Lexikalische Elemente (1)

Lexikalische Elemente sind:

- besondere **Begrenzungszeichen** (delimiter),
- **Bezeichner** (identifier),
- **Kommentare** (comment),
- **Zeichen** (character literal),
- **Zeichenketten** (string literal),
- **Bitzeichenketten** (bit string literal) und
- **abstrakte Zahlen** (abstract literal).

Lexikalische Elemente werden in der Sprache getrennt durch ***Separatoren***:

- Leerzeichen,
- Tabulator und
- Zeilenende-Zeichen.

Begrenzungszeichen (delimiter)

- für Operationen (Operatoren),
- als Begrenzer für bestimmte Teilkonstrukte der Sprache und
- zur Interpunktion.

" # & ' ( ) * + - , . / : ; < = > [ ] |

Paare von Sonderzeichen

=> ** := /= >= <= <>

<= steht für den Vergleich: kleiner oder gleich in Ausdrücken
ist aber auch das Zuweisungssymbol für Signale

Begrenzungszeichen können Separatoren ersetzen, z.B.

(A + B) kann auch geschrieben werden ***(A+B)***, also ohne Leerzeichen zwischen Operanden und Operator (Begrenzerfunktion von **+**).

Lexikalische Elemente (2)

Bezeichner (identifier)
benennen Benutzer-**Objekte** in VHDL, wie z.B. **Variable**, **Signale**, **Prozesse** u.s.w.
für die Sprache selbst belegte Bezeichner sind **reservierte Worte**

- sie dürfen nur Groß- und Kleinbuchstaben, Zahlen und den Unterstrich "_" enthalten,
- sie müssen mit einem Buchstaben beginnen,
- sie dürfen *nicht* mit einem Unterstrich beginnen oder enden und
- sie dürfen *keine zwei aufeinander folgenden* Unterstriche beinhalten.

erlaubte Bezeichner:

COUNT ist identisch **count**; **last_value**, **h3Z25**, **Date_2305**

unerlaubter Bezeichner:

last@value	-- illegales Zeichen
5bit_counter	-- beginnt mit numerischen Zeichen
_A0	-- beginnt mit Underline
A0_	-- endet mit Underline
clock_ _pulse	-- zwei Underlines
end	-- reserviertes Wort ist nicht erlaubt

reservierte Worte in VHDL sind:

abs	disconnect	label	package	sla
access	downto	library	port	sll
after	else	linkage	postponed	sra
alias	elsif	literal	procedure	srl
all	end	loop	process	subtype
and	entity	map	protected	then
architecture	exit	mod	pure	to
array	file	nand	range	transport
assert	for	new	record	type
attribute	function	next	register	unaffected
begin	generate	nor	reject	units
block	generic	not	rem	until
body	group	null	report	use
buffer	guarded	of	return	variable
bus	if	on	rol	wait
case	impure	open	ror	when
component	in	or	select	while
configuration	inertial	others	severity	with
constant	inout	out	shared	xnor
	is		signal	xor

Lexikalische Elemente (3)

Kommentare (comment)
beginnen mit dem Doppelminus “--“ und enden am Zeilenende:

```
--      es folgt eine Variablenzuweisung
C := A + B;          -- dies ist eine gültige Variablenzuweisung
```

Zeichen (character literal)
werden in VHDL in Hochkommas eingerahmt:

Großbuchstaben: **'A'**, **'B'**, ..., **'Z'**;   Kleinbuchstaben: **'a'**, **'b'**, ..., **'z'**;
Ziffern: **'0'**, **'1'**, ..., **'9'**;   Sonderzeichen: **'!'**, **'$'**, **'&'**, ...;
Hochkomma: **'''** und   Leerzeichen: **' '**.

Zeichenketten (string literal)
bestehen aus einer Folge von Einzelzeichen und werden in Anführungszeichen angegeben.

"Eine Zeichenkette" oder auch **"any printing chars (&%@^*)"**,
"0011ZZZ" oder auch **""**, der leere String.

Zeichenketten, die nicht in eine Zeile passen, können auch mit dem *Kettungsoperator für Strings*, dem "**&**" aneinander gekettet werden, wie z.B.

"Eine Zeichenkette, die nicht in eine Zeile passt" &
"kann mit dem Operator & gekettet werden"

um die zwei Teilstrings zu einem zu binden.

Im Praktikumsversuch 2 wird der Datentyp ***bit_vector*** verwendet, der einen String darstellt (ein String ist in VHDL ein eindimensionales Feld, ein Vektor). Das Verschieben eines 8 stelligen strings **d** wird durch die Verkettung des ***character_literal*** **'0'** mit dem String (bit_vector) von links nach rechts ohne die letzte Stelle erreicht:

```
d := ‘0’ & d(7 downto 1);         -- Verkettung von Zeichenketten mit &
```

Lexikalische Elemente (4)

Bitzeichenketten (***bit string literal***)
geben Zahlenwerte an.

