Glass: A system description language and its environment

Language Reference manual

Syntax definition and the Typing system

M. Seutter (ed.)

Catholic University of Nijmegen

January 20, 1992

Chapter 1

Introduction

M. Seutter, KUN

Glass (**G**eneral **la**nguage for **s**ystem **s**emantics) is a language for describing arbitrary systems, although its present usage is limited to electronic systems. The ideas for this language and its "describing environment" (as opposed to the term "programming environment") are based upon systems semantics [Bou88].

Inspired by the Algol-68 tradition [Wij76], [Lin80] the language is documented in two parts namely the "Language reference manual", (this book, previously called the "Revised Report on the language Glass") and the "Informal introduction to the language Glass", being one of the chapters in [Seu90], the first book of FORFUN.

The first of these parts describes the exact syntax and typing system of the language. The second is is intended to introduce the language Glass in an informal way to those readers that have not seen any Glass description before.

Glass also has two language representations, namely the *reference language* and the *implementation language*. The first is intended as *publication language* and is therefore used troughout this book in all of the examples. However it is not suited for automated interpretation. For this purpose the implementation language may be used.

Using the language the designer may mix analog and digital descriptions as freely as is deemed desirable. To support static semantic checks the language is strongly typed. The language contains macro facilities. Sentences, resulting after macro expansion will be referred to as sentences in the *kernel* language.

We will present the grammar of Glass in chapter 2. Chapter 3 will be devoted to the typing system of Glass. Chapter 4 will present the rules that steer macro expansion. Chapter 5 will give examples of descriptions in current Glass. Several appendices are provided that give "quick reference guides" on the syntax and typing system of Glass.

1.1 Differences with the previous version of Glass

The language differs from its previous version in the following aspects:

- A new typing system for constructing system types has been devised in order to remove the problems arising from the inconsistencies in the typing system of the previous version. Also other basic types may now be declared.

- The index notation has been dropped. To select an element from a list function application will be used. (A list of elements of A with length n is considered as a function with domain $\{0, \ldots, n-1\}$ and codomain A. The backgrounds of this interpretation can be found in [Bou90]). The notation for the slicing operator is also changed.

- Lambda and sigma abstractors are introduced in the language.

- Volumes, system definitions and reference sets have disappeared from the language. A piece of Glass text is merely a list of definitions. Inclusions of other (library) definitions is still possible, now via the include mechanisms of the standard C preprocessor.

- The punctuation in the syntax is improved.

Chapter 2

The grammar of Glass

M. Seutter, KUN

The syntax of Glass will be described by a two level van Wijngaarden grammar [Wij65]. The second level will not be used to express context sensitiveness, but only to create a denser form of the context free grammar. For some of the grammar rules simple examples are given. The context conditions and semantics for the syntactical constructs will be described verbally if necessary.

2.1 Grammatical abbreviations

Syntax

a) NOTION option:
    NOTION;
    .

b) NOTION sequence:
    NOTION, NOTION sequence;
    NOTION.

c) NOTION list:
    NOTION, comma token, NOTION list;
    NOTION.

d) NOTION series:
    NOTION, semicolon token, NOTION series;
    NOTION.

e) nested NOTION:
    left parenthesis token, NOTION, right parenthesis token.

A) NOTION ::
    ABC, NOTION;
    ABC.

B) ABC ::
  a;b;c;d;e;f;g;h;i;j;k;l;m;
  n;o;p;q;r;s;t;u;v;w;x;y;z.

2.2 Glass text

A Glass text is a series of definitions, which we will call Glass *text elements*. Such a Glass text element can be one of the following:

- base type declarations
  These declare the atomic types of the Glass description.
- type naming
  These serve to give names to types.
- atomic system definitions
  These define atomic systems.
- system definition
  This is a user system definition.
- macro definition
  This is a user definition to compose regular structures.

Syntax

a) glass text:
  glass text element series option.

b) glass text element:
  base type declarations;
  type namings;
  atomic system definitions;
  system definition;
  macro definition;
  .

Example

a) **Basetype** B;

Atom
  $\mathit{and} \in B \times B \Rightarrow B$,
  $\mathit{or} \in B \times B \Rightarrow B$,
  $\mathit{not} \in B \Rightarrow B$;

Def
  $\mathit{select} \in B \times B \times B \Rightarrow B$;
  $\mathit{select}\ \langle s,\ a,\ b\rangle = \mathit{or}\ \langle \mathit{and}\ \langle \mathit{not}\ s,\ a\rangle,\ \mathit{and}\ \langle s,\ b\rangle\rangle$;

2.2.1 Basetype declarations

Base type declarations serve to introduce atomic types. In this way more than one type of connection may be introduced like magnetic or optic connections.

Syntax

a) basetype declarations:
    basetype token, name list.

Example

a) **Basetype** B, M, O

2.2.2 Type naming

Type naming can be used to introduce shorthands for types.

Syntax

a) type namings:
    type token, type naming list.

b) type naming:
    name, equals token, type.

Example

a) **Type**
    $port = [B \times B]$

2.2.3 Atomic system definitions

Atomic system definitions serve to introduce primitive components in system descriptions. Semantic functions operating on these descriptions must therefore know the aspects of these primitive components.

Syntax

a) atomic system definitions:
    atomic token, declaration list.

b) declaration:
    name, element of token, type.

Example

a) **Atom**
    $Plus9V \in [B]$,
    $Ground \in [B]$

2.3 Types

The syntax of the new typing system of Glass is best described by means of an affix grammar. In this grammar every nonterminal carries two affixes named x and z, both of which may be "all" or "restr". These affixes are used to impose context conditions on the production rules. The first affix is used to restrict a type to be a type of connections, whereas the second is used to impose that a type may not contain directionality symbols (? and !)

2.3.1 Local type variables

A type may contain *local type variables*. In this way polymorphic types may be introduced.

Syntax

a) type:
    local type specification option,
        type0 (>"all", >"restr").

b) local type specification:
    left brace token, name list,
        right brace token.

Context conditions

a) The local type variables are limited in scope to the surrounding type.

Example

a) $\{A\}A \Rightarrow A$

2.3.2 Function types

Syntax

a) type0 (>"all", >z):
    type1 (>"all", >"restr"), arrow token,
        type0 (>"all", >"restr");
  type0 (>x, >z):
    type1 (>x, >z).

Imposed context conditions

a) Function types may never be used as types for connections, which is imposed by the first affix. The second affix imposes that the left hand side and right hand side of the function type are not partial system types like $?B$.

Examples

a) **Int** $\rightarrow [B \times B]$

2.3.3 System types

A system type describes a circuit interface. There are two kinds of systems, namely the directional and the adirectional ones. For both appropriate syntax may be used. However it is possible to specify directional systems in an adirectional way.

Syntax

a) type1 (>"all", >z):
    adirectional system type;
    directional system type;
  type1 (>x, >z):
    type2 (>x, >z).

b) adirectional system type:
    left square bracket token, type2 (>"restr", >"all"),
      right square bracket token.

c) directional system type:
    type2 (>"restr", >"restr"), fat arrow token,
      type2 (>"restr", >"restr").

Imposed context conditions

b) Only connection types may be specified within the brackets. This forbids types like $[Float]$.

c) Only wire types without directionality specification may be specified at both sides of the fat arrow. In this way types like $(B \rightarrow B) \Rightarrow B$ or $?B \Rightarrow B$ are forbidden.

Examples

b) $[B \times B]$

c) $B \times B \Rightarrow B$
  $B^n \times B^n \Rightarrow B^n$

2.3.4 Product types

Product types serve to specify bundles of connections.

Syntax

a) type2 (>x, >z):
    type3 (>x, >z), cartesian product token,
      type2 (>x, >z);
    type3 (>x, >z).

Imposed context conditions
No extra context conditions are imposed by this piece of grammar, apart from those arising from the propagation of affixes.

Examples

a) $B \times B \times B$
$(B \times B) \times (B \times B)$

2.3.5 Power types

Syntax

a) type3 (>x, >z):
 type4 (>x, >z), to the power of token, expression;
 type4 (>x, >z), kleene star token;
 type4 (>x, >z).

Imposed context conditions
No extra context conditions are imposed by this piece of grammar, apart from those arising from the propagation of affixes.

Examples

a) B^n
$(B \times B)^{n-1}$

b) B^*

2.3.6 Directioned types

In an adirectional system type certain connections may be given a direction. In this way even a directional system type may be specified in an adirectional manner.

Syntax

a) type4 (>x, >”all”):
 question mark token, type5 (>”restr”, >”restr”);
 exclamation mark token, type5 (>”restr”, >”restr”);
type4 (>x, >z):
 type5 (>x, >z).

Imposed context conditions

a) The second affix, both in the heading as well as in the alternatives inhibits that a directionality is imposed on a type that is already directional from itself. The first affix in the alternatives imposes that directionality can only be specified for connection types.

Examples

a) $?B$
$!(B \times B)$

2.3.7 Elementary types

Syntax

a) type5 (>"all", >z):
appset token;
parametrized integer type;
integer type token;
float type token;
bool type token;
string type token;
type5 (>x, >z):
nested type (>x, >z);
empty tuple type token;
name.

b) parametrized integer type:
integer type token, reversed element of token, name.

Examples

a) $(B \Rightarrow B)$
port
Appset
$\langle\rangle$
Int

b) **Int** $\ni s$

2.4 Definitions

User definitions are of two kinds, namely macro and system definitions. The syntax of macro definitions differs from that of kernel definitions in that they may consist of more than one definition part in order to support pattern matching.

Syntax

a) system definition:
define token, system declaration, semicolon token,
definition part.

b) macro definition:
    macro token, system declaration, semicolon token,
        definition part series.

c) definition part:
    name, formal parameter sequence option,
        equals token, rhs.

Examples

a) **Def**
    $select \in B \times B \times B \Rightarrow B;$
    $select\ \langle s, a, b\rangle = or\ \langle and\ \langle not\ s, a\rangle, and\ \langle s, b\rangle\rangle$

b) **Mac**
    $divider\_train \in \mathbf{Int} \rightarrow B \Rightarrow B \rightarrow B \Rightarrow B;$
    $divider\_train\ 0\ A\ clock = clock;$
    $divider\_train\ n\ A\ clock = divider\_train\ (n-1)\ A\ (A\ \langle clock\rangle)$

Context conditions

a) The system name in the definition part must be equal to the name in the system declaration.

b) The names with which the definition parts start must be equal to the name which starts the system declaration.

Semantics

b) If a macro definition contains more than one macro definition part, they are taken together as one definition, consisting of several alternatives. Formal macro parameters may contain patterns in order to select alternatives. During the expansion phase, the alternatives are tried from top to bottom, taking the first alternative whose formals match the actual arguments after their expansion.

2.4.1 Formal parameters

Syntax

a) formal parameter:
    formal parameter1;
    formal parameter1, cons token, formal parameter.

b) formal parameter1:
    name;
    literal;
    empty list token;
    nested formal parameter;
    formal tuple parameter.

c) formal tuple parameter:
brac token, formal parameter list,
ket token.

Semantics

- A name specifies a pattern that always matches.
- A literal specifies a pattern that matches if the value of the literal is the same as the value of the actual argument after its expansion.
- An empty list matches if the actual argument is an empty list after its expansion.
- A pattern 'formal:formal rest' matches if the actual argument is of the form 'actual:actual rest' where 'formal' matches 'actual' and 'formal rest' matches 'actual rest'.
- A pattern '$\langle f_0, \ldots, f_{n-1} \rangle$' matches if the actual argument is of the form '$\langle a_0, \ldots, a_{n-1} \rangle$' where '$f_i$' matches '$a_i$'.

Examples

a) $a : b : z$

b) *"BC547"*
$\langle\rangle$

c) $\langle sel, a, b \rangle$

2.4.2 Right hand sides

Syntax

a) rhs:
expr1, arrow token, rhs, semicolon token, rhs;
rhs1.

b) rhs1:
appset, where clause option;
expr1, where clause option.

c) where clause:
where token, local definition series,
end where token.

Example

c) **where**
$preset = nand\ \langle nload, nclr \rangle;$
$nclr = buf\ nclear;$
$en = and\ \langle enap, enat \rangle$
endwhere

2.4.3 Appsets

Syntax

a) appset:
 left brace token, appset element list option,
 right brace token.

b) appset element:
 expr1, arrow token, appset element,
 semicolon token, appset element;
 appset element1.

c) appset element1:
 expr1;
 appset;
 synonym term.

d) synonym term:
 synonym token, brac token, name tuple list,
 ket token.

Examples

a) $\{\ C\ 10p\ \langle in, x\rangle,$
 $Xtal\ freq\ \langle x, out\rangle$
 $\}$

c) $\star\langle x, y, \langle a, b\rangle\rangle$

2.4.3.1 Local definitions

Syntax

a) local definition:
 glass text element;
 appset;
 name tuple, equals token, expression.

b) name tuple:
 name tuple1, cons token, name tuple;
 name tuple1.

c) name tuple1:
 name;
 brac token, name tuple list option, ket token;
 nested name tuple.

Examples

a) $\langle q, q' \rangle = jkff\,\langle a, b \rangle$

b) $a : \langle x, y \rangle$

c) $(\langle q, q' \rangle)$

2.5 Expressions

2.5.1 Conditionals

The conditional takes the form of the SASL conditional. [Tur79]

Syntax

a) expression:
    expr0, arrow token, expression,
        semicolon token, expression;
    expr0.

Context conditions

a) The type of the expression at the left hand side of the arrow token should be **Bool**. The types of the expression following the arrow token and the expression following the semicolon token should be the same.

Example

a) $a < b \rightarrow and\ \langle x, y \rangle;\ x$

2.5.2 Abstractors

Abstractors serve to introduce nameless abstractions which result for instance from macro expansion. By introducing syntax for them *kernel* Glass may be properly deparsed.

Syntax

a) expr0:
    lambda abstraction;
    sigma abstraction;
    expr1.

b) lambda abstraction:
    lambda token, formal parameter, binding token,
        expr1.

c) sigma abstraction:
    sigma token, formal parameter, binding token,
        appset.

Examples

b) $\lambda\langle x, y\rangle.and\ \langle x, y\rangle$

c) $\sigma\langle a, b\rangle.\{\ R\ 180\ \langle a, b\rangle\}$

2.5.3 Formulas

The syntax for formulas is quite straightforward. Note, however, the associativeness of the diverse operators.

Syntax

a) expr1:
    expr2, cons token, expr1;
    expr2.

b) expr2:
    expr2, equality operator, expr3;
    expr3.

c) expr3:
    expr4, relational operator, expr4;
    expr4.

d) expr4:
    expr4, adding operator, expr5;
    expr5.

e) expr5:
    expr5, multiplicative operator, expr6;
    expr6.

f) expr6:
    expr7, to the power of token, expr6;
    expr7.

g) expr7:
    monadic operator, expr7;
    expr8.

h) equality operator:
    equals token;
    not equals token.

i) relational operator:
    less than token;
    less than or equal token;
    greater than or equal token;
    greater than token.

j) adding operator:
    plus token;
    minus token;
    or token.

k) multiplicative operator:
    times token;
    divides by token;
    div token;
    modulo token;
    and token.

l) monadic operator:
    plus token;
    minus token;
    not token.

Example

c) $a \times (1 + z) < b^2$

2.5.4 Slicing

Slicing serves to select a sub list from a list of elements.

Syntax

a) expr8:
    expr8, slice token, expression, upto token,
        expression;
    expr9.

Example

a) $databus@0 \ldots 7$

2.5.5 System application

This construct is probably the most essential one in Glass .

Syntax

a) expr9:
    expr9, expr10;
    expr10.

Syntax

a) $func\ a\ b\ 180\ \langle x, y \rangle$

2.5.6 Elementary expressions

Syntax

a) expr10:
  name;
  literal;
  tuple;
  nested expression.

b) tuple:
  brac token, expression list option, ket token.

Examples

b) *databus*
  909
  $(9 + z)$

c) $\langle s, a, b\rangle$

2.6 Scope rules

- A local type variable is limited in scope to its surrounding type.
- A definition part limits the scope of its formal parameters.
- Names within right hand sides are local to the right hand side unless they are already bound by formal parameters, system definitions or macro definitions.

2.7 Textual representations

A construct in Glass must be represented in some representation language such as the *reference language*, which is used throughout this report in all of the examples. However this reference language is not suited for automated interpretation. Other representation languages, more suited for this purpose, may be termed *implementation languages*. The implementation language that was used in Esprit project 881 is therefore also specified in this section.

