-end )? )?
                  | ( "+" | "-" ) (non-number ( lisp-id-rest* lisp-id-end )? )
                                  (":" ( lisp-id-rest* lisp-id-end )? )?
                  | ( "+" | "-" ) (non-number ( lisp-id-rest* lisp-id-end )? )
                                  ("::" ( lisp-id-rest* lisp-id-end )? )?
                  | ( "+" | "-" ) (":" (lisp-id-rest* lisp-id-end )? )
                  | ( "+" | "-" ) ("::" (lisp-id-rest* lisp-id-end )? ).
```

Objekt-Deskriptoren müssen vor Common Lisp-Symbolen definiert werden.

```
OBJ-DESCR         = (wlisp-id | common-wlisp-id) ("." (wlisp-id | common-wlisp-id) )+.
lisp-symbol       = symbol-start (symbol-rest* symbol-end)?
                                 ( "|" (symbol-rest | escape-char)* "|" )?
                                 (symbol-rest* symbol-end)?
                  | "|" (symbol-rest | escape-char)* "|" symbol-rest* symbol-end)? .
common-wlisp-id   = "#" ( lisp-id | lisp-symbol) .
sp-common-wlisp-id = "#" ( lisp-id | lisp-symbol) "(" .
```

Es folgen 3 Parser-Regeln, die aber auch zu den Terminalen zu rechnen sind.

```
COMMON-LISP-ID  = lisp-id | wlisp-id | lisp-symbol.
ID              = wlisp-id | common-wlisp-id.
SP-ID           = sp-slisp-id | sp-common-wlisp-id | sp-mark.

OBJ-CONSTR      = (wlisp-id | common-wlisp-id) "{".
```

Strings in W-Lisp und Common Lisp; zuerst der gemeinsame Teil:

```
lisp-string       = "" ( ANY - {"" '\'' """"} )* "".
wlisp-string-escape = """" ~> '\'".
wlisp-string      = "" ( wlisp-string-escape | ANY - {""} )* "" .
common-lisp-string = "" ( '\'' | ANY - {""} )* "" .
STRG              = lisp-string | wlisp-string .
COMMON-LISP-STRG  = lisp-string | common-lisp-string .
```

Kommentare in W-Lisp und Common Lisp; Common Lisp-Kommentare müssen mit ";;" oder ";-" eingeleitet werden.

```
optional-separator = ";" | ",".
lisp-comment      = (";;" | ";-") ANY*.
comment           = "--" ANY*.
```

Anhang C: Abweichende Syntaxregeln für die Implementierung

C.1: Zu Abschnitt 2:

```
function-definition  =  #!FCT fct-id-and-vars [externals] IS clause-sequ END-FCT.
fct-id               =  ID | MARK.
fct-id-and-vars      =  ( fct-id bound-vars )
                     |  ( SP-ID SP-bound-vars ).
SP-bound-vars        =  id-list ")".
```

C.2: Zu Abschnitt 3:

```
base-class-definition   =  #!CLASS ID [INHERIT ID ] [MIXINS id-list] [externals] IS
                           attrib* END-CLASS.
mixin-class-definition  =  #!MIXIN ID [MIXINS id-list] [externals] IS
                           attrib* END-CLASS.
```

C.3: Zu Abschnitt 4:

```
method-definition      =  #!METHOD method-id-ans-params [externals] IS
                          clause-sequ END-METHOD.
method-id              =  ID | MARK.
method-id-and-params   =  ( method-id "(" param* ")" )
                       |  ( SP-ID param* ")" ).
```

C.4: Zu Abschnitt 5:

```
clause       =  loop
             |  opd-                    [ "->" dest-part ]
             |  dest-part-                "<-" opd
             |  ID            [index]   [ "<-" opd | "->" dest-part ]
             |  OBJ-DESCR     [index]   [ "<-" opd | "->" dest-part ].
opd-         =  fct-or-method-appl | cond | list-constr | obj-constructor | "(" expr ")"
             |  lambda | literal | lisp-form.
dest-part-   =  list-descr.
```

C.5: Zu Abschnitt 6:

```
fct-or-method-appl  =  SP-ID opd* ")".
obj-constructor     =  OBJ-CONSTR init* "}".
```

C.6: Common Lisp - Formen

```
lisp-comment        =  ";" ANY*.
lisp-strg           =  '"' ( '""' | ANY-{'"'} )* '"'.
CommonLisp-Form     =  "(" ( NB | COMMON-LISP-STRG
                           | [ "'" ] (COMMON-LISP-ID | CommonLisp-Form)
                           | function-definition | class-definition | method-definition )* ")".
```

Anhang D: Semantikdefinition

In diesem Kapitel wird die Semantik der W-Lisp-Konstrukte durch Angabe des jeweiligen Common-Lisp-Äquivalentes definiert. Dies geschieht durch Definition der Übersetzungsfunktion τ. Argument dieser Funktion ist ein Programmfragment in konkreter W-Lisp-Syntax. τ wird für die meisten Syntax-Regeln definiert. Die Zugehörigkeit der Fragmente zu den Syntaxregeln ist aus dem Kontext ersichtlich. *Kursiv* gedruckte Wörter bezeichnen Variablen, die für Programmelemente stehen. Ihre Namen sind so gewählt, daß sie den zugehörigen Syntax-Regeln zugeordnet werden können. Schlüsselwörter sind unterstrichen.

D.1: Literale

D.1.1. Zahlen

τ(*nb*) = *nb*

D.1.2. Zeichen

τ($*char*) = #*char*

D.1.3. Strings

τ("*chars*") = "*chars*"

τ("$chars_1$""$chars_2$") = (concatenate ‘string τ("$chars_1$") (string #\") τ("$chars_2$"))

D.1.4. Wahrheitswerte, leere Liste

τ(true) = T

τ(false) = NIL

τ(nil) = NIL

D.1.5. Bezeichner

τ(*id*) = *id-* -- *id-* bezeichnet ein Bezeichnertoken mit dem Wert von *id*, ergänzt um "-"

τ(#*lisp-symbol*) = *lisp-symbol*

D.1.6. Lisp-Ausdrücke

τ(|*lisp-form*|) = *lisp-form*

D.1.7. Kommentare

Kommentare werden nicht nach Common-Lisp übersetzt.

D.2: Ausdrücke

D.2.1. Binäre Operatoren

τ(x_1 *op* x_2) = ($\Im$(*op*) τ(x_1) τ(x_2))

τ(x_1 op_1 ... op_{n-1} x_n) = ($\Im$(op_{n-1}) τ(x_1 op_1 ... op_{n-2} x_{n-1}) τ(x_n))

$\Im$ = { (+ → +), (- → -), (* → *), (/ → /), (= → equal), (/= → w-lisp::notequal-),
(>= → >=), (<= → <=), (< → <), (> → >), (equiv → w-lisp::equiv),
(in → w-lisp::in), (and → and), (or → or) }

x ∈ { expr, conj, relation, sum, term, primary, opd }

D.2.2. Monadische Operatoren

τ(not *opd*) = (not τ(*opd*))

τ(- *opd*) = (- τ(*opd*))

τ(+ *opd*) = τ(*opd*)

τ(‘*id*) = ‘τ(*id*)

D.2.3. Designatoren

τ(*x.id*) = (τ(*id*) τ(*x*))

τ(*x*[*opd*$_1$ ... *opd*$_n$]) = (nth τ(*opd*$_n$) τ(*x*[*opd*$_1$ ... *opd*$_{n-1}$]))

τ(*x*[*opd*]) = (nth τ(*opd*) τ(*x*))

τ(*x*[*opd*..]) = (nthcdr τ(*opd*) τ(*x*))

D.2.4. Bedingter Ausdruck

τ(if *expr*$_1$ then *clause*$_{1,1}$ ... *clause*$_{1,m}$
elsif *expr*$_2$ then *clause*$_{2,1}$ ... *clause*$_{2,m}$2
elsif ...
elsif *expr*$_n$ then *clause*$_{n,1}$ ... *clause*$_{n,m}$n
else *clause*$_1$ ... *clause*$_p$ fi) =
(cond (τ(*expr*$_1$) τ(*clause*$_{1,1}$) ... τ(*clause*$_{1,m}$))
(τ(*expr*$_2$) τ(*clause*$_{2,1}$) ... τ(*clause*$_{2,m}$2)) ...