- **binäre Angaben** mit **B"..."**,
- **oktale Angaben** mit **O"..."** oder
- **hexadezimale Angaben** mit **X"..."** .

Entsprechend sind die in Bitzeichenketten zugelassenen Zeichen

- bei **B** nur **0** oder **1**,
- bei **O** sind es die Zeichen **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7** und
- bei **X** die Zeichen **0**, **1**, ..., **9**, **A**, **B**, **C**, **D**, **E**, **F**.

Bitketten ermöglichen z.B. Speicherinhalte nicht immer binär sondern auch gekürzt als Oktal- oder Hexadezimalwert anzugeben. Das Resultat ist in jedem Fall ein als Bitstring angegebenes Bitmuster.

B"0100011", oder **B"10"**, oder auch **b"1111_0010_1010_0101"**, oder **B""**.

Oktal angegebene Strings fassen 3-bit Gruppen zusammen, wie

O"372" ist äquivalent zu **B"011_111_010"**, oder auch
o"00", das äquivalent **B"000_000"** ist.

Hexadezimal angegebene Strings fassen 4-bit Gruppen zusammen, wie

X"FA" äquivalent zu **B"1111_1010"**, oder auch
x"0d" äquivalent zu **B"0000_1101"** .

> Man beachte, dass **O"372"** <u>nicht</u> äquivalent zu **X"FA"** ist, da das erste Literal 9 Bitstellen anspricht, der letze jedoch nur 8!!!

Zahlenangaben (abstract literal)
wir können zwischen **Ganzahlen** (**Integer**) und
gebrochen rationalen Zahlen (**Real**) unterscheiden.

Integer (integer literal)
besteht aus Stellen (**digits**) ohne einen Dezimalpunkt.
Dezimale Integer-Literale sind z.B. ***23 0 146***.

Reals (real literal)
haben immer einen Dezimalpunkt, durch den sie sich von den Integer Literalen unterscheiden.
Dezimale Real Literale sind z.B. ***23.1 0.0 3.14159***.

Ohne Beweis sollte klar sein, dass die maschinelle Repräsentation solcher Rationalzahlen auf Grund der Stellenbeschränkung nur eine Näherung darstellen kann.

Lexikalische Elemente (5)

abstract_literal ist einerseits ***decimal_literal*** und andererseits ***based_literal***. Beide Literale können auch in der sog. **E-Notation** erscheinen. Die Zahl hinter dem "**E**" oder dem "**e**" bezeichnet die Anzahl der Positionen, die der Stellenpunkt **nach links** (**negativ**) oder **nach rechts** zu verschieben ist.

Beispiele dezimaler Literale:
Integerzahlen:

46E5	entspricht der Ganzzahl	4600000,	
1E+12	ist entsprechend	10^{12}	und
19e00	ist gleich	19.	

Realzahlen in E-Notation:

1.234E09	ist	$1.234*10^{9}$,
98.6E+21	ist	$0.986*10^{23}$ und
34.0e-08	ist	0.00000034.

Man kann in VHDL auch die Zahlen in verschiedenen Basis-Systemen als sog. ***based_literal*** angeben. Die Basis muss zwischen 2 und 16 liegen!

2#11111101#	(Dualzahl)
= **8#375#**	(Oktalzahl)
= **16#0fd#** = **16#FD#**	(Hexadezimalzahl).

Dies gilt auch für Real-Zahlen. Als Beispiel ist **dezimal 0.5** gleich

2#0.100# (Dualzahl) = 8#0.4# (Oktalzahl) = 12#0.6# (Basis 12 Zahl)

Auch die E-Notation gilt:

2#1#E10 = 2^{10}, 16#4#E2 = $4*16^{2}$ 10#1024#E+00 = 1024.

Man beachte, dass die Zahlenangabe hinter dem „E“ eine dezimale ist.

Aus Gründen der besseren Lesbarkeit kann man auch Gruppen von Ziffern mit dem Unterstrich "_" abtrennen, ähnlich der gewohnten Tausender-Trennung im Dezimalen.

123_456 , 3.141_592_6 , 2#1111_1100_0000_0000#.

Erweiterte Backus-Naur Form (EBNF)

Produktionsregel für syntaktische Einheit ***Variablenzuweisung***:

variable_assignment_statement ::= target **:=** expression;

::=	bedeutet „ist definiert zu“;
target	Bezeichner hier für eine Variable,
:=	Begrenzer, hier mit der Bedeutung „wird zugewiesen“ und
expression	(Ausdruck) weitere Einheit; wieder in EBNF beschrieben.

Die wait-Anweisung ist in der EBNF im VHDL LRM beschrieben:

wait_statement ::= wait [sensitivity_clause**] [**condition_clause**] [**timeout_clause**];**

Dem Schlüsselwort **wait** folgen drei geklammerte Klauseln.
[...] bedeutet, dass der Inhalt der Klammer optional ist, d.h. auch fehlen darf.