2.7.1 Identifiers

Syntax

a) name:
  identifier.

b) identifier:
tag token.

Example

a) *select'*
divider_train

2.7.2 Literals

Syntax

a) literal:
pragment sequence option, layout, denotation.

b) denotation:
string denotation;
integer denotation;
float denotation;
boolean denotation.

2.7.3 String denotations

Syntax

a) string denotatation:
double quote symbol, quote item sequence option,
double quote symbol.

b) quote item:
quote image;
{ any character not equal to double quote symbol }.

c) quote image:
double quote symbol, double quote symbol.

Examples

a) *"BC 547"*

c) *""*

Context conditions

a) A string denotation has type **String**.

2.7.4 Integer denotations

Syntax

a) integer denotation:
   digit cypher sequence.

Examples

a) 123

Context conditions

a) A integer denotation has type **Int**.

2.7.5 Float denotations

a) float denotation:
   digit cypher sequence, float denotation tail.

b) float denotation tail:
   electrical scale, digit cypher sequence option;
   point symbol, digit cypher sequence, scale;
   exponent.

c) scale:
   electrical scale;
   exponent.

d) exponent:
   exponent symbol, sign option,
      digit cypher sequence.

e) electrical scale:
   {any of the following characters:
      $afpn\mu mkMGTPE$ for the reference language;
      `afpnumkMGTPE` for the implementation language }.

f) sign:
   plus symbol;
   minus symbol.

g) digit cypher:
   {any of the following characters: 0123456789}.

Examples

a) $2k2$
   $1.3e-10$
   $2.2M$
   6

Context conditions

a) A integer denotation has type **int**.

2.7.6 Boolean denotations

Syntax

a) boolean denotation:
 true token;
 false token.

Context conditions

a) A boolean denotation has type **Bool**.

2.7.7 Tokens and symbols

Syntax

a) NOTION token:
 pragment sequence option, layout, NOTION symbol.

b) tag symbol:
 letter, letgits.

c) letgits:
 letgit, letgits;
 .

d) letter:
 {any of the small and capital letters}.

e) letgit:
 {any of the small and capital letters, digits,
 underscore or single quote}.

The following table describes the symbols used in Glass . The second column contains the reference Glass representation. The third contains the implementation language representation as used in the Forfun project. In this latter column, all keyword symbols are shown in small letters, although any mixture of capital and small letters, whose small letter equivalent equals such a keyword is a valid representation of that symbol in the implementation language.

symbol	representation language	implementation language
basetype symbol	**Basetype**	`basetype`
type symbol	**Type**	`type`
atomic symbol	**Atom**	`atom`
define symbol	**Def**	`def`
macro symbol	**Mac**	`mac`

where symbol	**Where**	`where`
end where wymbol	**Endwhere**	`endwhere`
left parenthesis symbol	(	`(`
right parenthesis symbol	)	`)`
empty list symbol	$\langle\rangle$	`[]`
left square bracket symbol	[	`[`
right square bracket symbol	]	`]`
left brace symbol	{	`{`
right brace symbol	}	`}`
brac symbol	$\langle$	`[`
ket symbol	$\rangle$	`]`
comma symbol	,	`,`
semicolon symbol	;	`;`
cartesian product symbol	$\times$	`&`
string type symbol	**String**	`string`
integer type symbol	**Int**	`int`
float type symbol	**Float**	`float`
boolean type symbol	**Bool**	`bool`
appset symbol	**Appset**	`appset`
kleene star symbol	$*$	`*`
synonym symbol	$\star$	`*`
exclamation mark symbol	!	`!`
question mark symbol	?	`?`
equals symbol	$=$	`=`
not equals symbol	$\neq$	`/=`
less than symbol	$<$	`<`
less than or equal symbol	$\leq$	`<=`
greater equal symbol	$\geq$	`>=`
greater than or equal symbol	$>$	`>`
plus symbol	$+$	`+`
minus symbol	$-$	`-`
begin comment symbol	$/*$	`/*`
end comment symbol	$*/$	`*/`
arrow symbol	$\rightarrow$	`->`
fat arrow symbol	$\Rightarrow$	`=>`
to the power of symbol		`^`
times symbol	$\times$	`*`
divides by symbol	$/$	`/`
div symbol	**Div**	`div`
modulo symbol	**Mod**	`mod`
and symbol	$\wedge$	
or symbol	$\vee$	`|`
cons symbol	:	`:`
element of symbol	$\in$	`:-`
reversed element of symbol	$\ni$	`-:`

lambda symbol	λ	`%`
sigma symbol	σ	`!`
slice symbol	@	`@`
upto symbol	...	`...`
double quote symbol	”	`"`
not symbol	$\neg$	`~`
true symbol	**True**	`true`
false symbol	**False**	`false`
exponent symbol	e	`e`
include symbol	*#include*	`#include`

2.7.8 Comments and layout

Syntax

a) pragment:
  pragmat;
  comment.

b) pragmat:
  layout, include symbol, layout,
    string denotation.

c) comment:
  layout, begin comment symbol,
    comment item sequence option,
    end comment symbol.

d) comment item:
  # any character with the exception of the
    end comment symbol #.

e) layout:
  # (possibly empty) sequence of blanks and newlines #.

Example

b) *#include ”mydefs.gl”*

c) */* this is a comment */*

Chapter 3

Grammar for Glass, context-sensitive part

H. Oolman, KUN

3.1 Introduction

The context-free part of the syntax of Glass is described in chapter 2. Here the context-sensitive part of the syntax of Glass is described. Therefore chapter 2 and this chapter taken together have to be seen as the formal definition of the language Glass. Readers who are more interested in a quick reference guide than in this formal definition are referred to appendix B.

The formalisms used in the rest of the text are described first (section 3.2). They are followed by the context-sensitive restrictions coping with the binding and scoping (section 3.4).

3.2 Used formalisms and Glass syntax

3.2.1 The functional formalism

Constraints on the language generated by the context-free grammar of Glass will be specified in this chapter by means of functions. I will explain the notation in short here.

The definition of a function f has the form:

$$\textbf{def } f \in t \textbf{ with } f\ p_0 \cdots\ p_{n-1} = rhs$$

or

$$\textbf{def } f = rhs$$

f is a function of type t; it has n parameters named p_i, and gives rhs as result for the p_i. This can also be used for defining f as name for a type: in that case t is $\mathcal{T}$, the type of types.

As to types, $t_0 \to t_1$ is the type of functions with parameter in t_0 and result in t_1, $t_0 \times \cdots \times t_{n-1}$ is the type of n-tuples with the i-th subpart in t_i, $\mathcal{P}X$ is the powertype of X (i.e. its elements are sets of elements of X), $\mathbb{B}$ are the binary values (or "booleans": 0 corresponds to "false", 1 to "true"), $\mathbb{N}$ the natural numbers.

A where construct introduces local definitions and other local names in scope:

$$expr\ \textbf{where}\ name\text{-}definitions$$

I use a few conventions to reduce the amount of writing, and prevent cluttering:

- if names $(k, m, n, \cdots)$ are introduced to name instantiations, their values are always ≥ 0;
- $sc_0 \oplus \cdots \oplus sc_{n-1}$ stands for a list of elements sc_i in increasing order, interspersed with separators $\oplus$ if $n \geq 2$ (and an empty list if $n = 0$).
- if the kind of a name follows from the context where it is used, then this will not always be explicitly given (e.g: if '$t_1 \Rightarrow t_2$' is used, then $t_1, t_2 \in L(Typ\ \underline{\text{restr}}\ \underline{\text{norestr}}\ \underline{\text{norestr}})$ must hold); if 't^e' is used, then $e \in L\ Expr$ must hold;
- if names are not otherwise bound (by a function parameter, in a where clause, $\exists$, $\forall$, $\{\cdots x \cdots \mid \cdots\}$) they are supposed to be existentially quantified.

Several functions I define have a Glass-argument and deliver a Glass type as result. If in such an argument patterns are used, for clarity it is placed between $[\![$ and $]\!]$, to distinguish it from the normal function text. The thus defined function may consist of several rules. In an application, the rule of which the formal pattern fits the actual argument holds. If a pattern is not unique, the rule will be preceded by an explanation, sometimes referring to nonterminals from the Glass reference grammar (chapter 2).

Useful functions

def
$rt \in (X \to \mathcal{P}X) \to X \to \mathcal{P}X$ **with** $rt\ f\ x = \{x\} \cup \{y \mid z \in f\ x \wedge y \in rt\ f\ z\}$

In words:

'$rt\ f\ x$' gives all the elements of X that x maps to after applying f zero or more times.

def
$rev \in \mathbb{B} \to (X \to \mathcal{P}X) \to (X \to \mathcal{P}X)$**with**
$rev\ b\ f\ x = b?\{y \mid x \in f\ y\}; f\ x$

In words:
applies f or its "inverse" (if b) to an argument.

def
$twoway \in (X \to \mathcal{P}X) \to (X \to \mathcal{P}X)$ **with** $twoway\ f\ x = f\ x \cup rev\ 1\ f\ x$

In words:
if f maps an element of X to others in X, then $twoway\ f\ x$ are the elements x maps to, plus those mapped to x.

3.2.2 Grammar notation

3.2.2.1 Formalism used by the author

In this chapter I will on occasions have to define a grammar. In designing these grammars, it turned out that they were sometimes more elegantly or compactly defined using a two-level formalism. This justifies introducing the following formalism, the more so because I will formally define its semantics.

In order to give the reader an idea of the intention though, I will first give an informal expalnation. Readers not interested in the formal definition of the semantics can than skip the rest of this section.

Expressions of the form $L\ grammar\ expression$ will be often used: L generates a set of pieces of Glass from the grammar expresssion given as argument. As to the grammar expression:
— terminals are underlined, nonterminals aren't;
— nonterminals may have parameters;
— $|$ separates alternatives;
— juxtaposion is for concatenating pieces of Glass text.

The formalism will for example be used to define $Dirs$ (defined second-level because that's more elegant and more compact), SPR and $SOME$ (more elegant), and $Type$ (more compact). They all could have been defined using normal context free grammars and functions on elements of those. Alternatives to the choices in function bodies are sometimes given between square brackets.

A grammar in my formalism consists of a number of instances of $ntdef$ (which defines a nonterminal) as defined below (here '$x*$' means zero or more occurrences of x, '$x+$' one or more — which can be rewritten in the simpler form defined by the grammar below, terminals (i.c. $=, |, \_$) are underlined):

$$\begin{array}{lcl}
ntdef & = & rule+ \\
rule & = & name\ pat* \underline{\equiv} alts \\
alts & = & alt(\underline{|}alt)* \\
alt & = & altelt+ \\
altelt & = & \underline{\underline{terminal}} \mid name\ alt* \\
pat & = & patelt+ \\
patelt & = & \underline{\underline{terminal}} \mid name \\
terminal & = & terminal\, terminal \mid character \mid \epsilon \\
name & = & \ldots \\
character & = & \ldots
\end{array}$$

The language generated by a grammar (this could even be the grammar above) will be defined as the result of a function taking grammar rules of the above form as argument. L is the most important function. The elements of the generated language are trees, not strings, because the intuitive semantics of all variants of pattern matching requires this (see *gs_altelt*, the use of '$[\![ ]\!]$' patterns and '$t = t_1\$\ t_2$' in the rest of this section etc.). Trees are represented by elements of '*L terminal*' (so in '*terminal terminal*' there is in fact an —invisible— tree composition operator).

Written text though (e.g. Glass (types)) is one-dimensional, so of string type. There must be a mapping from flat text (strings) to the tree/trees in the language generated by the grammar, such that the string is the flattened version of the tree (a flattened version of a tree is obtained by making the tree composition operator associative, with ϵ as identity).

How this mapping is exactly defined I do not bother; I just assume that it makes use of e.g. layout, round brackets used for grouping, knowledge about the number of arguments a function should have etc.. In the rest of this section I take (the necessarily flat) texts which should be generated by some grammar, to represent a tree version.

The mapping from string to tree(s) is part of the parsing process, leaving a mapping from this tree to the derivation from some *alt*, about which I don't worry either (e.g. if this is unique or not).

def

$gs\_alt \in L\ alt \rightarrow \mathcal{P}(L\ terminal)$**with**

$gs\_alt(ae_0 \cdots ae_m) = \{s_0 \cdots s_m \mid \forall i \in 0..m.\ s_i \in gs\_altelt\ ae_i\}$

def

$gs\_altelt \in L\ altelt \rightarrow \mathcal{P}(L\ terminal)$**with**

$gs\_altelt(\underline{t}) = \{t\}$

$$\begin{array}{l}
gs_altelt(n\, a_0 \cdots a_{m-1}) \\
\quad = \{y \mid \\
\qquad \exists \langle gac_0, \cdots, gac_{m-1}\rangle \in gs_alt\ a_0 \times \cdots \times gs_alt\ a_{m-1}. \\
\qquad \exists \langle ga_0, \cdots, ga_{n-1}\rangle \in (L\ terminal)^n. \\
\qquad y \in gs_alts(r_l[\underline{ga_0}/v_0] \cdots [\underline{ga_{n-1}}/v_{n-1}]) \\
\qquad \wedge\ match\ l \wedge \forall i < l.\ \neg match\ i
\end{array}$$

$\wedge\forall j \in 0..m-1.\ \{gac_j\} = gs\_alt(p_{j\,l}[\underline{ga_0}/v_0]\cdots[\underline{ga_{n-1}}/v_{n-1}])$
$\wedge\{v_0, \cdots, v_{n-1}\} = names\{p_{0\,l}, \cdots, p_{m-1\,l}\}$
where

$$\left.\begin{array}{llllll} n & p_{0\,0} & \cdots & p_{m-1\,0} & = & r_0 \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ n & p_{0\,k} & \cdots & p_{m-1\,k} & = & r_k \end{array}\right\} = \text{"the definition of } n \text{ as nonterminal"}$$

def
$match \in \mathbb{N} \to \mathbb{B}$**with**
$match\ l = \exists\langle ga_0, \cdots, ga_{q-1}\rangle \in (L\ terminal)^q.$
$\quad\forall j \in 0..m-1.\ \{gac_j\} = gs\_alt(p_{j\,l}[\underline{ga_0}/v_0]\cdots[ga_{q-1}/v_{q-1}])$
$\quad\wedge\{v_0, \cdots, v_{q-1}\} = names\{p_{0\,l}, \cdots, p_{m-1\,l}\}$
def
$names \in \mathcal{P}(L\ pat) \to \mathcal{P}(L\ name)$**with**
$names\ xs =$ "all the names in each of the elements of xs"
}

Here $e[x_0/y_0]\cdots[x_{n-1}/y_{n-1}]$ stands for e with each occurrence of y_i replaced by x_i $(i \in 0..n-1)$.

def
$gs\_alts \in L\ alts \to \mathcal{P}(L\ terminal)$**with**
$gs\_alts(a_0 \mid \cdots \mid a_m) = \bigcup_{i=0}^{m} gs\_alt\ a_i)$
def $L \in L\ alt \to \mathcal{P}(L\ terminal)$**with** $L = gs\_alt$

In this section the mapping from a grammar expression (an *alt*) to the language generated by it is explicit (the function L). It could also have been kept implicit, but for the fact that I found it necessary to precisely write down what I meant by the formalism. Had the formalism been interspersed with some more terminals, replacing for example the instances of the form '*name alt*+' in *altelt* by '*name* $\underline{@_0}$ (*alt* ($\underline{@_1}$ *alt*)+)' and '*alt* = *altelt*+' by '*alt* = *altelt* ($\underline{@_1}$ *altelt*)+', then these could have been given a definition as a function, and thus made L and the like superfluous:

def $\_ \in \cdots$ **with** $\underline{t} = \{t\}$

def
$@_0 \in \cdots$ **with**
$n@_0a =$ "something resembling *gs_altelt*, with only one argument"

def $@_1 \in \cdots$ **with** $a_0@_1a_1 = \{s_0s_1 | s_0 \in a_0, s_1 \in a_1\}$

def $\mid\ \in \cdots$ **with** $a_0 \mid a_1 = a_0 \cup a_1$

The disadvantage of this is that a grammar definition will be more cluttered with symbols.

3.2.3 Short reference grammar for Glass types

The functions to be defined in this chapter normally take a sentence from the Glass language as argument, and several of them have a sentence from the language generated by the nonterminals for types as result. Because of their importance they are introduced here again for convenience.

For use in *bindscope* I use *SimpType*, corresponding to *type* (see section 2) without attributes. This is used because the context-free grammar can not guarantee that *Names* are always used on the correct places: a name bound to a basetype or used as local type variable may not be used in all cases, and a name defined in a typenaming may only be used if the defined type may be used on the same place.