(τ(*expr*$_n$) τ(*clause*$_{n,1}$) ... τ(*clause*$_{n,m}$n))
(T τ(*clause*$_1$) ... τ(*clause*$_p$)))

D.2.5. Funktionsanwendung

τ(*id*(*opd*$_1$ ... *opd*$_n$)) = (τ(*id*) τ(*opd*$_1$) ... τ(*opd*$_n$))

D.2.6. Lambda-Ausdruck

τ(*id*$_1$ ... *id*$_n$.*opd*) = #’(lambda (τ(*id*$_1$) ... τ(*id*$_n$)) τ(*opd*))

D.3: Konstruktoren

D.3.1. Listen

τ({}) = nil

τ({*elem*$_1$ ... *elem*$_n$}) = (append τ’(*elem*$_1$) ... τ’(*elem*$_n$))

τ’(*opd*) = (list τ(*opd*))

τ’(..*opd*) = τ(*opd*)

D.3.2. Objektkonstruktor

τ(*id*{*id*$_1$ = *opd*$_1$ ... *id*$_n$ = *opd*$_n$}) =
(make-instance ‘τ(*id*) ‘τ(*id*$_1$) τ(*opd*$_1$) ... ‘τ(*id*$_n$) τ(*opd*$_n$))

D.4: Klauseln

D.4.1. Schleifenanweisung

τ(while *expr* do *clause*$_1$ ... *clause*$_n$ od) =
(let (@res) (loop
(cond (τ(*expr*) τ(*clause*$_1$) ... τ(*clause*$_{n-1}$) (setq @res τ(*clause*$_n$))) (T (return @res)))))

D.4.2. Assoziation

τ(*opd -> dest-part*) = τ(*dest-part <- opd*)

$\tau(id \text{ <- } opd) = (\text{setf } \tau(id)\ \tau(opd))$

$\tau(obj\text{-}descr \text{ <- } opd) = (\text{setf } \tau(obj\text{-}descr)\ \tau(opd))$

$\tau(\text{<}descr_1 \ldots descr_n \text{ ..}id\text{>} \text{ <- } opd) = (\text{let } ((\text{@tmp } \tau(opd)))$
$\tau(descr_1 \text{ <- } (\text{nth } 0 \text{ @tmp})) \ldots \tau(descr_n \text{ <- } (\text{nth } n \text{ @tmp}))$
$(\text{setf } \tau(id)\ (\text{nthcdr } n+1 \text{ @tmp}))$
@tmp)

D.4.3. Result

$\tau(\underline{\text{result}}\ clause_1 \ldots clause_n) = (\text{prog1 } \tau(clause_1) \ldots \tau(clause_n))$

D.5: Definitionen

In Methoden- und Funktionsdefinitionen wird jede Variable, der ein Wert zugewiesen wird, als lokale Variable deklariert. Ausgenommen hiervon sind nur die Bezeichner, die entweder Parameter sind oder als *external* definiert wurden. Die Funktion *vars-of* bestimmt die Menge aller Bezeichner des ihr übergebenen Programmfragmentes, denen dort ein Wert zugewiesen wird. (Beachte: Ein Bezeichner wird nicht in diese Menge aufgenommen, wenn nur einer Komponente der Variablen mittels einer Feldselektion oder Indizierung ein Wert zugewiesen wird!)

$vars\text{-}of(clause_1 \ldots clause_k) = vars\text{-}of(clause_1) \cup vars\text{-}of(clause_2; \ldots; clause_k)$
$vars\text{-}of(id \text{ <- } opd) = \{id\}$
$vars\text{-}of(\text{<}name\text{-}part_1 \ldots name\text{-}part_n \text{ ..}id\text{>} \text{ <- } opd)$
$= \{id\} \cup vars\text{-}of(name\text{-}part_1 \text{ <- } opd) \cup \ldots \cup vars\text{-}of(name\text{-}part_n \text{ <- } opd)$
$vars\text{-}of(x) = \{\}$

D.5.1. Funktionsdefinition

$\tau(\underline{\text{fct}}\ id(id_1 \ldots id_n)\ \underline{\text{ext}}\ id_{n+1} \ldots id_m\ \underline{\text{is}}\ clause_1 \ldots clause_k\ \underline{\text{end-fct}}) =$