Weiter abgeleitet ergibt sich für die erste Klammer:

sensitivity_clause ::= on sensitivity_list

Dem Schlüsselwort **on** folgt eine weitere Einheit (Sensitäts-Signalliste), die weiter abzuleiten ist:

sensitivity_list ::= *signal*_name **{ ,** *signal*_name **}**

Die syntaktische Einheit *signal*_name steht in der geschweiften Klammer nach einem Begrenzer Kommabegrenzer.
{ ... } bedeutet, dass der Inhalt ***nicht oder beliebig oft*** wiederholt vorkommen darf.

Die Einheit ***Signalname*** ist gegeben zu:

***signal*_name ::=** identifier

womit die Regel-Ableitung an einem Terminal ***identifier*** (Bezeichner, s.o.) endet.

Die Bedingungsklausel ist wie folgt abzuleiten:

condition_clause ::= until condition

condition ::= *boolean*_expression

Eine Bedingung nach dem Schlüsselwort ***until*** ist somit ein boolescher Ausdruck, also einer der zu ***true*** oder ***false*** evaluierbar ist.

Die Zeitklausel beginnt mit dem Schlüsselwort ***for*** und verlangt einen Ausdruck, der in einen ***Zeitwert*** evaluiert werden soll:

timeout_clause ::= for *time*_expression

Zuweisungen

Man erkennt, dass der Ausdruck (expression) formal eine sehr komplexe syntaktische Einheit ist und teilweise **semantische Details** (in Kursivschrift angedeutet) genauere Aussagen treffen.

Wir wollen etwas abweichen vom LRM unter Wertbestimmungs-Ausdruck (*value*_expression) verstehen:

***value*_expression** ::= *numerical*_expression **|** *boolean*_expression

Die obige Zeitwert-Ausdruck (*time*_expression) ist eine besondere Wertbestimmung.

Unter numerischer Wertbestimmung

***numerical*_expression** ::= [+ | -] *num*_term { (+ | - |) *num*_term }
***num*_term** ::= [abs] *numerical*_factor { (* | / | mod | rem) *numerical*_factor }
***numerical*_factor** ::= constant **|** variable **|** signal -- als numerischen Typ

Die Wertbestimmung eines booleschen Typs (true | false):

condition ::= *boolean*_expression
***boolean*_expression** ::= [not] *boolean*_factor { (and | or | ....) [not] factor }
***boolean*_factor** ::= constant **|** variable **|** signal -- als boolean Typ

Die Bedingung ist also auch ein *boolean*_expression!

Variablenzuweisung (wie oben)

variable_assignment_statement ::= target **:=** expression;

Signalzuweisung (sequenziell)

signal_assignment_statement ::= target **<=** {delay_mechanism] waveform;
delay_mechanism ::= **transport** | [**reject** *time*_expression] **inertial**
waveform ::= waveform_element { , waveform_element }
waveform_element ::= *value*_expression [**after** *time*_expression]

Prozessdeklaration

process_statement ::=
[*process*_label **:**]
  process [(sensitivity_list)] [**is**]
    process_declarative_part
  begin
    process_statement_part
  end process [*process*_label]**;**

process_statement_part ::= { sequential_statement }

Architekturdeklaration

architecture_body ::=
  architecture identifier **of** *entity*_name **is**
    architecture_declarative_part
  begin
    architecture_statement_part
  end [architecture] [*architecture*_name]**;**

architecture_statement_part ::= { concurrent_statement }

Blockdeklaration

block_statement ::=
[*block*_label **:**]
  block [(*guard*_expression)] [**is**]
    block_header
    block_declarative_part
  begin
    block_statement_part
  end block [*block*_label]**;**

block_statement_part ::= { concurrent_statement }

Befehlsfolgen in Prozessen

sequence_of_statements ::= { sequential_statement }

sequential_statement ::= wait_statement
| assertion_statement
| report_statement
| signal_assignment_statement
| variable_assignment_statement
| procedure_call_statement
| if_statement
| case_statement
| loop_statement
| next_statement
| exit_statement
| return_statement
| null_statement

nebenläufige Befehle in Architektur und in Blöcken

block_statement_part ::= { concurrent_statement }

concurrent_statement ::= block_statement
| process_statement
| concurrent_procedure_call_statement
| concurrent _assertion_statement
| concurrent _signal_assignment_statement
| component_instantiation_statement
| generate_statement

Aus-/Einschluss bestimmter Signalzuweisungen in Blöcken (guarded blocks)

Guard ist ein logisches Signal (true, false) und hat steuernde Funktion für entsprechend gekennzeichnete Signalzuweisungen.

- ist der ***Guard = true***, dann werden alle Guarded-Signalzuweisungen im Block ausgeführt, wenn sich die Zuweisungswerte ändern,
- ist der ***Guard = false***, werden diese Signalzuweisungen deaktiviert und
- ***wechselt der Guard von false nach true***, werden die Signalzuweisungen auf jeden Fall ausgeführt.