For use in *ty_constr* (and *bindscope*) '*Type pv*' is used. '*Type* $\underline{\text{norestr}}$' corresponds to *type*, '*STyp* $\underline{\text{norestr}}$' to *(a)directional system type*, *TVar* to *name* used as *type* and *Expr* to *expression*.

To express that directions (?, !) are not part of '*Typ* $\underline{\text{restr}}$ $\underline{\text{norestr}}$ *pv*', they are added separately (by means of *Dirs*):

$SimpType = (\underline{\{}TVar\underline{,}\cdots\underline{,}TVar\underline{\}})?\ SimpTyp$

$SimpTyp =$

$\underline{\langle\rangle}\ |$ (type for empty tuple)
$SimpTyp\ \underline{\times}\cdots\underline{\times}\ SimpTyp\ |$
$SimpTyp^{Expr}\ |$
$SimpTyp\underline{^*}$ (for arbitrary length)
$Name\ |$
$\underline{\textbf{Appset}}\ |$
$SimpTyp\ \underline{\rightarrow}\ SimpTyp\ |$
$SimpSTyp\ |$
$\underline{\textbf{Int}}\ \underline{\ni}Name\ |\ \underline{\textbf{Int}}\ |\ \underline{\textbf{Float}}\ |\ \underline{\textbf{String}}\ |\ \underline{\textbf{Bool}}$

$SimpSTyp = SimpTyp\ \underline{\Rightarrow}\ SimpTyp\ |\ \underline{[}Dirs\ SimpTyp\underline{]}$

$Type\ pv = (\underline{\{}TVar\underline{,}\cdots\underline{,}TVar\underline{\}})?\ Typ\ \underline{\text{norestr}}\ \underline{\text{norestr}}\ pv$

$Typ\ c\ e\ pv =$

(c: only for connection-types ($\underline{\text{restr}}$) or not;
e: needs to end with $\langle\rangle$ ($\underline{\text{restr}}$) or not;
pv: only basic, empty and composed types ($\underline{\text{restr}}$) or not)
$\underline{\langle\rangle}\ |$ (type for empty tuple)
$Typ\ c\ \underline{\text{norestr}}\ pv\ \underline{\$}\ Typ\ c\ \underline{\text{restr}}\ pv\ |$
(type for composed tuple: '$t_0 \times \cdots \times t_{n-1}$' is equivalent to
'$t_0\$\ \cdots \$\ t_{n-1}\$\ \langle\rangle$'; '\$' is introduced to represent the case $n = 1$)

(the next four alternatives are excluded if $pv = \underline{\text{restr}}$:)
$Typ\, c\, \underline{\text{norestr}}\, pv \,\underline{\times} \cdots \underline{\times}\, Typ\, c\, \underline{\text{norestr}}\, pv \mid$
$Typ\, c\, \underline{\text{norestr}}\, pv^{Expr} \mid$
$Typ\, c\, \underline{\text{norestr}}\, pv\underline{^*} \mid$ (for arbitrary length)
(the following up to the end are not allowed if $e = \underline{\text{restr}}$:)
$TVar \mid$ (type variable)
$ATypename \mid$ (bound to **Basetype** decl.)
(the following up to the end are not allowed if $c = \underline{\text{restr}}$:)
$\underline{\textbf{Appset}} \mid$
$Typ\, \underline{\text{norestr}}\, \underline{\text{norestr}}\, pv \,\underline{\rightarrow}\, Typ\, \underline{\text{norestr}}\, \underline{\text{norestr}}\, pv \mid$
$STyp\, pv \mid$
$\underline{\textbf{Int}}\, \underline{\ni} Name \mid \underline{\textbf{Int}} \mid \underline{\textbf{Float}} \mid \underline{\textbf{String}} \mid \underline{\textbf{Bool}}$

$STyp\, pv =$
$Typ\, \underline{\text{restr}}\, \underline{\text{norestr}}\, pv \,\underline{\Rightarrow}\, Typ\, \underline{\text{restr}}\, \underline{\text{norestr}}\, pv \mid$
$\underline{[}Dirs(Typ\, \underline{\text{restr}}\, \underline{\text{norestr}}\, pv)\underline{]}$

$$
\begin{array}{ll}
Dirs\, n & = (\underline{?} \mid \underline{!})\, n \mid n \qquad \text{for } n \in L\ Name \\
Dirs\underline{\langle\rangle} & = (\underline{?} \mid \underline{!})\, \underline{\langle\rangle} \mid \underline{\langle\rangle} \\
Dirs(t_1\, \underline{\$}\, t_2) & = (\underline{?} \mid \underline{!})\, (t_1 \underline{\$}\, t_2) \mid t_1 \underline{\$}\, t_2 \mid (Dirs\ t_1) \underline{\$}\, (Dirs\ t_2) \\
Dirs(t^e) & = (\underline{?} \mid \underline{!})\, (t^e) \mid t^e \mid (Dirs\ t)^e \\
Dirs(t\underline{^*}) & = (\underline{?} \mid \underline{!})\, (t\underline{^*}) \mid t\underline{^*} \mid (Dirs\ t)\underline{^*} \\
Dirs(t_0\, \underline{\times} \cdots \underline{\times}\, t_{n-1}) & = (\underline{?} \mid \underline{!})(t_0\, \underline{\times} \cdots \underline{\times}\, t_{n-1}) \mid t_0\, \underline{\times} \cdots \underline{\times}\, t_{n-1} \mid \\
& \quad (Dirs\ t_0)\, \underline{\times} \cdots \underline{\times}\, (Dirs\ t_{n-1})
\end{array}
$$

3.3 Name binding and scoping constraints

This section describes context-sensitive constraints related to declarations of names, like if a name must be bound, to what it may be bound, at most one binding occurrence for one name. A Glass text g satisfies these constraints if $bindscope\ (\lambda n.\text{Undef})\ g$ (to be defined in the following) holds.

def
$bindscope \in (L\ Name \rightarrow (L\ SimpType \cup \{\text{Undef}, \text{BTName}\}))$
$\quad \rightarrow L\ \mathsf{Glass} \rightarrow \mathbb{B}\textbf{with}$
$bindscope\, btns\, [\![elt_0; \cdots ; elt_{m-1}]\!]$
$\quad = \#eltnames = m \wedge$
$\quad nocycle \wedge$
$\quad \forall t \in types\_in\_sys.\ (\forall n \in non\_local\_names\, t.btns'\, n \neq \text{Undef} \wedge$
$\qquad expd\, btns\, bt\_tns\, t \in L(Type\ \underline{\text{norestr}}) \wedge$
$\qquad \#(numnames\, t) = nrofnumnames\, t) \wedge$
$\quad \forall i \in 0..m-1.bindscope\, btns'\, elt_i$
where
def $eltnames \in \mathcal{P}(L\ Name)$**with** $eltnames = \{s | \langle s, t\rangle \in names\_types\}$
def
$names\_types \in \mathcal{P}(L\ Name \times L\ SimpType)$**with**

$$
\begin{array}{l}
names\_types \\
\quad = \{\langle s_j, t_j\rangle \mid \\
\qquad \exists i \in 0..m-1.\ (elt_i = \mathbf{Atom}\, s_0 \in t_0, \cdots, s_{k-1} \in t_{k-1} \vee \\
\qquad\qquad\qquad elt_i = \mathbf{Def}\ s_0 \in t_0;\ desc \vee \\
\qquad\qquad\qquad elt_i = \mathbf{Mac}\ s_0 \in t_0; desc_0; \cdots;\ desc_{k-1} \vee \\
\qquad\qquad\qquad elt_i = \mathbf{Type}\, s_0 = t_0, \cdots, s_{k-1} = t_{k-1} \vee \\
\qquad\qquad\qquad (elt_i = \mathbf{Basetype}\, s_0, \cdots, s_{k-1} \wedge t_j = \mathrm{BTName})) \\
\qquad \}
\end{array}
$$

def

$$
\begin{array}{l}
nocycle \in \mathbb{B}\mathbf{with} \\
nocycle = \forall \langle n, t\rangle \in bt\_tns. \\
\qquad t \neq \mathrm{BTName} \Rightarrow n \notin \{n' \mid\ n' \in names\_in\_type\, t' \wedge \\
\qquad\qquad\qquad\qquad t' \in rt\ (substty\ btns\ bt\_tns)\ t\}
\end{array}
$$

def

$$
\begin{array}{l}
bt\_tns \in \mathcal{P}(L\ Name \times L\ SimpType)\mathbf{with} \\
bt\_tns \\
= \{\langle s_j, t_j\rangle \mid \exists i \in 0..m-1. elt_i = \mathbf{Type}\, s_0 = t_0, \cdots, s_{k-1} = t_{k-1}\} \cup \\
\quad \{\langle s_j, \mathrm{BTName}\rangle \mid \exists i \in 0..m-1. elt_i = \mathbf{Basetype}\, s_0, \cdots, s_{k-1}\}
\end{array}
$$

def

$$
\begin{array}{l}
types\_in\_sys \in \mathcal{P}(L\ SimpType)\mathbf{with} \\
types\_in\_sys = \{t \mid \langle s, t\rangle \in names\_types \wedge t \neq \mathrm{BTName}\}
\end{array}
$$

def

$non\_local\_names \in L(Type\ \underline{\mathrm{norestr}}) \to \mathcal{P}(L\ Name)\mathbf{with}$

$non\_local\_names[\![\{n_0, \cdots, n_{m-1}\}\ t]\!]$

= "all names in t that are not in $\{n_0, \cdots, n_{m-1}\}$
and not bound to an **Int** type or are in e in t^e";

$non\_local\_names\, t$

= "all names in t that are not bound to an **Int** type or in e in t^e
(i.e. not type variables, not bound by '$\ni n$')"

def

$numnames \in L\ SimpType \to \mathcal{P}(L\ Name)\mathbf{with}$

$numnames\, t =$ "all names in t appearing as '$\ni n$' "

def

$nrofnumnames \in L\ SimpType \to \mathbb{N}\mathbf{with}$

$nrofnumnames\, t =$ "the number of '$\ni n$' in t"

def

$btns' \in L\ Name \to (L\ SimpType \cup \{\mathrm{Undef}, \mathrm{BTName}\})$

$btns'\ n = \langle n, t\rangle \in bt\_tns?\ expd\ btns\ bt\_tns\ t;\ btns\ n$

In words:

all defined names must differ, there may not be typenamings defined cyclicly (i.e. in filling in the types for type names in a named type, the name given to it may not be encountered), a name used as a type must be declared as something (as a local type variable, basetype name or in a typenaming), after expanding typenamings in a declared type a grammatically correct type must result (this can not be wholly checked by the parser, because names in types are used for different purposes), and the names bound to **Int** types must differ.

The parameter *btns* gives for each name: if introduced in a typenaming, the type that it is a name for; or 'BTName' in case the name is for a basetype, or else 'Undef'. Names used as types are bound to the nearest enclosing (with inner where clauses enclosing nearer) occurrence of this name as: —a local type variable for this type-declaration, —a basetype or typenaming.

def

$$
\begin{array}{l}
substty \in (L\ Name \rightarrow (L\ SimpType \cup \{\mathrm{Undef}, \mathrm{BTName}\})) \rightarrow \\
\qquad \mathcal{P}(L\ Name \times L\ SimpType) \rightarrow L\ SimpType \rightarrow \mathcal{P}(L\ SimpType)\mathbf{with} \\
substty\ btns\ bt\_tns\ t \\
\quad = \{t'' \mid\ t'' = t[t'/n'] \wedge n' \in non\_local\_names\ t \wedge \\
\qquad\quad ((\langle n', t'\rangle \in bt\_tns \wedge t' \neq \mathrm{BTName}) \vee \\
\qquad\quad (\neg\langle n', t'\rangle \in bt\_tns \wedge btns\ n' = t' \wedge t' \notin \{\mathrm{BTName}, \mathrm{Undef}\}))\} \\
\qquad \}\mathbf{where}\ t[t'/n'] = \text{"}t \text{ with } t' \text{ replacing } n'\text{, after renaming local} \\
\qquad\qquad\qquad\qquad \text{type variables and names bound to } \mathbf{Int} \text{ types and} \\
\qquad\qquad\qquad\qquad \text{adding the local type variables of } t' \text{ to the set for } t\text{"}
\end{array}
$$

def

$$
\begin{array}{l}
expd \in (L\ Name \rightarrow (L\ SimpType \cup \{\mathrm{Undef}, \mathrm{BTName}\})) \rightarrow \\
\qquad \mathcal{P}(L\ Name \times L\ SimpType) \rightarrow L\ SimpType \rightarrow L\ SimpType\mathbf{with} \\
expd\ btns\ bt\_tns\ t \\
\quad = [t' \mid t' \in rt\ (substty\ btns\ bt\_tns)\ t \wedge rt\ (substty\ btns\ bt\_tns)\ t' = \{t'\}]
\end{array}
$$

$$
\begin{array}{l}
bindscope\ btns\ [\![\mathbf{Def}\ f \in t; g\ pc_0 \cdots pc_{k-1} = rhs]\!] \\
\quad = f = g \wedge \forall pc \in formparms.pc \in L\ Name \wedge \\
\qquad \#parnames = nrnames\ formparms \wedge \\
\qquad names(\{pc_0, \cdots, pc_{k-1}\} \setminus formparms) \cap parnames = \emptyset \wedge \\
\qquad bindscope\ btns\ rhs \\
\qquad \mathbf{where} \\
\qquad \mathbf{def}\ parnames \in \mathcal{P}(L\ Name)\mathbf{with}\ parnames = names\ formparms \\
\qquad t_0 \rightarrow \cdots t_{m-1} \rightarrow t' = expd\ btns\ \emptyset\ t \wedge t' \neq t_m \rightarrow t'' \\
\qquad \mathbf{def} \\
\qquad formparms \in \mathcal{P}(L\ Name)\mathbf{with} \\
\qquad formparms = \{pc_0, \cdots, pc_{min(k,m)-1}\}
\end{array}
$$

In words:

formal parameters (those for which there is declared a type t_1 in $t_1 \rightarrow t_2$) of system definitions may only be names, which must differ; names in the system λ-or σ-abstractor (the $m+1$-th parameter, if present) must differ from those.

def

$$
\begin{array}{l}
names \in \mathcal{P}(L\ \mathsf{Glass}) \rightarrow \mathcal{P}(L\ Name)\mathbf{with} \\
names\ x = \text{"the collection of names occurring in the elements of } x\text{"}
\end{array}
$$

def
$nrnames \in \mathcal{P}(L\ \mathsf{Glass}) \rightarrow \mathcal{P}(L\ Name)$**with**
$nrnames\ x =$ "the number of name occurrences in the elements of x"

$$
\begin{array}{l}
bindscope\ btns\ [\![\mathbf{Mac}\ f \in t; \\
\qquad\qquad\qquad \begin{array}{lllll}
g_0 & pc_{0\,0} & \cdots & pc_{0\,k_0-1} & = rhs_0 \\
. & & & & \\
. & & & & \\
. & & & & \\
g_{l-1} & pc_{l-1\,0} & \cdots & pc_{l-1\,k_{l-1}-1} & = rhs_{l-1}
\end{array} \\
\qquad]\!]
\end{array}
$$

$$
\begin{array}{l}
= \forall i \in 0..l-1. \\
\quad f = g_i \wedge \\
\quad \#parnames = nrnames\ formparms \wedge \\
\quad names(\{pc_{i\,0}, \cdots, pc_{i\,k_i-1}\} \setminus formparms) \cap parnames = \emptyset \wedge \\
\quad bindscope\ btns\ rhs \\
\quad \mathbf{where} \\
\quad \mathbf{def}\ parnames \in \mathcal{P}(L\ Name)\mathbf{with}\ parnames = names\ formparms \\
\quad t_0 \rightarrow \cdots t_{m-1} \rightarrow t' = expd\ btns\ \emptyset\ t \wedge t' \neq t_m \rightarrow t'' \\
\quad \mathbf{def}\ formparms \in \mathcal{P}(L\ Name)\mathbf{with} \\
\quad formparms = \{pc_{i\,0}, \cdots, pc_{i\,min(k_i,m)-1}\}
\end{array}
$$

In words:
with the exception of the requirement that formal parameters be names, the same holds for each macro alternative as for a system definition.

$bindscope\ btns\ rhs$
= "for 'ae**Where** $ld_0; \cdots; ld_m$ **Endwhere**' in rhs,
$bindscope\ btns\ \{ld_i \mid ld_i \neq ns = expr\}$ must hold"

In words:
for all local '*glass text element*'s the same must hold as on the top level (with bindings in the '*glass text element*'s in the where clause overruling those outside).