$(\text{defun } \tau(id)\ (\tau(id_1) \ldots \tau(id_n))$
$(\text{let } (local_1 \ldots local_p)\ \tau(clause_1) \ldots \tau(clause_k)))$

mit $\{local_1 \ldots local_p\} = vars\text{-}of(clause_1 \ldots clause_k) \setminus \{id_1 \ldots id_n\ id_{n+1} \ldots id_m\}$

D.5.2. Methodendefinition

$\tau(\underline{\text{method}}\ id(id_1\ \underline{\text{of}}\ id'_1 \ldots id_n\ \underline{\text{of}}\ id'_n)\ \underline{\text{ext}}\ id_{n+1} \ldots id_m\ \underline{\text{is}}\ clause1 \ldots clause_k\ \underline{\text{end-fct}}) =$
$(\text{defmethod } \tau(id)\ ((\tau(id_1)\ (\tau(id'_1)) \ldots (\tau(id_n)\ \tau(id'_n)))$
$(\text{let } (local_1 \ldots local_p)\ \tau(clause_1) \ldots \tau(clause_k)))$

mit $\{local_1 \ldots local_p\} = vars\text{-}of(clause_1; \ldots; clause_k) \setminus \{id_1, \ldots, id_n, id_{n+1}, \ldots, id_m\}$

D.5.3. Klassendefinition

$\tau(\underline{\text{class}}\ id\ \underline{\text{inherit}}\ id_1\ \underline{\text{mixin}}\ id_2 \ldots id_n\ \underline{\text{is}}\ attrib_1 \ldots attrib_m\ \underline{\text{end-class}}) =$
$(\text{defclass } \tau(id)\ (\tau(id_n) \ldots \tau(id_1))\ (\tau'(attrib_1) \ldots \tau'(attrib_m)))$

$\tau'(id = opd) = (\tau(id) \text{ :accessor } \tau(id) \text{ :initarg } \tau(id) \text{ :initform } \tau(\text{opd}))$

$\tau'(id) = (\tau(id) \text{ :accessor } \tau(id) \text{ :initarg } \tau(id))$

D.5.4. Mixin-Klassendefinition

$\tau(\underline{\text{mixin}}\ id\ \underline{\text{inherit}}\ id_1\ \underline{\text{mixin}}\ id_2 \ldots id_n\ \underline{\text{is}}\ attrib_1 \ldots attrib_m\ \underline{\text{end-class}}) =$
$(\text{defmixin } \tau(id)\ (\tau(id_n) \ldots \tau(id_1))\ (\tau(attrib_1) \ldots \tau(attrib_m)))$

D.6: Laufzeitfunktionen

D.6.1. Prädikat IN

```
fct #in (elem list) is #member(elem list #:test '#equal) end-fct
```

D.6.2. Äquivalenz

```
fct #equiv (x y) is if x then y else (not y) fi end-fct
```

D.6.3. Ungleichheit

```
fct notequal (x y) is (not #equal(x y)) end-fct
```

D.7: Schnittstelle des W-Lisp-Systems

Das W-Lisp-System ist im Package w-lisp implementiert und exportiert die Bezeichner "sc", "sc-", "load-", "vload-" und "compile-file-".

Anhang E: Abbildung vom Meta Class System (MCS) nach CLOS

Die aktuelle Implementierung des objektorientierten Teils von W-Lisp beruht auf dem Meta Class System (MCS) der Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD) [Bretthauer, Kopp]. Dieser Abschnitt definiert die notwendigen Common Lisp - Formen für die Umsetzung der MCS-Konstrukte in das Common Lisp Object System (CLOS) [Steele 90].

```
(defmacro defmixin (       class-name
                           direct-superclasses
                           direct-slots
                           &rest options)
  `(defclass ,class-name ,direct-superclasses
    ,direct-slots
    ,@options))
```