Guarded Block Konstrukt

```
GaBlo:
 block (guard-expression)
  signal sig: bit;
begin
  sig <= guarded waveform-elements;
end block GaBlo;
```

vergleichbares Prozess-Konstrukt

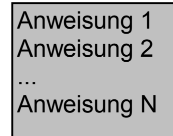
```
architecture …..
  signal sig: bit;
  signal guard: boolean;
begin
….
  guard <= guard-expression;

  GaBlo:
  process (guard, signals in waveform-elements) is
  begin
    if guard then
    sig <= waveform-elements;
    end if;
  end process GaBlo;
end architecture ...;
```

Steuerung des Programmflusses

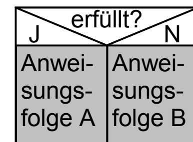
Struktogramm-Elemente zur Darstellung des Steuerflusses bei sequenzieller Bearbeitung mit den entsprechenden Sprachkonstrukten.

sequence_of_statements ::= { sequential_statement }

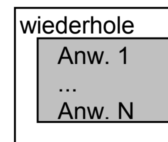


1. Sequenz

sequential_statement ::= wait_statement
| assertion_statement
| report_statement
| signal_assignment_statement
| variable_assignment_statement
| procedure_call_statement
| if_statement
| case_statement
| loop_statement
| next_statement
| exit_statement
| return_statement
| null_statement



2. Selektion



3. Iteration

IF-Anweisungen (Alternative)

EBNF Produktionsregel:

```
if_statement ::=
  [ if_label : ]
  if condition then
    sequence_of_statements
  { elsif condition then
      sequence_of_statements }
  [ else
      sequence_of_statements ]
  end if [ if_label ];
```

Im Versuch 2 ist ***d*** deklariert als 8-stelligen Bitvektor in der umgekehrten Stellenfolge

```
variable d : bit_vector(7 downto 0);
```

Das einfachste *if* Konstrukt ohne *elsif* oder *else* löst die Abfrage, ob das niedrigstwertige Bit 0 gesetzt ('1') ist:

einfaches if Konstrukt

```
if (d and "00000001") = "00000001" then
  n := n + 1;
end if;
```

erfüllt?
J | N
Anweisungsfolge A

if-then-else Konstrukt

```
if (d and "00001111") = "00000000" then
  no1_lh <= false;
else
  no1_lh <= true;
end if;
```

erfüllt?
J | N
Anweisungsfolge A | Anweisungsfolge B

if Schachtelung

```
if (d and "00001111") /= "00000000" then
  no1_lh <= false;
  if (d and "00000011") = "00000000" then
    no1_lq <= true;
  else
    no1_lq <= false;
  end if;
else
  no1_lh <= true;
  no1_lq <= true;
end if;
```

erfüllt?
J | N
erfüllt?
J | N
Anweisungsfolge A | Anweisungsfolge B | Anweisungsfolge C

Vergleich der Alternativen im sequenziellen Fluss und in nebenläufigen Signal-zuweisungen:

if- elsif Konstrukt

```
if en = '0' then
  result <= 0;
elsif op = yes then
  result <= input_0;
else
  result <= input_1;
end if;
```

EBNF der nebenläufigen Signalzuweisung

```
concurrent_signal_assignment_statement ::=
    [ label : ] conditional _signal_assignment
|   [ label : ] selected _signal_assignment

conditional_signal_assignment ::=
    target <= options conditional_waveforms;

options ::= [ guarded ] [ delay_mechanism ]

conditional_waveforms ::=
    { waveform when condition else }
      waveform [ when condition ]
```

Setzen wir das obige elsif-Konstrukt um in eine solche Signalzuweisung mit gleichem Verhalten, so folgt:

```
result <=   0  when  en = '0' else
        input_0 when  op = yes else
        input_1;
```

Case-Anweisungen (Selektion, Fallunterscheidung)

EBNF:

```
case_statement ::=
  [ case_label : ]
 case expression is
   case_statement_alternative
   { case_statement_alternative }
 end case [ case_label ];

case_statement_alternative ::=
 when choices =>
    sequence_of_statements

choices ::=
   expression
 | discrete range
 | element_name
 | others
```

Beispiel:

```
 case state is
   when 0              => act := 0;
   when 1 to 7         => act := 1;
   when 15 downto 8    => act := 2;
   when others         => act := 3;
 end case;
```

*Die Auswahl **choices** muss alle auftretenden Fälle abdecken! Daher wird **others** benötigt und ggf. auch das **null** Statement.*

```
type state is (idle, active, waiting, undefined);
.....
 case state is
   when active    => act := 0;
   when waiting   => act := 1;
   when others    => act := 3;
 end case;


case state is
   when active    => act := 0;
   when waiting   => act := 1;
   when others    => null;
 end case;
```

Vergleich der Selektion im sequenziellen Fluss und in nebenläufigen Signalzuweisungen:

In einem Prozess z.B.:

```
case d is
  when "00000000" => intsig <= 0;
  when "00000001" => intsig <= 1;
  when "00000010" => intsig <= 2;
  when "00000011" => intsig <= 3;
  ........
  when others => intsig <= 255;
end case;
```

EBNF (2. Teil der nebenläufigen Signalzuweisung):

```
selected_signal_assignment  ::=
  with expression select
     target <= options selected_waveforms;

selected_waveform  ::=
  { waveform when choices, }
    waveform when choices
```

obiges Case-Konstrukt umgesetzt ergibt:

```
with d select
  intsig <= 0 when "00000000",
  intsig <= 1 when "00000001",
  intsig <= 2 when "00000010",
  intsig <= 3 when "00000011",
............
  intsig <= 255 when others;
```

LOOP-Anweisung (Programmschleife)

EBNF der Schleife:

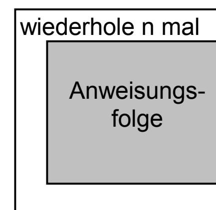
loop_statement ::=
 [*loop*_label:]
 [iteration_scheme] **loop**
  sequence_of_statements
 end loop [*loop*_label];

iteration_scheme ::=
 while condition
 | for *loop*_parameter_specification

FOR-LOOP-Anweisung (Zählschleife)

Die *for*-Schleife ist ein Konstrukt aus mehreren möglichen Schleifenkonstrukten:

***for-loop*_Anweisung** ::=
 [*loop*_Bezeichner:]
 for Zähler **in** diskreter_Bereich **loop**
  sequence_of_statements
 end loop [*loop*_Bezeichner];



Strukturblock (Struktogramm, rechts) ist die grafische Darstellung dieses Konstrukts.
Der *Zähler* ist eine *implizit vereinbarte* Variable zum Zählen der Durchläufe.

Zwei verschiedene Variablen ***a***; ***a*** ist am Ende des Prozesse gleich 10!

```
example: process is
  variable a, b: integer;
begin
  a := 10;
  for a in 10 downto 5 loop
    b := a;
  end loop;
end process;
```

Aufzählung als Schleifenzähler; ***a*** ist nur innerhalb der Schleife definiert!

```
type state is (initial, idle, active, error);
variable b: state;
.....
for a in state loop
  b := a;
end loop;
```

Zählschleife aus Praktikumsversuch

Im Versuch 2 wollen wir die Anzahl der zu '1' gesetzten Bits im Bitvektor ***d*** mit einer Integer Variablen ***n*** zählen, die vorher zu Null gesetzt war.

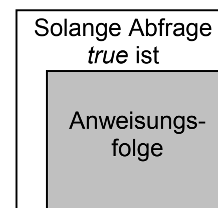
Das For-Schleifenkonstrukt in Verbindung mit den oben angegebenen zusätzlich benötigten Konstrukten IF und der Verschiebeoperation lösen das Problem:

```
for k in 1 to 8 loop                  -- for Schleife
  if (d and "00000001") = "00000001" then
    n := n + 1;                       -- zähle die '1' en
  end if;
  d := '0' & d(7 downto 1);           -- verschiebe nach rechts
end loop;
```

WHILE-LOOP (abweisende Schleife)

Schleife kann auch umgangen werden (abweisen, wenn Abfrage false); *EBNF*:

```
while-loop_Anweisung  ::=
   [ loop_identifier:]
  while Abfrage loop
    sequence_of_statements
  end loop [ loop_ identifier ];
```



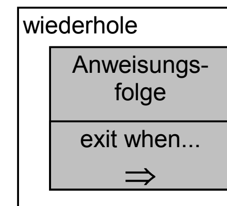
Beispiel: am Ende des Prozesses ist *a = 4* und *b = 5* !

```
alt_example: process is
  variable a, b: integer;
begin
  a := 10;
  while a >= 5 loop
    b := a;
    a := a - 1;
  end loop;
end process;
```

(REPEAT-) LOOP (allgemeine und annehmende Schleife)

Die *EBNF* der allgemeinen Schleife, die prinzipiell endlos ist, aber mit der speziellen *exit*-Anweisung abgebrochen werden kann:

***loop*_Anweisung** ::=
  [*loop_* identifier:]
 loop
  sequence_of_statements
 end loop [*loop_* identifier];



Mit den folgenden Sonderanweisungen kann man das Schleifenkonstrukt steuern:

exit_statement ::= [label **:**] **exit** [*loop*_label] [**when** condition]**;**

next_statement ::= [label **:**] **next** [*loop*_label] [**when** condition]**;**

Beispiel annehmende Schleife (repeat-until) im Vergleich zu oben:

```
neu_example: process is
 variable a, b: integer;
begin
 a := 10;
 loop
  b := a;
  a := a - 1;
  exit when a < 5
 end loop;
end process;
```