3.4 Typing constraints for Glass

For simplicity it is assumed that typenamings have been expanded (see *expd* in the previous section), and that all basetype names differ (since the same basetype name declared on different where-levels does not stand for the same, and assuming these names to have been made unique is the easiest way to make the distinction).

The constraints can be defined with two alternative definitions for *val* (of importance only if indexing/slicing expressions are in the kernel language) and *emptyequiv*:

def $val \in L$ Glass $\rightarrow \mathbb{N}$**with** $val\ e =$ "the numeric value of e"
def $emptyequiv \in \mathcal{P}(L(Typ\,\underline{\text{norestr}}\ \underline{\text{norestr}}\ \underline{\text{restr}}))$**with** $emptyequiv = \{\langle\rangle\}$

or:

def $val \in L$ Glass $\rightarrow \mathbb{N}$**with** $val\ e = 0$
def
$emptyequiv \in \mathcal{P}(L(Typ\,\underline{\text{norestr}}\ \underline{\text{norestr}}\ \underline{\text{restr}}))$**with**
$emptyequiv = \{\ t\$ \cdots t\$\langle\rangle\ |$
$\quad t \in L(Typ\,\underline{\text{norestr}}\ \underline{\text{norestr}}\ \underline{\text{restr}})$ with $\#(t\$) \geq 1\}$

(≥ 1: otherwise all types ending in $\langle\rangle$ would be equivalent; that is too lenient)

[alternative: $\{t' \mid t' \in L(SOME\ \underline{t}) \wedge t \in L(Typ\,\underline{\text{norestr}}\ \underline{\text{norestr}}\ \underline{\text{restr}})\}$
$SOME\ t = t\underline{\$}\ \underline{\langle\rangle} \mid t\underline{\$}\ SOME\ t$]

Explanation:
there are two forms of the type constraints: one severe form for Glass texts without (user-defined or built-in) macros (corresponds to the former definitions of *val* and *emptyequiv*), and a less severe form for Glass texts with macros (latter definitions). The severe form checks that connected interfaces have exactly the same size, and that indices are in range, the less severe form doesn't.

Definitions of some useful types:

def $envtype \in \mathcal{T}$**with** $envtype = L\ Name \rightarrow L(Type\,\underline{\text{norestr}})$
def $conntype \in \mathcal{T}$**with** $conntype = L(Typ\,\underline{\text{restr}}\ \underline{\text{norestr}}\ \underline{\text{restr}})$
def $typstype \in \mathcal{T}$**with** $typstype = \mathcal{P}(L(Typ\,\underline{\text{norestr}}\ \underline{\text{norestr}}\ \underline{\text{restr}}))$

In words:
envtype is the type for a function that gives the Glass type (in the current scope) of names. Such a function will be called a type environment.
conntype is the type that 'connections' between differents parts of a circuit must have, in their simplest form: such connections may have a base-type, a type for an empty connection, or a type for a connection composed of subparts.
typstype is the type for sets of Glass types as will be delivered by various typing functions (excludes $t_0 \times \cdots \times t_{n-1}$, t^e, t^*).

3.4.1 Typing constraints on a Glass text

A context-free syntax correct Glass text fulfils the type constraints if the following conditions hold:

$ty\_constr\ g\ (\lambda n.\text{Undef})$	for a Glass text g ($\lambda n.$Undef never applied)
$\mathbf{Int} \in types\_ae\ 0\ SB\ (\lambda n.\mathbf{Int})\ e$	for e in t^e (SB arbitrary)

In words:

e in t^e must be an integer, names in e assumedly are integers too. A Glass text g is subject to certain typing constraints.

def

$ty\_constr \in L$ Glass $\rightarrow envtype \rightarrow \mathbb{B}$**with**

$ty\_constr\ [\![md_0; \cdots ; md_{m-1}]\!]\ env$

$= \forall i \in 0..m-1.$

$(((\mathbf{Def}\ f \in t; f\ desc_0) = md_i \wedge n = 1)\vee$

$((\mathbf{Mac}\ f \in t; f\ desc_0; \cdots ; f\ desc_{n-1}) = md_i)$

$) \Rightarrow \forall j \in 0..n-1. t \in types\ env'\ desc_j$

where $env' = update\ env\ \{md_0, \ldots, md_{m-1}\}$

In words:

the type declared for a system/macro definition must be one of those that is valid for the system definition/macro alternative.

def

$update \in envtype \rightarrow \mathcal{P}(L$ Glass$) \rightarrow envtype$**with**

$update\ env\ mds\ n$

$= \exists md \in mds.\ \mathbf{Atom}\ n \in t = md \vee \mathbf{Def}\ n \in t; d = md \vee$

$\mathbf{Mac}\ n \in t; ds = md?\ t;\ env\ n$

In words:

updates the function giving types for names with the types declared for atoms, system- and macro-definitions (the names of atoms, system- and macro-definitions in mds must differ).

3.4.2 Types for system- and macro-definitions

$types$ will be defined piece by piece (for each possible Glass construct) in the rest of this chapter, interspersed with explanations and definitions of additional functions.

def

$types \in envtype \rightarrow L$ Glass $\rightarrow \mathcal{P}(L(Type\ \underline{\text{norestr}}))$**with**

$types\ TE\ desc$

$= \{t \mid \exists tp \in allpowers\ t.$

$(tp \in tr \vee$

$(tp = \{v_0, \cdots, v_{n-1}\}\ tp' \wedge$

$\forall \langle t_0, \cdots, t_{n-1}\rangle \in (L\ conntype)^n. tp'[t_0/v_0] \cdots [t_{n-1}/v_{n-1}] \in tr$

$)$

$)$

where def $tr \in typstype$**with** $tr = types\_d\ TE\ desc$

$\}$

In words:

types TE d gives all valid severe types for d given that the names in scope have certain types (see TE). Local type variables may be used when they can stand for all connection types (i.e. elements of '*L conntype*').

The type tp in the definition is either a type derivable for *desc*, or one containing type variables, if by filling in any value for the type variables a type derivable for *desc* is obtained.

It is not demanded that local type variables can stand for all *arbitrary* types, because then the use of them would not be permitted in many interesting cases (since not all types are allowed as subpart of another; e.g. one may not use **Int** as subpart of a system type).

Correspondingly, when using a name declared with a type containing local type variables, the local type variables may only be replaced by a connection type (see *types_ae* applied to a name).

def

$$
\begin{array}{l}
allpowers \in L(Type\,\underline{\mathrm{norestr}}) \rightarrow \mathcal{P}(L(Type\,\underline{\mathrm{norestr}}))\mathbf{with}\\
allpowers\,t\\
\quad = \{\{v_0, \cdots, v_{n-1}\}\,t'_1 \mid t = \{v_0, \cdots, v_{n-1}\}\,t_1 \wedge t'_1 \in allpowers\,t_1\} \cup\\
\quad\quad \{t \mid t \in L\,TVar \cup L\,ATypename \cup\\
\quad\quad\quad\quad \{\mathbf{Int}, \mathbf{Int} \ni n, \mathbf{Float}, \mathbf{Bool}, \mathbf{String}, \mathbf{Appset}\}\} \cup\\
\quad\quad \{t' \mid t = \langle\rangle \wedge t' \in emptyequiv\} \cup\\
\quad\quad \{t' \mid t = t_0 \times \cdots \times t_m \wedge t' \in allpowers[\![t_0\$ \cdots \$\, t_m\$\, \langle\rangle]\!]\} \cup\\
\quad\quad \{t'_0\$ \cdots t'_0\$\, \langle\rangle \text{ with one or more '}t'_0\$\text{'s} \mid\\
\quad\quad\quad\quad\quad (t = t_0^e \vee t = t_0^*) \wedge t'_0 \in allpowers\,t_0\} \cup\\
\quad\quad \{t' \mid t \in L(Dirs(Typ\,\underline{\mathrm{restr}}\,\underline{\mathrm{norestr}}\,\underline{\mathrm{norestr}})) \wedge\\
\quad\quad\quad t' \in overDCt\,allpowers\,t\} \cup\\
\quad\quad \{t' \mid t' \in oversubFSt\,0\,allpowers\,t\}
\end{array}
$$

In words:

in the severe typing restrictions (assumed to be imposed on a Glass text which does not contain types t^*, t^e or $t_1 \rightarrow t_2$) $allpowers\,t = \{t\}$. In the less severe check, $allpowers\,t$ gives all types obtained by replacing the types $\langle\rangle$, t^* and t^e by $t'\$ \cdots \$t'\$\langle\rangle$. The types in $allpowers\,t$ are the less severe types that correspond to the broadened version of the severe syntactic category corresponding to t.

def

$$
\begin{array}{l}
oversubFSt \in \mathbb{B} \rightarrow (L(Dirs(Typ\,\underline{\mathrm{restr}}\,\underline{\mathrm{norestr}}\,\underline{\mathrm{norestr}}))\\
\quad\quad\quad\quad\quad\quad\quad \rightarrow \mathcal{P}(L(Dirs(Typ\,\underline{\mathrm{restr}}\,\underline{\mathrm{norestr}}\,\underline{\mathrm{norestr}}))))\\
\quad\quad\quad\quad\quad \rightarrow L(Type\,\underline{\mathrm{norestr}}) \rightarrow \mathcal{P}(L(Type\,\underline{\mathrm{norestr}}))\mathbf{with}\\
oversubFSt\,sw\,f\,t\\
\quad = \{t'_1 \rightarrow t'_2 \mid\\
\quad\quad t = (t_1 \rightarrow t_2) \wedge ((sw \wedge t_1 \in f\,t'_1) \vee (\neg sw \wedge t'_1 \in f\,t_1)) \wedge t'_2 \in f\,t_2\} \cup\\
\quad\quad \{[t'_1] \mid t = [t_1] \wedge t'_1 \in f\,t_1\} \cup\\
\quad\quad \{t'_1 \Rightarrow t'_2 \mid t = (t_1 \Rightarrow t_2) \wedge t'_1 \in f\,t_1 \wedge t'_2 \in f\,t_2\}
\end{array}
$$

In words:

oversubFSt spreads some function f on a type over all subparts of a function- or system- type. It is used with first parameter 1 when an (asymmetrical) f must be applied reversely to the type of a function argument.

def

$$
\begin{aligned}
&overDCt \in \mathbb{B} \rightarrow (L(Dirs(Typ\,\underline{\text{restr}}\,\underline{\text{norestr}}\,\underline{\text{norestr}}))\\
&\qquad\qquad \rightarrow \mathcal{P}(L(Dirs(Typ\,\underline{\text{restr}}\,\underline{\text{norestr}}\,\underline{\text{norestr}}))))\\
&\qquad \rightarrow L(Dirs(Typ\,\underline{\text{restr}}\,\underline{\text{norestr}}\,\underline{\text{norestr}}))\\
&\qquad \rightarrow \mathcal{P}(L(Dirs(Typ\,\underline{\text{restr}}\,\underline{\text{norestr}}\,\underline{\text{norestr}}))))\textbf{with}\\
&overDCt\ f\ t\\
&\quad = \{t_1'\$\,t_2' \mid t = (t_1\$\,t_2) \wedge t_1' \in f\ t_1 \wedge t_2' \in f\ t_2\} \cup\\
&\qquad \{x\,t_1' \mid t = x\,t_1 \wedge t_1' \in f\ t_1 \wedge x \in \{?,!\}\}
\end{aligned}
$$

In words:

oversubDCt spreads a function over direction-annotated and composed types.

3.4.3 Types for descriptions with(out) parameters

def

$$
\begin{aligned}
&types\_d \in envtype \rightarrow L\ \mathsf{Glass} \rightarrow typstype\textbf{with}\\
&types\_d\ TE\ [\![fp\,rest]\!]\\
&\quad = \{t \rightarrow t' \mid \exists\langle t_{n_0}, \cdots, t_{n_{k-1}}\rangle \in (L(Typ\,\underline{\text{norestr}}\,\underline{\text{norestr}}\,\underline{\text{restr}}))^k.\\
&\qquad\qquad t \in types\_ae\ 1\ TE'\ TE'\ fp \wedge t' \in types\_d\ TE'\ rest\\
&\qquad\qquad \textbf{where}\\
&\qquad\qquad \textbf{def}\\
&\qquad\qquad TE' \in envtype\textbf{with}\\
&\qquad\qquad TE'\ n = n \in \{n_0, \cdots, n_{k-1}\}?\ t_n;\ TE\ n\\
&\qquad\qquad \{n_0, \cdots, n_{k-1}\} = names\{fp\}\\
&\qquad \}\cup\\
&\qquad types\_s\ 1\ TE\ [\![fp\,rest]\!] \cup add\_dirs\ (types\_s\ 0\ TE\ [\![fp\,rest]\!]);\\
&types\_d\ TE\ [\![= rhs]\!]\\
&\quad = types\_ae\ 0\ TE\ rhs \cup types\_d\ TE\ [\![x = rhs\ x]\!] \text{ with } x \text{ a new name}
\end{aligned}
$$

In words:

types_d gives the types of abstractions, including η-conversion. If a parameter is present, a function or system type is in the result set.

There may be types in $types\_s\ 1\ TE\ d \cap add\_dirs\ (types\_s\ 0\ TE\ d)$, which implies that the description d may describe two different systems; these are not excluded in *types_d*, because that would exclude many descriptions for which both interpretations describe the same system after all. In the implementation of the context-sensitive checks though, a warning for the intersection not being empty can be given.

The usage of add_dirs expresses that direction information may be added in the type of a adirectional description (contrarily, it may not be omitted in the type for a unidirectional description).

def
$add\_dirs \in \mathcal{P}(L(STyp\,\underline{\text{restr}})) \to \mathcal{P}(L(STyp\,\underline{\text{restr}}))$**with**
$add\_dirs\; ts$
$\quad = \{t \mid (t \in L(\underline{[}Dirs\;\underline{t'}\underline{]}) \vee$
$\qquad (t = t_1 \Rightarrow t_2 \wedge t' = (t_1\$\, t_2\$\; emp) \wedge emp \in emptyequiv)$
$\qquad) \wedge$
$\qquad [t'] \in ts\,\}$

In words: add_dirs adds directions to a system type.

def
$types\_s\; \in \mathbb{B} \to envtype \to L$ Glass $\to \mathcal{P}(L(STyp\,\underline{\text{restr}}))$**with**
$types\_s\; 1\; TE\; [\![fp = rhs]\!]$
$\quad = \{ts \mid \exists \langle t_{n_0}, \cdots, t_{n_{k-1}} \rangle \in (L\; conntype)^k . ts \in f\; TE'$
$\qquad$ **where**
$\qquad$ **def**
$\qquad f \in envtype \to \mathcal{P}(L(STyp\,\underline{\text{restr}}))$**with**
$\qquad f\; TEc$
$\qquad\quad = \{ts \mid\; t \in types\_ae\; 1\; TE\; TEc\; fp \cap L\; conntype \wedge$
$\qquad\qquad t' \in types\_ae\; 0\; TE\; TEc\; rhs \cap L\; conntype \wedge$
$\qquad\qquad (ts = (t \Rightarrow t') \vee$
$\qquad\qquad (ts \in L(\underline{[}(SPR\;\underline{?}\;\underline{t})\underline{\$}\;(SPR\;\underline{!}\;\underline{t'})\underline{\$}\;\underline{emp}\underline{]}) \wedge$
$\qquad\qquad emp \in emptyequiv$
$\qquad\qquad)$
$\qquad\qquad)\,\}$
$\qquad$ **def** $TE' \in envtype$**with** $TE'\; n = n \in \{n_0, \cdots, n_{k-1}\} ?\; t_n;\; TE\; n$
$\qquad \{n_0, \cdots, n_{k-1}\} = local\_connames\; TE\; [\![fp = rhs]\!]$
$\qquad \};$
$types\_s\; 0\; TE\; [\![fp = rhs]\!]$
$\quad = \{ts \mid \exists \langle t_{n_0}, \cdots, t_{n_{k-1}} \rangle \in (L\; conntype)^k . ts \in f\; TE'$
$\qquad$ **where**
$\qquad$ **def**
$\qquad f \in envtype \to \mathcal{P}(L(STyp\,\underline{\text{restr}}))$**with**
$\qquad f\; TEc = \{[t] \mid$
$\qquad\qquad$ **Appset** $\in types\_ae\; 0\; TE\; TEc\; rhs \wedge$
$\qquad\qquad t \in types\_ae\; 1\; TE\; TEc\; fp \cap L\; conntype$
$\qquad\qquad \}$
$\qquad$ **def** $TE' \in envtype$**with** $TE'\; n = n \in \{n_0, \cdots, n_{k-1}\} ?\; t_n;\; TE\; n$
$\qquad \{n_0, \cdots, n_{k-1}\} = local\_connames\; TE\; [\![fp = rhs]\!]$
$\qquad \};$
$types\_s\; da\; TE\; d = \emptyset$

In words:

types_s gives the types of system abstractions, which may be interpreted either unidirectionally (first parameter 1) or adirectionally (first parameter 0).

def
$local\_connames \in envtype \to L\ \mathsf{Glass} \to \mathcal{P}(L\ Name)$**with**
$local\_connames\ SB\ x$
$= $"all names n in x (excepting local system/macro definitions and λ/σ-abstractions) which are not visibly bound on an outer level, that is, for which $SB\,n \neq$ Undef"

$SPR\,d\,(t_1\underline{\$}\,t_2) = d\,(t_1\underline{\$}\,t_2)\mid(SPR\,d\,t_1)\underline{\$}\,(SPR\,d\,t_2)$
$SPR\,d\,y = d\,y$

In words:
a direction for a composed type is equivalent to that direction spread to the subtypes.

[Some alternatives for parts of the last few definitions (for definitions of the used functions see later on):

- for ts in $types\_s$ 1: $t \Rightarrow t' \in rt\ enlarge\ ts$
- for SPR: $(ts = [t_1\$t_2\$emp] \wedge t_1 \in rt\ sprdirs\ (?t) \wedge t_2 \in rt\ sprdirs\ (!\,t'))$ $\wedge emp \in emptyequiv$
- for add_dirs: $add\_dirs\ ts = \{t' \mid t \in rt\ enlarge\ t' \wedge t \in ts\}$

These are not used because they do not make clear very specifically what is meant, or are less elegant.]

3.4.4 Types for formal parameters and right hand sides

In the definition of *types_ae*, which now follows, SB gives types for names visibly bound on an outer level (names of systems/macros, formal parameters of the current definition, needed for the computing the types of nameless system abstractions and local declarations), TE for all names; *lhs* denotes if the expression is a formal (connection) parameter or not.

3.4.4.1 Types for where clauses

def
$types\_ae \in \mathbb{B} \to envtype \to envtype \to L\ \mathsf{Glass} \to typstype$**with**
$types\_ae\ lhs\ SB\ TE\ [\![e\ \textbf{Where}\ ldd_0; \cdots; ldd_m\ \textbf{Endwhere}]\!]$
$= \{t \mid\ t \in types\_ae\ lhs\ SB'\ TE'\ e \wedge$
$\forall on \in outputnamings.\textbf{Appset} \in types\_ae\ lhs\ SB'\ TE'\ on \wedge$
$ty\_constr\ elts\ SB$

$\}$ **where**
def
$outputnamings \in \mathcal{P}(L\ \mathsf{Glass})$**with**
$outputnamings = \{ldd_i \mid (ns = e) = ldd_i\}$
def
$elts \in \mathcal{P}(L\ \mathsf{Glass})$**with** $elts = \{ldd_i \mid ldd_i \in L\ glass\ text\ element\}$
def $SB' \in envtype$**with** $SB' = update\ SB\ \{ldd_0, \ldots, ldd_m\}$
def $TE' \in envtype$**with** $TE' = update\ TE\ \{ldd_0, \ldots, ldd_m\}$

In words:
types for e**Where** ... **Endwhere** are those for e (with a connection name in e of the same type as an instance of it in the outputnamings); local definitions should satisfy the typing restrictions (scope of these: the whole expression).

For an outputnaming:

$types\_ae\ lhs\ SB\ TE\ [\![ns = e]\!]$
$= \{$**Appset** $\mid$
$\exists t \in L\ conntype.t \in types\_ae\ lhs\ SB\ TE\ ns \cap types\_ae\ lhs\ SB\ TE\ e\}$

In words:
both sides of an outputnaming should have the same type. This must be a *connection* type: an outputnaming is not meant for things like '$n = 4$'.

3.4.4.2 Types for system abstractions

$types\_ae\ lhs\ SB\ TE\ [\![\lambda fp.e]\!]$
$= omitdirs(types\_s\ 1\ SB\ [\![fp = e]\!])$
where
def
$omitdirs \in L(STyp\ \underline{\mathrm{restr}}) \to L(STyp\ \underline{\mathrm{restr}})$**with**
$omitdirs\ ts = \{t' \mid (t' = t \vee$
$(t \in L(Dirs\ conntype) \wedge t' \in forgetdir\ t)) \wedge$
$t \in ts)\}$
$types\_ae\ lhs\ SB\ TE\ [\![\sigma fp.as]\!] = types\_s\ 0\ SB\ [\![fp = as]\!]$

In words:
λ/σ-abstractions get a type as if they were nameless systems, with the directions as can be deduced from the syntactic form (possibly with some directions omitted).

3.4.4.3 Types for appsets

$types\_ae\ lhs\ SB\ TE\ [\![\{a_0, \cdots, a_{m-1}\}]\!]$
$= \{$**Appset** $\mid \forall i \in 0..m-1.$**Appset** $\in types\_ae\ lhs\ SB\ TE\ a_i\}$

In words:

an appset gets type **Appset**; its constituent parts should have this type also.

$$
\begin{aligned}
&types\_ae\ lhs\ SB\ TE\ [\![\star\langle ns_0, \cdots, ns_m\rangle]\!] \\
&\quad = \{\mathbf{Appset} \mid \exists t \in L\ conntype.\forall i \in 0..m.t \in types\_ae\ lhs\ SB\ TE\ ns_i\}
\end{aligned}
$$

In words:

a synonym term gets type **Appset**; its subparts must be connections all of the same type.

3.4.4.4 Types for names

If n is a *name*:

$$
\begin{aligned}
&types\_ae\ lhs\ SB\ TE\ n \\
&\quad = \{t \mid \langle \exists \langle t_0, \cdots, t_{k-1}\rangle \in (L\ conntype)^k. \\
&\qquad\qquad t' \in allpowers(tl[t_0/v_0] \cdots [t_{k-1}/v_{k-1}]) \wedge \\
&\qquad\qquad t \in rt(rev\ lhs\ enlarge)t' \\
&\qquad\qquad \mathbf{where} \\
&\qquad\qquad \langle \{v_0, \cdots, v_{k-1}\}, tl\rangle = split\ (TE\ n) \\
&\qquad\qquad \mathbf{def} \\
&\qquad\qquad split \in L\ (Type\ \underline{\text{norestr}}) \\
&\qquad\qquad\qquad \rightarrow \mathcal{P}(L\ TVar) \times L\ (Typ\ \underline{\text{norestr}}\ \underline{\text{norestr}}\ \underline{\text{norestr}})\mathbf{with} \\
&\qquad\qquad split[\![\{v_0, \cdots, v_{k-1}\}tl]\!] = \langle \{v_0, \cdots, v_{k-1}\}, tl\rangle; \\
&\qquad\qquad split\ tl = \langle \emptyset, tl\rangle\ \}
\end{aligned}
$$

In words:

a name has the type declared for it in the environment, where types $\langle\rangle$, t^* and t^e are replaced by $t'\$ \cdots \$t'\$\langle\rangle$, local type variables may be replaced by any connection type (see also the treatment of local type variables in *types*) and the type may be 'enlarged'.

$$
\begin{aligned}
&\mathbf{def} \\
&enlarge \in L(Typ\ \underline{\text{norestr}}\ \underline{\text{norestr}}\ \underline{\text{restr}}) \rightarrow typstype\mathbf{with} \\
&enlarge\ t \\
&\quad = \{t\} \cup \\
&\qquad \{t' \mid t \in L(Dirs\ conntype) \wedge t' \in forgetdir\ t\} \cup \\
&\qquad oversubFSt\ 1\ enlarge\ t \\
&\qquad \{t' \mid t \in L(Dirs\ conntype) \wedge t' \in twoway\ sprdirs\ t\} \cup \\
&\qquad \{t' \mid t \in L(STyp\ \underline{\text{restr}}) \wedge t' \in twoway\ diradir\ t\} \cup \\
&\qquad \{t' \mid t' \in twoway\ forgetnumname\ t\}
\end{aligned}
$$

In words:

something with type t may also be considered to have one of the types obtained by forgetting about the directions in a system type, or have a type using equivalents: spreading/joining directions over subparts, unidirectional and adirectional system types, **Int** and **Int** $\ni n$. All this may be done to subparts of the type too (with inversed operation performed on types for formal parameters).

If $t' \in rt\ enlarge\ t$, t' is called an enlargement of t, and t a reverse enlargement of t'.

def
$sprdirs \in L(Dirs\ conntype) \rightarrow \mathcal{P}(L(Dirs\ conntype))$**with**
$sprdirs\ t = \{t\} \cup \{d\, t_1\$\, d\, t_2 \mid t = d\,(t_1\$\, t_2) \wedge d \in \{?, !\}\} \cup overDCt\ sprdirs\ t$

In words:
a direction (? or !) for a composed type means the same as each of its parts having this direction.

def
$forgetdir \in L(Dirs\ conntype) \rightarrow \mathcal{P}(L(Dirs\ conntype))$**with**
$forgetdir\ t = \{t\} \cup \{t' \mid t = x\, t'\} \cup overDCt\ forgetdir\ t$

In words:
a system with directions in its type may be used as if (some of the) directions had not been there.

def
$diradir \in L(STyp\ \underline{\mathrm{restr}}) \rightarrow \mathcal{P}(L(STyp\ \underline{\mathrm{restr}}))$**with**
$diradir\ t = \{[?t_1\ \$\ !t_2\ \$\ emp] \mid t = t_1 \Rightarrow t_2 \wedge emp \in emptyequiv\}$

In words:
unidirectional systems are equivalent to systems in adirectional form of a certain type.

def
$forgetnumname \in L(Typ\ \underline{\mathrm{norestr}}\ \underline{\mathrm{norestr}}\ \underline{\mathrm{restr}})$
$\rightarrow \mathcal{P}(L(Typ\ \underline{\mathrm{norestr}}\ \underline{\mathrm{norestr}}\ \underline{\mathrm{restr}}))$**with**
$forgetnumname\ t = \{\mathbf{Int} \mid t = \mathbf{Int} \ni n\}$

In words:
Int and $\mathbf{Int} \ni n$ are treated as equivalent.

3.4.4.5 Types for composed expressions

$$
\begin{aligned}
&types\_ae\ lhs\ SB\ TE\ [\![\langle ae_0, \cdots, ae_{m-1}\rangle]\!] \\
&\quad = \{tae_0\$\ \cdots tae_{m-1}\$\ e \mid \\
&\qquad \forall i \in 0..m-1. tae_i \in types\_ae\ lhs\ SB\ TE\ ae_i \wedge \\
&\qquad e \in rt(rev\ lhs\ enlarge) emp \wedge\ emp \in emptyequiv\} \\
&types\_ae\ lhs\ SB\ TE\ [\![a : b]\!] \\
&\quad = \{ta\$\, tb \mid\ ta \in types\_ae\ lhs\ SB\ TE\ a \wedge \\
&\qquad\qquad tb \in types\_ae\ lhs\ SB\ TE\ b \cap L(Typ\ \underline{\mathrm{norestr}}\ \underline{\mathrm{restr}}\ \underline{\mathrm{restr}})\}
\end{aligned}
$$

In words:

composed expressions get a composed type, built form the types for the subparts.

The type $ta\$tb$ must be of the form $t_0\$\cdots\$t_{n-1}\$\langle\rangle$. This is ensured if tb has also such a form; this requirement is expressed by the restriction that tb be in $L(Typ\,\underline{\text{norestr}}\,\underline{\text{restr}}\,\underline{\text{restr}})$ (see the second parameter).

3.4.4.6 Types for juxtaposition and slicing

$$
\begin{aligned}
&types\_ae\; lhs\; SB\; TE\; [\![fa]\!]\\
&\quad = \{t \mid\\
&\qquad ((t_1 \rightarrow t) \in types\_ae\; lhs\; SB\; TE\; f \wedge t_1 \in types\_ae\; lhs\; SB\; TE\; a)\vee\\
&\qquad ((t\$\;\cdots t\$\;\langle\rangle) \in types\_ae\; lhs\; SB\; TE\; f\; (\text{with } \#(t\$) \geq a'+1) \wedge a' \geq 0 \wedge\\
&\qquad\quad \mathbf{Int} \in types\_ae\; lhs\; SB\; TE\; a\; \mathbf{where}\; a' = val\; a)\vee\\
&\qquad ((t_1 \Rightarrow t) \in types\_ae\; lhs\; SB\; TE\; f \wedge t_1 \in types\_ae\; lhs\; SB\; TE\; a)\vee\\
&\qquad ([t_1] \in types\_ae\; lhs\; SB\; TE\; f \wedge t_1 \in types\_ae\; lhs\; SB\; TE\; a \wedge t = \mathbf{Appset})\\
&\qquad \}
\end{aligned}
$$

In words:

juxtaposition is used for supplying actual parameters, indexing, connecting outputs to inputs of a unidirectional system, connecting something to an adirectional system.

$$
\begin{aligned}
&types\_ae\; lhs\; SB\; TE\; [\![a@l\ldots h]\!]\\
&\quad = \{t' \mid\\
&\qquad t' \in rt\; enlarge\; (t\$\;\cdots t\$\; emp)\; (\text{with } \#(t\$) = s)\wedge\\
&\qquad emp \in emptyequiv \wedge emp = t\$\;\cdots t\$\;\langle\rangle\wedge\\
&\qquad (t\$\cdots t\$\langle\rangle) \in types\_ae\; lhs\; SB\; TE\; a\; (\text{with } \#(t\$) \geq h'+1)\wedge\\
&\qquad \mathbf{Int} \in types\_ae\; lhs\; SB\; TE\; l \wedge \mathbf{Int} \in types\_ae\; lhs\; SB\; TE\; h\wedge\\
&\qquad (h' < l' \vee l' \geq 0)\\
&\qquad \}\mathbf{where}\;\; h' = val\; h\\
&\qquad\qquad\qquad l' = val\; l\\
&\qquad\qquad\qquad s = h' < l'?\; 0;\;\; h' - l' + \#(\$\text{s in } emp) = 0
\end{aligned}
$$

In words:

the type for a sliced expression is composed, with for its subparts the type that the type of the sliced expression is composed of.

3.4.4.7 Types for arithmetical expressions

If s is a *string denotation*:

$$types\_ae\; lhs\; SB\; TE\; s = \{\mathbf{String}\}$$

If i is a *integer denotation*:

$$types\_ae\; lhs\; SB\; TE\; i = \{\mathbf{Int}, \mathbf{Int} \ni n \mid n \in L\; Name\}$$

If f is a *float denotation*:

$$types\_ae\ lhs\ SB\ TE\ f = \{\textbf{Float}\}$$

If b is a *boolean denotation*:

$$types\_ae\ lhs\ SB\ TE\ b = \{\textbf{Bool}\}$$

$$\begin{aligned} &types\_ae\ lhs\ SB\ TE\ [\![a \to b;\ c]\!] \\ &\quad = \{t \mid \textbf{Bool} \in types\_ae\ lhs\ SB\ TE\ a \wedge \\ &\qquad\quad t \in types\_ae\ lhs\ SB\ TE\ b \cap types\_ae\ lhs\ SB\ TE\ c\} \end{aligned}$$

In words: types for the conditional are those that both branches can have.

Note that this handles both conditionals in appsets and in expressions.

If $\oslash \in \{=, \neq\}$:

$$\begin{aligned} &types\_ae\ lhs\ SB\ TE\ [\![a \oslash b]\!] \\ &\quad = \{\textbf{Bool} \mid types\_ae\ lhs\ SB\ TE\ a \cap types\_ae\ lhs\ SB\ TE\ b \cap \\ &\qquad\quad \{\textbf{Int}, \textbf{Int} \ni n, \textbf{Float}, \textbf{Bool}, \textbf{String}\} \neq \emptyset\ \} \end{aligned}$$

In words: $=$ and $\neq$ compare integer and real numbers, booleans and strings.