*Beipiel mit **next**-Anweisung; **b** nimmt nur **gerade** Werte von **a** an:*

```
variable a, b: integer;
.....
a := 0;
loop
 a := a + 1;
 exit when a > 16;
 next when a mod 2 = 0;
 b := a;
end loop;
```

Weitere Beispiele mit Aufzählungen:

```
type state is (initial, idle, active, error);
variable a, b: state;
.....
a := state'left;
loop
  exit when a = state'right;
  b := a;
  a := state'succ(a);
end loop;
```

```
type state is (initial, idle, active, error);
variable a, b: state;
.....
a := state'right;
loop
  b := a;
  a := state'pred(a);
  if a = state'left then
    b := a;
    exit;
  end if;
end loop;
```

Unterprogramme (Procedure und Function)

***subprogram*_body** ::=
subprogram_specification **is**
subprogram_declarative_part
begin
subprogram_statement_part
end [subprogram_kind] [designator]**;**

***subprogram*_specification** ::=
procedure identifier [**(** formal_parameter_list **)**]
| **function** designator [**(** formal_parameter_list **)**]
return type_mark

EBNF der Prozedur-Deklaration:

***procedure*_subprogram_body** ::=
procedure identifier [**(** formal_parameter_list **)**] **is**
subprogram_declarative_part
begin
subprogram_statement_part
end [**procedure**] [identifier]**;**

*Die **Schnittstelle** zu rufenden Programmen ist der Name **identifier** und die Formal-Parameterliste*:

formal_parameter_list ::= *parameter*_interface_list

interface_list ::= interface_element { **;** interface_element }

interface_element ::= identifier_list : mode type_name

mode ::= in **|** out **|** inout

EBNF der Funktions-Deklaration:

***function*_subprogram_body** ::=
function designator [**(** formal_parameter_list **)**]
return type_mark **is**
subprogram_declarative_part
begin
subprogram_statement_part
end [**function**] [identifier]**;**

*Im **subprogram_statement_part** muss mindestens ein*

return expression;

*vom Typ **type_mark** angegeben werden.*

EBNF zum Prozeduraufruf

procedure_call ::= *procedure*_name [**(** actual_parameter_part **)**]

actual_parameter_part ::= *parameter*_association_list

association_list ::= association_element { **,** association_element }

association *element* ::= [formal_part **=>**] actual_part

Beispiel-Vereinbarungen

procedure myproc (a, b, c : in integer; y : out boolean; z : inout positive) **is**
.....
begin
.....
end procedure;

function myfunct (a, b, c : in positive) **return** boolean **is**
.....
begin
.....
return true;
.....
end procedure;

Spezifikationen zu den Beispielen

procedure myproc (a, b, c : in integer; y : out boolean; z : inout positive)**;**

function myfunct (a, b, c : in positive) **return** boolean**;**

Erlaubte Aufrufe zu den Beispielen

myproc (av, bv, cv, yb, zvar)**;**
myproc (16*35, bv, cv, yb, zvar)**;**
myproc (33, bv, 55, yb, zvar)**;**
myproc (33, b, c, y, z)**;**

Nicht erlaubte Aufrufe zu den Beispielen

myproc (av, bv, cv, true, zvar)**;**
myproc (true, bv, cv, 20, zvar)**;**
myproc (33, bv, -55, yb, 25)**;**
myproc (33, b, c, yb, -77)**;**

Positionsunabhängige Aktualparameter-Übergabe

myproc (a => av, b => bv, c => cv, y => yb, z => zvar)**;**
myproc (b => bv, z => zvar, c => cv, y => yb, a => av)**;**

Prozedurbeispiel: Rotation von Bitvektoren

```
-- vorher war deklariert
  subtype myvector is bit_vector(7 downto 0);
  variable d : myvector;
.....
procedure rotate (d : inout myvector; nb : in natural; dir : in bit) is
  variable merk : bit;
begin
  for i in 0 to nb loop
    if dir = '0' then
      merk := d(0);
      d := merk & d(7 downto 1);
    else
      merk := d(7);
      d := d(6 downto 0) & merk;
    end if;
  end loop;
end procedure;
```

rotate(d, 3, '0')   - rotiere d um 3 Stellen nach rechts

rotate(d, 1, '1')   - rotiere d um 1 Stelle nach links

rotate(d=>d, dir=>'0', nb=>0)   - rotiere d keine Stelle

EBNF zum Funktionsaufruf

function_call ::= *function*_name [**(** actual_parameter_part **)**]

Funktionsaufruf

```
y := myfunct (av, bv, cv);
y := myfunct (b, c, 20);
y := myfunct (33, bv, -55);
y := myfunct (33, 44, -77);
```