If $\oplus \in \{+, -, \times, \textbf{Mod}\}$:

$$\begin{aligned} &types\_ae\ lhs\ SB\ TE\ [\![a \oplus b]\!] \\ &\quad = \{t \mid t \in types\_ae\ lhs\ SB\ TE\ a \cap types\_ae\ lhs\ SB\ TE\ b \cap \{\textbf{Int}, \textbf{Int} \ni n, \textbf{Float}\}\} \end{aligned}$$

$$\begin{aligned} &types\_ae\ lhs\ SB\ TE\ [\![a^b]\!] \\ &\quad = \{t \mid t \in types\_ae\ lhs\ SB\ TE\ a \cap types\_ae\ lhs\ SB\ TE\ b \cap \{\textbf{Int}, \textbf{Int} \ni n, \textbf{Float}\}\} \end{aligned}$$

$$\begin{aligned} &types\_ae\ lhs\ SB\ TE\ [\![a/b]\!] \\ &\quad = \{\textbf{Float} \mid types\_ae\ lhs\ SB\ TE\ a \cap types\_ae\ lhs\ SB\ TE\ b \cap \\ &\qquad\quad \{\textbf{Int}, \textbf{Int} \ni n, \textbf{Float}\} \neq \emptyset\} \end{aligned}$$

$$\begin{aligned} &types\_ae\ lhs\ SB\ TE\ [\![a\ \textbf{Div}\ b]\!] \\ &\quad = \{ti \mid types\_ae\ lhs\ SB\ TE\ a \cap types\_ae\ lhs\ SB\ TE\ b \cap \{\textbf{Int}, \textbf{Int} \ni n, \textbf{Float}\} \neq \emptyset \\ &\qquad \wedge ti \in \{\textbf{Int}, \textbf{Int} \ni n\}\} \end{aligned}$$

If $\odot \in \{<, \leq, \geq, >\}$:

$$\begin{aligned} &types\_ae\ lhs\ SB\ TE\ [\![a \odot b]\!] \\ &\quad = \{\textbf{Bool} \mid types\_ae\ lhs\ SB\ TE\ a \cap types\_ae\ lhs\ SB\ TE\ b \cap \\ &\qquad\quad \{\textbf{Int}, \textbf{Int} \ni n, \textbf{Float}\} \neq \emptyset\} \end{aligned}$$

If $\oslash \in \{\wedge, \vee\}$:

$$types\_ae\ lhs\ SB\ TE\ [\![a \oslash b]\!]$$

$$= \{\mathbf{Bool} \mid \mathbf{Bool} \in \mathit{types\_ae}\ \mathit{lhs}\ SB\ TE\ a \cap \mathit{types\_ae}\ \mathit{lhs}\ SB\ TE\ b\}$$

If $\oplus \in \{+, -\}$:

$$\begin{aligned}&\mathit{types\_ae}\ \mathit{lhs}\ SB\ TE\ [\![\oplus a]\!]\\ &\quad = \{t \mid t \in \mathit{types\_ae}\ \mathit{lhs}\ SB\ TE\ a \cap \{\mathbf{Int}, \mathbf{Int} \ni n, \mathbf{Float}\}\}\end{aligned}$$

$$\mathit{types\_ae}\ \mathit{lhs}\ SB\ TE\ [\![\neg a]\!] = \{\mathbf{Bool} \mid \mathbf{Bool} \in \mathit{types\_ae}\ \mathit{lhs}\ SB\ TE\ a\}$$

$$\mathit{types\_ae}\ \mathit{lhs}\ SB\ TE\ [\![\mathit{itof}\ a]\!] = \{\mathbf{Float} \mid \mathbf{Int} \in \mathit{types\_ae}\ \mathit{lhs}\ SB\ TE\ a\}$$

In words:
itof converts integer denotations to float denotations.

Chapter 4

Specification of macro expansion rules

H. Oolman, KUN

4.1 Context and purpose

The macro expander `glassme` operates in the following context:

$$\mathsf{Glass}\mathit{text} \xrightarrow{\texttt{glassp}} fas_1 \xrightarrow{\texttt{csf}} fas_2 \xrightarrow{\texttt{glassme}} kas_1 \xrightarrow{\texttt{sk}} kas_2$$

fas_2 is an instance of the abstract syntax for the full Glass language. kas_1 an instance of the slightly extended (this is always intended in the following) syntax for the kernel language; it is slightly extended because of a few type-dependent operations that are difficult to incorporate in `glassme`, but easy for `sk`. For definitions of the abstract syntaxes (and conventions for use) see [Ree90].

The tasks of `glassme` are to repeatedly perform the following operations on the input until no more can be done; if a correct kernel Glass text results this is the output, otherwise error messages are given:

- Replace a macro name followed by the required number of actual parameters by the body of the macro definition, in which formal parameter names have been replaced by the corresponding actual values.

  Use the first alternative of the macro definition for which the actual parameters fit the patterns in the formal ones.

  Rename names when necessary to avoid wrong bindings in the rewritten text.

- Treat a system definition with formal parameters like a macro definition.

- Write out expressions using built-in constructs: the conditional, indexing, slicing, arithmetical and boolean operators, and t^e types.

- Remove macro- and parameterized system definitions.

- Rewrite constructs from the full abstract syntax to constructs in the kernel abstract syntax which stand for the same piece of Glass text. This kind of rewriting is necessitated by the use of two different abstract syntaxes.

4.2 Expansion rules, given as a function

In the following the rewritings done by `glassme` will be specified. This is done in the form of functions with as argument fas_2 and as result kas_1, because defining it as a function is often shorten than saying it in words. In some cases the right hand side of the function is defined informally (between quotes) as English text though, because a completely formalized version would become too unreadable.

The functions f are defined in the form

$\mathbf{Def}\, f \in t\ \mathbf{with} f\ p_0 \cdots p_{n-1} = rhs$

i.e. f has type t, n parameters p_i and for those gives result rhs. $L\ X$ is the language generated by the nonterminal X (if used as parameter type, the nonterminals are supposed to refer to the abstract syntax for the full language, in the result type to the kernel language). Instances of abstract syntax are enclosed in $[\![$ and $]\!]$ for clarity. w **where** *defs* is for local definitions (*defs*) with e and *defs* as scope. $c?t; e$ is the conditional, $\mathbb{B}$ is the type for "booleans" (using 0 rather than "false", 1 rather than "false"). Alternative definitions are tried from top to bottom (for the case that the alternatives do not exclude each other).

The meaning of 'error' is that the evaluation of the whole expression this is contained in results in an error message instead of abstract syntax. More formally: functions that can deliver error (or take it as parameter) should have there result (or parameter) type united with $\cup\{\text{error}\}$. If an application can give it as result, this should be tested, and if so the expression being evaluated should also deliver error in stead of abstract syntax (and have its result type extended).

$\mathbf{def}\ exp \in (L\, Glassfas \cup L\, def \cup L\, typ) \rightarrow (L\, Glasskas \cup L\, def \cup L\, typ)\ \mathbf{with}$

for a Glass text:

$exp\ [\![[d_0 \cdots d_{n-1}]]\!] = [exp\ [\![d_0]\!] \cdots exp\ [\![d_{n-1}]\!]]$

for the arguments and results being in (full and kernel) $L\ def$:

$$
\begin{aligned}
&exp\ [\![\text{DefVal}\ or\ a\ (\text{VType}\ t\ (\text{VAtom}\ a))]\!]\\
&\quad = \text{DefAtom}\ or\ a\ [expp t\ [\![t_0]\!] \cdots expp t\ [\![t_{n-1}]\!]]\ (exp\ [\![t_n]\!])\\
&\qquad \textbf{where}\ (\text{TypFun}\ t_0\ (\text{TypFun}\ t_1 \cdots (\text{TypFun}\ t_{n-1} t_n) \cdots) = t) \wedge\\
&\qquad\qquad (t_n \neq \text{TypFun}\ t_a t_r)\\
&exp\ [\![\text{DefVal}\ or\ m\ (\text{VType}\ t\ (\text{VMacAlts}\ rs))]\!]\\
&\quad = \text{"remove from surrounding list"}\\
&exp\ [\![\text{DefVal}\ or\ d\ (\text{VType}\ t\ (\text{VMacLambda}\ fc\ rest))]\!]\\
&\quad = \text{"remove from surrounding list"}\\
&exp\ [\![\text{DefVal}\ or\ d\ (\text{VType}\ t\ rhs)]\!] = \text{DefVal}\ or\ d (exp\ [\![t]\!])(expv\ [\![rhs]\!]\ 0)\\
&exp\ [\![\text{DefCon}\ or\ ns\ e]\!] \qquad = \text{DefCon}\ or (expv\ [\![ns]\!]\ 0)\ (expv\ [\![e]\!]\ 0)\\
&exp\ [\![\text{DefTyp}\ or\ b\ (\text{TypAtom}\ b)]\!] = \text{DefBasetype}\ or\ b\\
&exp\ [\![\text{DefTyp}\ or\ n\ t]\!] \qquad = \text{DefTyp}\ or\ n\ (exp\ [\![t]\!])
\end{aligned}
$$

The two instances of "remove from surrounding list" are for macro- and parameterized system definitions respectively.

For the arguments and results being in (full and kernel) $L\ typ$:

$$
\begin{aligned}
&exp\ [\![\text{TypPwr}\ t\ e]\!] &&= ev = \text{VInt}\ e'?\text{TypProd}\ ts;\ \text{error}\\
& && \quad \textbf{where}\ ev = expv\ [\![e]\!] 1;\\
& && \qquad\quad t' = exp\ [\![t]\!];\\
& && \qquad\quad ts = [t' \cdots t']\ \text{"with } e' \text{ times } t'\text{"}\\
&exp\ [\![\text{TypIn}\ t]\!] &&= \text{TypIn}\ (exp\ [\![t]\!])\\
&exp\ [\![\text{TypOut}\ t]\!] &&= \text{TypOut}\ (exp\ [\![t]\!])\\
&exp\ [\![\text{TypUni}\ t_0\ t_1]\!] &&= \text{TypUni}\ (exp\ [\![t_0]\!])\ (exp\ [\![t_1]\!])\\
&exp\ [\![\text{TypNon}\ t]\!] &&= \text{TypNon}\ (exp\ [\![t]\!])\\
&exp\ [\![\text{TypProd}\ [t_0 \cdots t_{n-1}]]\!] &&= \text{TypProd}\ [exp\ [\![t_0]\!] \cdots exp\ [\![t_{n-1}]\!]]\\
&exp\ [\![\text{TypSym}\ n]\!] &&= \text{TypSym}\ n\\
&exp\ [\![\text{TypAtom}\ n]\!] &&= \text{TypBase}\ n\\
&exp\ [\![t]\!] &&= \text{error}
\end{aligned}
$$

exppt is defined for turning elements of $L\ typ$ from the full abstract syntax used as atom parameter type into the corresponding elements of $L\ partyp$. This is mostly data structure conversion, but also reject several types not allowed for atom parameters in the kernel language.

$$
\begin{aligned}
&\textbf{def}\ exppt \in L\ typ \to L\ partyp\ \textbf{with}\\
&exppt\ [\![\text{TypInt}\ ]\!] &&= \text{PTInt}\\
&exppt\ [\![\text{TypFlo}\ ]\!] &&= \text{PTFlo}\\
&exppt\ [\![\text{TypString}\ ]\!] &&= \text{PTStr}\\
&exppt\ [\![\text{TypBool}\ ]\!] &&= \text{PTBool}\\
&exppt\ [\![t]\!] &&= \text{error}
\end{aligned}
$$

The function *expv* does most of the expansion work. If the second argument is 0 only constructs that fit in $L\ val$ from the kernel abstract syntax are accepted (so not integers, floating numbers, booleans, strings and :-expressions). If this argument is 1 the outcome of this expansion is used in further expansion, implying that constructs not fitting in the kernel abstract syntax can be allowed.

The first parameter and the result are in the language generated by a nonterminal that has as alternatives all alternatives for val in the full and the kernel abstract syntax (both vals are "merged"). This nonterminal will be called $valfk$.

$$
\begin{array}{ll}
\textbf{def}\ expv \in L\ valfk \rightarrow \mathbb{B} \rightarrow L\ valfk\ \textbf{with} & \\
expv\ [\![\text{VType}\ t\ v]\!]\ all & = expv\ [\![v]\!]\ all \\
expv\ [\![\text{VSysLambda}\ fc\ rhs]\!]\ all & = \text{VLambda}\ (expf\ [\![fc]\!])(expv\ [\![rhs]\!]\ all) \\
expv\ [\![\text{VLambda}\ fc\ rhs]\!]\ all & = \text{VLambda}\ fc\ rhs \\
expv\ [\![\text{VSysSigma}\ fc\ rhs]\!]\ all & = \text{VSigma}\ (expf\ [\![fc]\!])\ (expv\ [\![rhs]\!]\ all) \\
expv\ [\![\text{VSigma}\ fc\ rhs]\!]\ all & = \text{VSigma}\ fc\ rhs \\
expv\ [\![\text{VSym}\ or\ n]\!]\ all &
\end{array}
$$

$$
\begin{array}{l}
= (d = \text{DefVal}\ or'n\ (\text{VType}\ t\ (\text{VMacAlts}\ alts)) \vee \\
\quad (d = \text{DefVal}\ or'n\ (\text{VType}\ t\ rest) \wedge k \geq 1)) \wedge \\
\quad \text{"VSym}\ n\ \text{is contained in}\ a\text{"}\ ? \\
\quad \text{"replace}\ a\ \text{by}\ fill\_in\ a\ all \\
\quad ; \\
\quad (\neg all \wedge d = \text{DefVal}\ or'\ n\ (\text{VType}\ t\ (\text{VAtom}\ a))) \wedge \\
\quad \text{"VSym}\ n\ \text{is contained in VApply}\ a\ a_k\text{"}\ ? \\
\quad \text{"replace}\ a\ \text{by" VAtom}\ or'\ n\ [exppv\ [\![a_0]\!] \cdots exppv\ [\![a_{k-1}]\!]]\ a_k\text{"} \\
\quad ; \\
\quad \text{VSym}\ or\ n \\
\quad \textbf{where}\ d = \text{"system/macro/atom declaration that}\ n\ \text{is bound to"}; \\
\qquad\qquad k = \text{"number of parameter (type)s of}\ d\text{"}; \\
\qquad\qquad a = \text{VValApply}\ (\cdots(\text{VValApply}\ (\text{VSym}\ n)a_0)\cdots)a_{k-1}
\end{array}
$$

Explanation: here macro- and parameterized system applications and atom applications of n are rewritten if n is applied to sufficiently many actual parameters.

"is contained in" is not formal; to do it formally, the enclosing applications should have been passed as parameters to the expansion function; doing this would make this definition rather unreadable.

$$
\begin{array}{ll}
expv\ [\![\text{VInt}\ i]\!]\ all & = all?\text{VInt}\ i;\ \text{error} \\
expv\ [\![\text{VFlo}\ f]\!]\ all & = all?\text{VFlo}\ f;\ \text{error} \\
expv\ [\![\text{VBool}\ b]\!]\ all & = all?\text{VBool}\ b;\ \text{error} \\
expv\ [\![\text{VStr}\ s]\!]\ all & = all?\text{VStr}\ s;\ \text{error} \\
expv\ [\![\text{VValApply}\ f\ a]\!]\ all & = \text{VValApply}\ (expv\ [\![f]\!]\ all)\ (expv\ [\![a]\!]\ 1) \\
expv\ [\![\text{VSysApply}\ s\ c]\!]\ all & = \text{VApply}\ (expv\ [\![s]\!]\ 0)\ (expv\ [\![c]\!]\ 0) \\
expv\ [\![\text{VApply}\ s\ c]\!]\ all & = \text{VApply}\ s\ c \\
expv\ [\![\text{VWhere}\ [ld_0 \cdots ld_k]\ v]\!]\ all & = \text{VWhere}\ [exp\ [\![ld_0]\!] \cdots exp\ [\![ld_k]\!]](expv\ [\![v]\!]\ all) \\
expv\ [\![\text{VList}\ [v_0 \cdots v_{k-1}]]\!]\ all & = \neg all?\text{VList}\ [(expv\ [\![v_0]\!]\ 0) \cdots (expv\ [\![v_{k-1}]\!]\ 0)]; \\
& \qquad \text{"}(expv\ [\![v_0]\!]\ 1) : \cdots (expv\ [\![v_{k-1}]\!]\ 1) : [\,]\text{"}
\end{array}
$$

If $all = 1$ this specification does not defined the semantics lazily: the function is applied to each element of the list. This is done so as to keep the specification simple.

$$
\begin{array}{ll}
expv\ [\![\text{VAppset}\ [a_0 \cdots a_{k-1}]]\!]\ all & = \text{VAppset}\ [(expv\ [\![a_0]\!]\ 0) \cdots (expv\ [\![a_{k-1}]\!]\ 0)] \\
expv\ [\![\text{VSyn}\ [ns_0 \cdots ns_{k-1}]]\!]\ all & = \text{VSyn}\ [(expv\ [\![ns_0]\!]\ 0) \cdots (expv\ [\![ns_{k-1}]\!]\ 0)] \\
expv\ [\![\text{VBuiltin}\ "-\ >"\ [c, t, e]]\!]\ all & = c' = \text{VBool True}?expv\ [\![t]\!]\ all; \\
 & \quad c' = \text{VBool False}?expv\ [\![e]\!]\ all; \\
 & \quad \text{error} \\
 & \quad \textbf{where}\ c' = expv\ [\![c]\!]\ 1 \\
expv\ [\![\text{VBuiltin}\ "\ :\ "\ [e, e']]\!]\ all & = \neg all?\text{VComp}\ (expv\ [\![e]\!]\ 0)\ (expv\ [\![e']\!]\ 0); \\
 & \qquad \text{VBuiltin}\ "\ :\ "\ [expv\ [\![e]\!]\ 1, expv\ [\![e']\!]\ 1]
\end{array}
$$

Evaluating e' in case $all = 1$ ($expv\ [\![e']\!]\ 1$) implies that the semantics is not defined lazily.

$$
\begin{array}{l}
expv\ [\![\text{VBuiltin}\ "[\,]"\ [e, i]]\!]\ all \\
\quad = iv = \text{VInt}\ i'?i' \leq 0?\ \text{error}; \\
\qquad\qquad i' \leq k-1?expv\ [\![e_i]\!]\ all; \\
\qquad\qquad er = \text{VList}\ [\,]?\ \text{error}; \\
\qquad\qquad \text{VInd}\ er\ (i'-k); \\
\qquad \text{error} \\
\qquad \textbf{where}\ iv = expv\ [\![i]\!]\ 1; \\
\qquad\qquad ev = expv\ [\![e]\!]\ 1; \\
\qquad\qquad "e_0 : \cdots : e_{k-1} : er" = ev \wedge\ "er \text{ is not of the form } elt : rest" \\
expv\ [\![\text{VBuiltin}\ "[..]"[e, l, m]]\!]\ all \\
\quad = lv = \text{VInt}\ l' \wedge mv = \text{VInt}\ m'? \\
\qquad\qquad m' < l'?\text{VList}\ [\,]; \\
\qquad\qquad l' < 0?\ \text{error}; \\
\qquad\qquad m' \leq k-1?\ "e_{l'} : \cdots : e_{m'} : [\,]"; \\
\qquad\qquad l' \leq k-1?\ "e_{l'} : \cdots : e_{k-1} : \text{VSlice}\ er\ 0\ (m'-k); \\
\qquad\qquad er = \text{VList}\ [\,]?\ \text{error}; \\
\qquad\qquad \text{VSlice}\ er(l'-k)\ (m'-k); \\
\qquad \text{error} \\
\qquad \textbf{where}\ lv = expv\ [\![l]\!]\ 1; \\
\qquad\qquad mv = expv\ [\![m]\!]\ 1; \\
\qquad\qquad ev = expv\ [\![e]\!]\ 1; \\
\qquad\qquad "e_0 : \cdots : e_{k-1} : er" = ev \wedge\ "er \text{ is not of the form } elt : rest"
\end{array}
$$

In stead of "VSlice $e\ a\ b$" the following would be even nicer:
"VInd $z\ a : \cdots :$ VInd $z\ b : [\,]$" (z a new name) and '$z = e$' added to the first surrounding where clause.

$$
\begin{array}{ll}
\multicolumn{2}{l}{expv\ [\![\text{VBuiltin}\ en\ [l, r]]\!]\ all} \\
\quad = lv = \text{VInt}\ l' \wedge rv = \text{VInt}\ r' & ?\text{VBool}\ (l'\ enop\ r'?\ \text{True}; \text{False}); \\
\qquad lv = \text{VFlo}\ l' \wedge rv = \text{VFlo}\ r' & ?\text{VBool}\ (l'\ enop\ r'?\ \text{True}; \text{False}); \\
\qquad lv = \text{VBool}\ l' \wedge rv = \text{VBool}\ r' & ?\text{VBool}\ (l'\ enop\ r'?\ \text{True}; \text{False}); \\
\qquad lv = \text{VStr}\ l' \wedge rv = \text{VStr}\ r' & ?\text{VBool}\ (l'\ enop\ r'?\ \text{True}; \text{False}); \\
\qquad \text{error} & \\
\qquad \textbf{where}\ lv = expv\ [\![l]\!]\ 1; rv = expv\ [\![r]\!]\ 1 &
\end{array}
$$

This rule holds for en = "=" (in which case $enop$ ==), and for en = "/=" (in which case $enop$ =≠).

$expv$ [[VBuiltin $op2$ $[l, r]$]] all
$= lv = Varg\ l' \wedge rv = Varg\ r' ? Vres\ (oper\ l'\ r')$; error
where $lv = expv$ [[l]] $1; rv = expv$ [[r]] $1;$
$oper$ = "operation corresponding to $op2$"

This rule holds for:

$op \in$	with $Varg$ =	and $Vres$ =
{"<", "<=", ">=", ">"}	VInt or VFlo	VBool
{"+2", "-2", "^", "*", "MOD"}	VInt or VFlo	$Varg$
{"/"}	VInt or VFlo	VFlo
{"DIV"}	VInt or VFlo	VInt
{"&", "|"}	VBool	VBool

$expv$ [[VBuiltin $op1$ $[l]$]] all
$= lv = Varg\ l' ? Vres\ (oper\ l')$; error
where $lv = expv$ [[l]] $1; oper$ = "operation corresponding to $op1$"

This rule holds for:

$op \in$	with $Varg$ =	and $Vres$ =
{"+1", "-1"}	VInt or VFlo	$Varg$
{"itof"}	VInt	VFlo
{"~"}	VBool	VBool

The function $expf$ turns a formal connection parameter into the kernel abstract syntax for that:

def $expf \in L\ fp \rightarrow L\ formcon$ **with**
$expf$ [[FpList $[fc_0 \cdots fc_{n-1}]$]] = FCList $[expf$ [[fc_0]] $\cdots expf$ [[fc_{n-1}]]$]$
$expf$ [[FpSym n]] = FCSym n
$expf$ [[FpCompos $f\ r$]] = FCComp ($expf$ [[f]]) ($expf$ [[r]])
$expf$ [[Fpx]] = error

The function $exppv$ turns parameter values for atoms into the kernel abstract syntax for that, if the parameter value is allowed in the kernel language:

def $exppv \in L\ valfk \rightarrow L\ parval$ **with**
$exppv$ [[VInt i]] =ParInt i
$exppv$ [[VFlo f]] =ParFlo f
$exppv$ [[VBool b]] =ParBool b
$exppv$ [[VStr s]] =ParStr s
$exppv$ [[Vx]] = error

$fill\_in\ a\ all$ gives the expanded version of a (a macro/parameterized system application, for which the parameters have already been expanded with all = 1):

def $fill\_in \in L\ valfk \rightarrow L\ valfk$ **with**
$fill\_in\ a\ all$
$\quad = expv\ [\![rest']\!]\ all$
where
$a =$ VValApply $(\cdots($VValApply (VSym $n)a_0)\cdots a_{k-1})$;
("n is bound by DefVal *or* n (VType t (VMacAlts $[alt_0 \cdots alt_{m-1}]$))" $\wedge$
VMacLambda $f_0(\cdots($VMacLambda $f_{k-1}\ rest)\cdots) =$
"the first alt_i such that all formal parameters f_i match the a_i
(with FpList $[x_0 \cdots x_{m-1}]$ equivalent to "$x_0 : \cdots : x_{m-1} : [\,]$")"
$\vee$
"n is bound by the parameterized system definition
'DefVal *or* n (VType t (VMacLambda $f_0(\cdots($VMacLambda $f_{k-1} rest)\cdots))$'
;
$rest' =$"$rest$, after renaming and then replacing all instances of VSym s,
with s a name from the f_i, by the corresponding part of a_i"

Remarks:
The type for *rest all* would have been t, in which values from the a_i for `INT-:n` names have been filled in. Since in the kernel language there is nothing like the VType constructor, adding this type can not be done.

Renaming: in *rest* replace all connection names (those not explicitly bound as parameter to be filled in or as atom, system definition or macro definition), names of atoms, system definitions and macro definitions in where clauses, formal parameters of system definitions and atoms in where clauses.

Where 'where clauses' is used in the above, this also means where clauses found in macro and system definitions defined in where clauses etc.

4.3 An example

Consider the Glass text:

```
BASETYPE B;
ATOM not :- B => B;
DEF d :- B => B;
    d  = not
```

Applying the parser and type-checker gives as result:

```
[(DefTyp or "B" (TypAtom "B")),
 (DefVal or "not"
 (VType (TypUni (TypAtom "B") (TypAtom "B")) (VAtom or "not"))),
 (DefVal or  "d"
 (VType (TypUni (TypAtom "B") (TypAtom "B"))
              (VSysLambda (FpName "x")
                  (VSysApply (VSym or "not") ( VSym or "x")))))]
```

where `or` is an abbreviation of the origin of the definition.

The definition of *exp* for the DefTyp says that the first element in this list is rewritten (since the names in DefTyp and TypAtom are the same: b=`"B"`) to `DefBaseType or "B"`.

The second element in the list is rewritten using the rule

$$\begin{aligned}
&exp\ [\![\text{DefVal}\ or\ a\ (\text{VType}\ t\ (\text{VAtom}\ a))]\!] \\
&\quad = \text{DefAtom}\ or\ a\ [exppt\ [\![t_0]\!] \cdots exppt\ [\![t_{n-1}]\!]]\ (exp\ [\![t_n]\!]) \\
&\qquad \textbf{where}\ (\text{TypFun}\ t_0\ (\text{TypFun}\ t_1 \cdots (\text{TypFun}\ t_{n-1} t_n) \cdots) = t) \wedge \\
&\qquad\qquad (t_n \neq \text{TypFun}\ t_a t_r)
\end{aligned}$$

Now `"not"` is used for a, `TypUni (TypAtom "B") (TypAtom "B")` for t and $n = 0$. Thus $exp\ [\![t_n]\!]$ is `TypUni (TypBase "B") (TypBase "B")`, so the second element of the list is rewritten to:
`DefAtom or "not" [] (TypUni (TypBase "B") (TypBase "B"))`.

Analogously, the third element in the list is rewritten to:

```
(DefVal or "d" (TypUni (TypBase "B") (TypBase "B"))
       (VLambda (FCSym "x") (VAtom or "not" [] (VSym or "x"))))
```

Chapter 5

Example descriptions

5.1 A one second pulse generator

Basetype B;

Atom
$\quad divide\_by\_10 \in B \Rightarrow B,$
$\quad not \in B \Rightarrow B;$

Atom
$\quad R \in \mathbf{Float} \rightarrow [B \times B],$
$\quad C \in \mathbf{Float} \rightarrow [B \times B],$
$\quad Xtal \in \mathbf{Float} \rightarrow [B \times B];$

Def
$\quad Amp \in [B \times B];$
$\quad Amp\ \langle in, out \rangle =$
$\quad\quad \{\ not\ \langle in, out \rangle,$
$\quad\quad\quad R\ 680\ \langle in, out \rangle$
$\quad\quad \};$

Def
$\quad Xtal\_feedback \in \mathbf{Float} \rightarrow [B \times B];$
$\quad Xtal\_feedback\ freq\ \langle in, out \rangle =$
$\quad\quad \{\ C\ 10p\ \langle in, x \rangle,$
$\quad\quad\quad Xtal\ freq\ \langle x, out \rangle$
$\quad\quad \};$

Def

$Xtal\_oscillator \in \mathbf{Float} \rightarrow [?\langle\rangle, !B];$
$Xtal\_oscillator\ freq\ out =$
$\{\ Amp\ \langle x, y\rangle;$
$Amp\ \langle y, z\rangle;$
$Xtal\_feedback\ freq\ \langle z, x\rangle$
$\}$
where $out = not\ z$
endwhere;

Mac

$divider\_train \in \mathbf{Int} \rightarrow B \Rightarrow B \rightarrow B \Rightarrow B;$
$divider\_train\ 0\ A\ clock = clock;$
$divider\_train\ n\ A\ clock = divider\_train\ (n-1)\ A\ (A\ \langle clock\rangle);$

Def

$divide\_by\_million \in B \Rightarrow B;$
$divide\_by\_million\ clock = divider\_train\ 6\ divide\_by\_10\ clock;$

Def

$onesecondpulse \in \langle\rangle \Rightarrow B;$
$onesecondpulse = divide\_by\_million\ (Xtal\_oscillator\ 1M);$

5.2 A Clapp-Eco Oscillator

Basetype B;

Atom

$Penthode \in \mathbf{String} \rightarrow [B \times B \times B \times B \times B];$
$/*\ [k, g1, g2, g3, a]\ */$

Atom

$R \in \mathbf{Float} \rightarrow [B \times B],$
$C \in \mathbf{Float} \rightarrow [B \times B],$
$Cvar \in \mathbf{Float} \rightarrow [B \times B],$
$Sm \in \mathbf{Float} \rightarrow [B \times B],$
$L \in \mathbf{Float} \rightarrow [B \times B];$

Atom

$Plus310V \in B,$
$Ground \in B;$

Def

$ClappEco \in \langle\rangle;$
$ClappEco\ \langle\rangle =$
$\{\ Penthode\ "EF80"\ \langle kathode, grid1, grid2, kathode, anode\rangle,$
$C\ 180p\ \langle grid1, kathode\rangle,$
$C\ 180p\ \langle kathode, minus\rangle,$
$Sm\ 1m\ \langle kathode, minus\rangle,$
$L\ 2\mu\ \langle grid1, x\rangle,$
$C\ 15p\ \langle x, minus\rangle,$
$C\ 10n\ \langle grid2, minus\rangle,$
$R\ 10k\ \langle grid2, plus\rangle,$
$C\ 15p\ \langle anode, plus\rangle,$
$L\ 2\mu\ \langle anode, plus\rangle,$
$Plus310V\ \langle plus\rangle,$
$Ground\ \langle minus\rangle$
$\};$

5.3 A counter

Atom

$jkff \in B \times B \Rightarrow B,$
$and2 \in B \times B \Rightarrow B,$
$and3 \in B^3 \Rightarrow B,$
$and4 \in B^4 \Rightarrow B,$
$and5 \in B^5 \Rightarrow B,$
$or \in B \times B \Rightarrow B,$
$buf \in B \Rightarrow B,$
$nand \in B \times B \Rightarrow B;$

Def

$counter \in B^8 \Rightarrow B^5;$
$counter\langle d_3, d_2, d_1, d_0, nload, nclear, enap, enat\rangle =$
$\langle carry, q_3, q_2, q_1, q_0\rangle$
where
$carry = and5\ \langle q_3, q_2, q_1, q_0, enat\rangle;$
$q_3 = counterstage\ \langle d_3, preset, and4\ \langle q_2, q_1, q_0, en\rangle, nclr\rangle;$
$q_2 = counterstage\ \langle d_2, preset, and3\ \langle q_1, q_0, en\rangle, nclr\rangle;$
$q_1 = counterstage\ \langle d_1, preset, and2\ \langle q_0, en\rangle, nclr\rangle;$
$q_0 = counterstage\ \langle d_0, preset, en, nclr\rangle;$
$preset = nand\ \langle nload, nclr\rangle;$
$nclr = buf\ \ nclear;$
$en = and2\ \langle enap, enat\rangle;$

Mac
$counterstage \in B^4 \Rightarrow B;$
$counterstage\ \langle d, preset, toggle, nclr \rangle =$
$jkff\ \langle and2 \langle ja, ea \rangle, and2\ \langle ka, ea \rangle \rangle$
where
$ja = nand\ \langle ka, preset \rangle;$
$ka = nand\ \langle d, nclr \rangle;$
$ea = or\ \langle toggle, preset \rangle;$
endwhere;
endwhere;

Appendix A

Context-free syntax of Glass

H. Oolman, KUN

This text is intended as a quick reference grammar.

A.1 Humanly readable grammar

The following grammar is meant to be easily readable; it does not concern itself with problems like ambiguity, and parsers not able e.g. to handle left-recursion.

Notation: e stands for an empty string, 'x?' means x is optional, '$xy \cdots yx$' stands for one or more xs, separated by ys, '$(\cdots)$' are used for priority, terminals are underlined. Some nonterminals are 'parameterised'; these stand for several nonterminals, each with one of the possible values for the parameter filled in.