Funktionsbeispiel: Wertermittlung für Bitvektor als Dualzahl interpretiert

```
architecture base of convert is
  subtype myvector is bit_vector(7 downto 0);
  signal a: integer;
  signal b: myvector;

  function value(vect : myvector) return integer is
    variable val: integer := 0;
  begin
    for i in 0 to 7 loop
      val := val + a * 2**i;
    end loop;
    return val;
  end function;

begin
  b <= "00001111", "00100001" after 2 ns,
        "00111111" after 3 ns, "11111111" after 4 ns;

  a <= value(b);

end base;
```

Sichtbarkeit von Vereinbarungen

```
architecture arch of ent is
  type t is ...;
  signal s : t;

  procedure p1 (...) is
    variable v1 : t;
  begin
    v1 := s;
  end procedure p1;

begin  -- arch

  proc1:
  process is
    variable v2 : t;

    procedure p2 (...) is
      variable v3 : t;
    begin
      p1 (v2, v3, ...);
    end procedure p2;

  begin   -- proc1
    p2 (v2, ...);
  end process proc1;

  proc2:
  process is
    ....
  begin  -- proc2
    p1 (...);
  end process p2;

end architecture arch;
```

Überdeckung von Vereinbarungen in Schachtelungen

```
procedure p1 is
  variable v : integer;

  procedure p2 is
    variable v : integer;
  begin  -- p2
    .....
    v := v + 1;
    .....
    p1.v := v + 33;      -- visibility by selection
    .....
    end procedure p2;

begin   -- p1
    .....
    v := 2 * v;
    .....
end procedure p1;
```

Programmbeispiel:
Sichtbarkeit von Parametern; sequenzieller Prozeduraufruf

```
entity visibility_1 is
end visibility_1;

architecture algo of visibility_1 is
begin

  visibility:
  process is
    variable A, B : integer := 200;
    variable Var_Out : integer;
    constant Fak : integer := 55;

    procedure InnerA (Ab, Ac : in Positive; Re : inout integer) is
      variable Var_Out : positive;
      variable Var_In : integer := 333;
    begin  -- InnerA
      Var_Out := Ab * Ac + Re;
      Re := Var_Out + Var_In;
    end InnerA;

    function InnerB (Bb, Bc : in natural) return integer is
--    procedure InnerB (Bb, Bc : in natural) is
--      variable Var_Out : positive;
--      variable A : positive;
    begin    -- InnerB
      Var_Out := Bb * Bc;
      A := Bb + Bc;
      return A;
    end InnerB;

--  variable Var_Out : integer;
  begin  -- of process
    Var_Out := 30;
    InnerA (50, 70, Var_Out);
--    InnerB (11 * 34, Fak);   -- procedure call
    B := InnerB (11 * 34, Fak);   -- function call
    wait;
  end process;
end architecture algo;
```

Nebenläufiger Prozeduraufruf

Vereinbarung und Aufruf einer Prozedur in der Architektur, z.B.

```
mypro(s1, s2, val1);
```

ist im Verhalten identisch zu dem des Prozesses mit sequenziellem Prozeduraufruf:

```
process is
begin
  mypro (s1, s2, val1);
  wait on s1, s2;
end process;
```

Programmbeispiel (Rahmenvorgabe): Sichtbarkeit von Parametern; nebläufiger Prozeduraufruf

```
entity visibility_2 is
end visibility_2;

architecture algo of visibility_2 is
  signal as, bs;

  procedure outer(signal as, bs: inout integer) is
    --  Deklarationen und Schachtelungen
  begin  -- outer
    .....
  end procedure outer;

begin  -- architecture
  outer(as, bs);
end architecture algo;
```

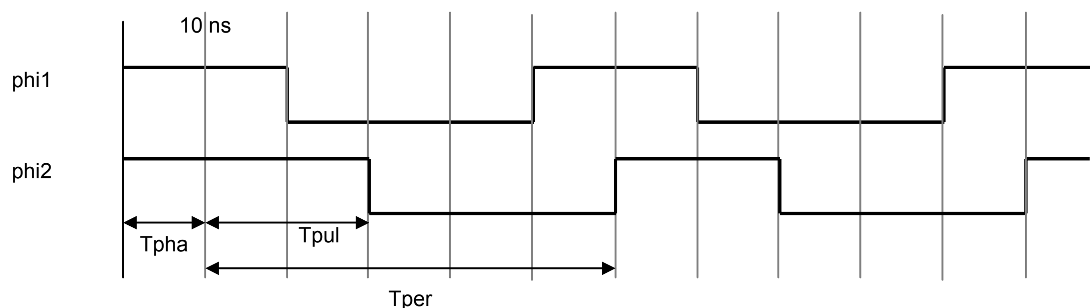
Programmbeispiel:
Zweiphasentakt mit nebenläufigen Prozeduraufrufen

```
entity proc2clk is
end proc2clk;
architecture algo of proc2clk is
  signal phi1, phi2 : bit;

  procedure generate_clock(signal clk : out bit;
                          constant Tper, Tpul, Tpha: in time) is
  begin  -- of procedure generate_clock
    wait for Tpha;
    loop
      clk <= '1', '0' after Tpul;
      wait for Tper;
    end loop;
  end generate_clock;

begin    -- architecture
  gen_phi1: generate_clock(phi1, Tper => 50 ns, Tpul => 20 ns, Tpha => 0 ns);
  gen_phi2: generate_clock(phi2, Tper => 50 ns, Tpul => 20 ns, Tpha => 10 ns);
end architecture algo;
```

Erzeugt Zweiphasentakt mit
Tpha: Phasenverschub
Tpul: Aktivzeit (Signal = ’1’)
Tper: Periodendauer



Package Konzept

Package Vereinbarung zur Zusammenfassung von Modellteilen:

package_declaration ::=
  package identifier **is**
    { package_declarative_item }
  end [**package**] [identifier]**;**

Package Rumpf für Prozeduren und Funktionen, wenn solche in der Vereinbarung sind:

package_body ::=
  package body identifier **is**
    { package_body_declarative_item }
  end [**package**] [identifier]**;**

Packages sind in Bibliotheken gespeichert.
Standard Bibliotheken sind STD und WORK, die implizit referiert sind.
Referieren von Bibliotheken mit der library Klausel, z.B. greift

library ieee;

auf die Bibliothek ***ieee*** zu.
Diese enthält unter anderem ein Package ***std_logic_1164***.
In diesem Package ist u.a. ein Datentyp ***std_logic*** definiert.

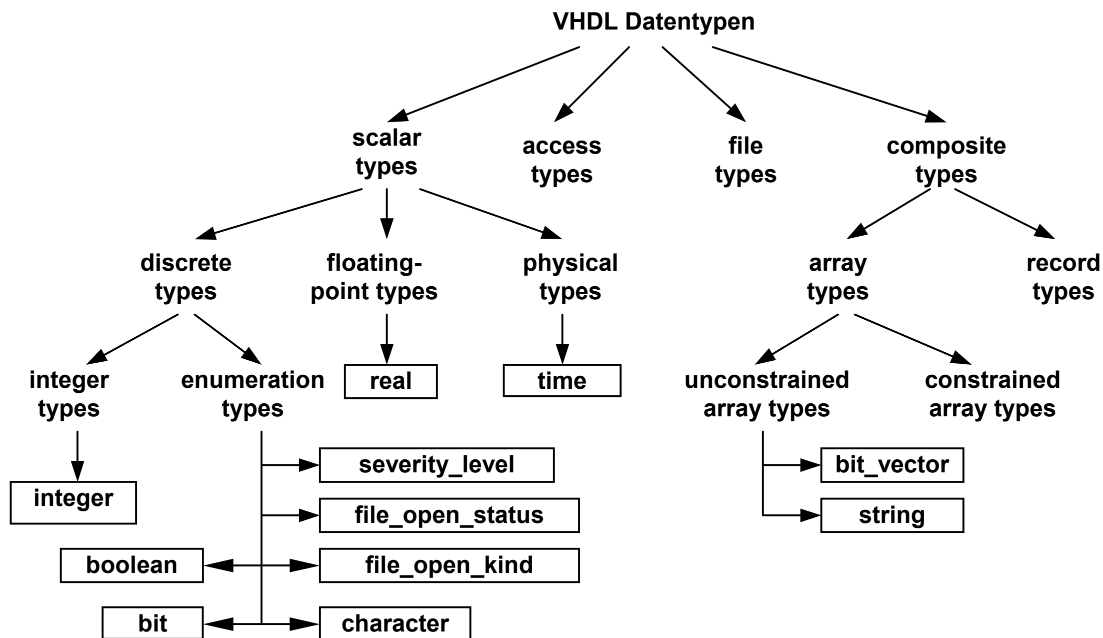
Ein Variablenvereinbarung könnte wie folgt sein:

variable carry: ieee.std_logic_1164.std_logic;

Zur Abkürzung der gesamten Referenz gibt es die ***use*** Klausel

library ieee;
use ieee.std_logic_1164.**all**;   -- **use** ieee.std_logic_1164.std_logic
.....
variable carry: std_logic;
.....

Datentypen Übersicht



Alle eingerahmten Terminale sind im Standard-Package vordefiniert.

VHDL unterscheidet die Datenobjektklassen ***constant***, ***variable***, ***signal*** und ***file***.

Entsprechend vereinbaren wir:

constant Name : *Typ*_Bezeichner;   z.B. integer
variable Name : *Typ*_Bezeichner;   z.B. boolean
signal Name : *Typ*_Bezeichner;   z.B. bit_vector(7 downto 0)

Die **file** Vereinbarung ist etwas anders (später).

Eigene Typvereinbarungen:

Typ_Vereinbarung ::= **type** Bezeichner **is** Typ_Definition**;**

Ein Aufzählungstyp ist z.B.

type MeinTyp **is range** 1 **to** 100;

Eine Variablenvereinbarung von diesem Typ ist z.B.

variable MyVar : MeinTyp;