Some nonterminal from the Glass report are given shorter names here for readability. Official names will be given too.

Comment is enclosed in /* and */. $\#include$ "$filename$" can be used to combine with definitions from other files.

$glasstext = elt\underline{;}\cdots\underline{;}elt$

$elt =$ $\qquad$ $(= glass\ text\ element)$

$\quad \underline{\textbf{Basetype}}\ atomic\_type\_name\underline{,}\cdots\underline{,}atomic\_type\_name \mid$
$\quad \underline{\textbf{Type}}\ (typename \underline{=} type)\underline{,}\cdots\underline{,}(typename \underline{=} type) \mid$
$\quad \underline{\textbf{Atom}}\ (name \underline{\in} type)\underline{,}\cdots\underline{,}(name \underline{\in} type) \mid$
$\quad def \mid$
$\quad mac \mid$
$\quad \epsilon$

$def =$ $\qquad$ $(= system\ definition)$

$\quad \underline{\textbf{Def}}\ name \underline{\in} type\underline{;}\ name(name \cdots name)?\ vars?\ \underline{=} rhs$

$mac = \underline{\textbf{Mac}}\ name \underline{\in} type\underline{;}\ alt\underline{;}\cdots\underline{;}\ alt$ $\qquad$ $(= macro\ definition)$

$alt = name(fp \cdots fp)?\ \underline{\equiv}\, rhs$ $\qquad$ $(= definition\ part)$

$rhs = appset \mid expr$

$expr = expr\ \underline{\textbf{Where}}\ ld\underline{;} \cdots \underline{;} ld\ \underline{\textbf{Endwhere}} \mid$ $\qquad$ $(= expression)$
$\quad expr\ \underline{\rightarrow}\ expr\underline{;}\ expr \mid$
$\quad name \mid$
$\quad \underline{\langle}(expr\underline{,} \cdots \underline{,}expr)?\underline{\rangle} \mid$ $\qquad$ (so $\langle\rangle$, $\langle e_0\rangle$, $\langle e_0, \cdots\rangle$)
$\quad expr\underline{:}\ expr \mid$
$\quad expr\ expr \mid$
$\quad expr\underline{@}expr\underline{\ldots}expr \mid$
$\quad expr\ binop\ expr \mid$
$\quad monop\ expr \mid$
$\quad literal \mid$
$\quad \underline{(}expr\underline{)} \mid$
$\quad \underline{\lambda} vars\underline{.}expr \mid \underline{\sigma} vars\underline{.}appset$

$ld = vars\ \underline{\equiv}\ expr \mid elt$

$binop = \underline{\equiv} \mid \underline{\neq} \mid \underline{\leq} \mid \underline{\leqq} \mid \underline{\geqq} \mid \underline{\geq} \mid \underline{+} \mid \underline{-} \mid \underline{\hat{\ }} \mid \underline{\times} \mid \underline{/} \mid \underline{\textbf{Div}} \mid \underline{\textbf{Mod}} \mid \underline{\wedge} \mid \underline{\vee}$

$monop = \underline{+} \mid \underline{-} \mid \neg \mid \underline{itof}$

$appset = appset\ \underline{\textbf{Where}}\ ld\underline{;} \cdots \underline{;}\ ld\ \underline{\textbf{Endwhere}} \mid$
$\quad \underline{\{}(apselt\underline{,} \cdots \underline{,}apselt)?\underline{\}}$

$apselt = expr\ \underline{\rightarrow}\ apselt\underline{;}\ apselt \mid$ $\qquad$ $(= appset\ element)$
$\quad expr \mid$
$\quad appset \mid$
$\quad \underline{\star\langle} vars\underline{,} \cdots \underline{,}vars\underline{\rangle}$

$vars = name \mid \underline{\langle}(vars\underline{,} \cdots \underline{,}vars)?\underline{\rangle} \mid$ $\qquad$ $(= name\ tuple)$
$\quad vars\underline{:}\ vars \mid \underline{(}vars\underline{)}$

$fp = literal \mid name \mid \underline{(}fp\underline{)} \mid \underline{\langle}(fp\underline{,} \cdots \underline{,}fp)?\underline{\rangle} \mid fp\underline{:}\ fp$ $\qquad$ $(= formal\ parameter)$

$literal = int \mid float \mid string \mid boolean$

$int = \ldots$ an integer $\ldots$
$float = \ldots$ floating point, incl. exponential and electrical scale notation $\ldots$
$string = \ldots$ enclosed in double quotes, "" for " $\ldots$
$boolean = \underline{\textbf{True}} \mid \underline{\textbf{False}}$

$type = (\underline{\{}typevar\underline{,} \cdots \underline{,}typevar\underline{\}})?typ$ all restr

In the following,

if x=restr the nonterminal is only for connection-types, i.e. for t_0 and t_1 from $t_0 \Rightarrow t_1$ and $[t_0]$;

if z=restr, no ? or ! are allowed: ? and ! are only allowed (unnested) in adirectional systemtypes.

$$
\begin{aligned}
typ\ x\ z\ =\ & \underline{(}typ\ x\ z\underline{)}\ | \\
& typename\ | \\
& \text{(the next line is excluded if } z\text{=restr:)} \\
& \underline{?}typ\ \text{restr restr}\ |\ \underline{!}typ\ \text{restr restr}\ | \\
& typ\ x\ z\underline{\times}\cdots\underline{\times}typ\ x\ z\ | \\
& \underline{\langle\rangle}\ | \qquad\qquad \text{(type for empty tuple)} \\
& typ\ x\ z\underline{\hat{}}expr\ | \\
& typ\ x\ z\underline{^*}\ | \\
& typevar\ | \\
& atomic\_type\_name \\
& \text{(the following up to the end are not allowed if } x = \text{restr:)} \\
& | \\
& \underline{\textbf{Appset}}\ | \\
& typ\ \text{all restr}\ \underline{\rightarrow}\ typ\ \text{all restr}\ | \\
& typ\ \text{restr restr}\ \underline{\Rightarrow}\ typ\ \text{restr restr}\ | \quad (t_1 \Rightarrow t_2 \text{ equiv. } [?t_1\times!t_2]) \\
& \underline{[}typ\ \text{restr all}\underline{]}\ | \\
& \underline{\textbf{Int}}\ \underline{\ni}\ name\ |\ \underline{\textbf{Int}}\ |\ \underline{\textbf{Float}}\ |\ \underline{\textbf{String}}\ |\ \underline{\textbf{Bool}}
\end{aligned}
$$

$$
\begin{aligned}
atomic\_type\_name &= name \\
typename &= name \\
typevar &= name \\
name &= \ldots \text{ starts with letter, uses letters, digits, } ' \text{ and } \_ \ldots
\end{aligned}
$$

A.2 Nonambiguous grammar

The next grammar redefines some nonterminals from the previous one, in order to remove ambiguities (hopefully), by expressing precedences (rules for operators with lower precedences appear first) and associativity. For example, if *op* is right associative: '$x = y\ op\ x\ |\ y$'.

$$def = \underline{\textbf{Def}}\ name\ \underline{\in}\ type\underline{;}\ name(vars\cdots vars)?\ \underline{=}\ rhs$$

This change is made because the parser can not know if a name, used as last parameter, is meant as the optional *vars* or if this option was not used. This new syntax admits too much: not-allowed forms will be rejected after parsing.

If a macro is defined, all consecutive $n \ldots = \ldots$ with n the name of the macro are part of the same macro definition.

$rhs = appset \mid expr$

$expr = expr0 \,\underline{\rightarrow}\, expr\underline{;}\, expr \mid expr0$

$expr0 = \underline{\lambda} fp\underline{.} expr1 \mid \underline{\sigma} fp\underline{.} appset \mid$
$\quad expr0 \; \underline{\textbf{Where}} \; ld\underline{;} \cdots \underline{;} ld \; \underline{\textbf{Endwhere}} \mid$
$\quad expr1$
$expr1 = expr2\underline{:}expr1 \mid expr2$
$expr2 = expr2 \; ene \; expr3 \mid expr3$
$expr3 = expr4 \; cmp \; expr4 \mid expr4$
$expr4 = expr4 \; add \; expr5 \mid expr5$
$expr5 = expr5 \; mult \; expr6 \mid expr6$
$expr6 = expr7\underline{\hat{}}expr6 \mid expr7$
$expr7 = monop \; expr7 \mid expr8$
$expr8 = expr8\underline{@}expr \ldots expr \mid expr9$
$expr9 = expr9 \; expr10 \mid expr10$
$expr10 = name \mid$
$\quad \underline{(}expr\underline{)} \mid$
$\quad \underline{\langle}(expr\underline{,} \cdots \underline{,}expr)?\underline{\rangle} \mid$
$\quad literal$

$apselt = expr0 \,\underline{\rightarrow}\, apselt\underline{;}\, apselt \mid$
$\quad expr0 \mid$
$\quad appset \mid$
$\quad \underline{\star\langle} vars\underline{,} \cdots \underline{,}vars\underline{\rangle}$

$ene = \underline{=} \mid \underline{\neq}$
$cmp = \underline{<} \mid \underline{\leq} \mid \underline{\geq} \mid \underline{>}$
$add = \underline{+} \mid \underline{-} \mid \underline{\vee}$
$mult = \underline{\times} \mid \underline{/} \mid \underline{\textbf{Div}} \mid \underline{\textbf{Mod}} \mid \underline{\wedge}$

$vars = vars1\underline{:}\, vars \mid vars1$
$vars1 = name \mid \underline{(}vars\underline{)} \mid \underline{\langle}(vars\underline{,} \cdots \underline{,}vars)?\underline{\rangle}$

$fp = fp1\underline{:}\, fp \mid fp1$
$fp1 = literal \mid \underline{(}fp\underline{)} \mid \underline{\langle}(fp\underline{,} \cdots \underline{,}fp)?\underline{\rangle} \mid name$

$typ \; x \; z = type1 \; x \; z$

$\quad \mid type1 \; \mathrm{all} \; \mathrm{restr} \,\underline{\rightarrow}\, typ \; \mathrm{all} \; \mathrm{restr}$ (not if x is restr)
$type1 \; x \; z = type2 \; x \; z$

(the following not if x is restr:)
$type2 \; \mathrm{restr} \; \mathrm{restr} \,\underline{\Rightarrow}\, type2 \; \mathrm{restr} \; \mathrm{restr} \mid \underline{[}type2 \; \mathrm{restr} \; \mathrm{restr}\underline{]}$
$type2 \; x \; z = type3 \; x \; z \,\underline{\times}\, type2 \; x \; z \mid type3 \; x \; z$

$type3\ x\ z = type4\ x\ z\underline{\hat{\ }}expr \mid type4\ x\ z\underline{^*} \mid type4\ x\ z$
$type4\ x\ z = type5\ x\ z$

(the following line not if z=restr:)
$\mid \underline{?}type5$ restr restr $\mid \underline{!}type5$ restr restr
$type5\ x\ z = \underline{(}typ\ x\ z\underline{)} \mid$
$typename \mid$
$\underline{\langle\rangle} \mid$
$typevar \mid$
$atomic\_type\_name$

(the following up to the end not if x=restr:)
$\mid$
Appset $\mid$
Int $\underline{\ni}$ $name \mid$ **Int** $\mid$ **Float** $\mid$ **String** $\mid$ **Bool**

Appendix B

Context restrictions for Glass

H. Oolman, KUN

This text is meant as a quick reference guide for the context-sensitive aspects of Glass.

B.1 Restrictions on the binding of names

For a Glass text there should hold:

- all *glass text element* names on one level must differ.
- Type namings must not be defined cyclicly.
- Names in types must be local type variables, basetype names or names from type namings (these names are bound to the nearest enclosing definition).
- After expanding type namings, a correct type must result.
- The names from **Int** $\ni n$ in one type must differ.

For a system definition and each alternative of a macro definition there should hold:

- In case of a system definition the formal parameters (the first n, if the type has n $\rightarrow$s) may only be names.
- The names in the formal parameters of one definition must all differ.
- The names in the formal connection parameter (optionally following the formal parameters) of one definition must differ from those in the formal parameters.

B.2 Typing rules

The are two variants: a severe one that checks the exact size of tuples/lists and does index-out-of-bounds checks, and a less severe one that does not (meant for texts with macros).

B.2.1 Concerning declared types

For types found in a Glass text the following holds:

- **Int** and **Int** $\ni n$ are always equivalent (so also: if you see any of them in the following, the other is equally valid).
- e in t^e must have type **Int**.
- t^e is treated the same as t^*, namely as $t\$ \cdots \$t\$\langle\rangle$.

  $t_0\$ \cdots t_{n-1}\$\langle\rangle$ is not Glass: it is another notation for $t_0 \times \cdots \times t_{n-1}$, used because tuple/list types ($\times$ types) can not be used for $n = 1$ and $n = 0$.
- A type variable may only be used if any connection type filled in for (all occurrences of) it would give a valid type for the definition.

  Likewise, when a name is used, it is supposed to have a type obtained by filling in some connection type for each type variable.

  A connection type is any type that connections between parts of a circuit can have.

B.2.2 Typing rules for the syntactic constructs

To save writing, t_x stands for the type of x.

For **Mac** $n \in t; n\ p_0 \cdots p_{k-1} = ea$ (the same for every alternative of a macro defintion, and if **Mac** is replaced by **Def**):

- If t is $t_1 \rightarrow t_2$, $t_1 \Rightarrow t_2$ or $[t_1]$, but $k = 0$, the definition is equivalent to one where a (new) name has been added both as parameter and at the end of the right hand side.
- If $t = t_1 \rightarrow t_2$, k must be ≥ 1, and given t_1 for p_0, **Mac** $n \in t_2; n\ p_1 \cdots p_{k-1} = ea$ must be type-correct (p_0 is a formal parameter).
- As to system types, $t_1 \Rightarrow t_2$ and $[?t_1 \times !t_2]$ are equivalent anywhere, and $?t$ and $!t$ are equivalent to the directions (? and !) being spread over the subparts of t (so if you find any of these in this text, one of its equivalents is also valid).
- If $t = t_1 \Rightarrow t_2$, k must be 0 and, given t_1 for p_0, ea must have type t_2 (p_0 is a formal connection parameter).

- If $t = [t_1]$, k must be 0 and, given t_1 (without the ? and !) for p_0, ea must have type **Appset** (p_0 is a formal connection parameter).

Scoping rules:

names in expressions are bound to the nearest enclosing (w.r.t. where clauses) formal parameter or atomic component/system/macro declaration, or else they are not-explicitly bound names for connections.

Typing rules for expressions:

- e **Where**...**Endwhere** has type t_e; in $ns = e$ used as one of the elements in a where-clause, ns and e must have the same connection type.

- $\lambda fp.e$ has type $t_{fp} \Rightarrow t_e$, or this type with some of the directions left off.

- $\sigma fp.e$ has type $[t_f p]$; t_e must be **Appset**.

- $\{a_0, \cdots, a_{n-1}\}$ has type **Appset**; the t_{a_i} must be **Appset** too.

- $\star\langle ns_0, \cdots, ns_{n-1}\rangle$ has type **Appset**, if all the t_{ns_i} are equal connection types.

- $\langle ae_0, \cdots, ae_{n-1}\rangle$ is exactly equivalent to $ae_0 : \cdots ae_{n-1} : \langle\rangle$.

- $a : b$ has type $t_a\$\, t_b$; t_b must be of the form $t_0\$ \cdots \$t_{n-1}\$\langle\rangle$.

- $\langle\rangle$ has type $\langle\rangle$ in the severe variant, and $t\$ \cdots t\$\langle\rangle$ (t arbitrary) in the less severe variant (because no check is done on the exact size then).

- A name has the type declared for it, if the name was declared at all, with some type filled in for all the type variables, as already explained. If the name was not explicitly declared (i.e. it is a connection name), it gets the (connection) type that is given to all the occurrences of it.

  If a system type was declared, it may be used as if (some of the) directions in the type were not present.

- fa has type t if t_f is $t_a \rightarrow t$, or
  t_f is $t_a \Rightarrow t$ (t_a a connection type), or
  t_f is $t\$ \cdots t\$\langle\rangle$ and t_a is **Int**
  (severe: check if index in range);
  Appset if t_f is $[t_a]$ (t_a a connection type).

- $a@x \ldots y$ has type $t\$ \cdots t\$\langle\rangle$, if t_a is $t\$ \cdots t\$\langle\rangle$ and $t_x = t_y =$ **Int**. In the severe variant it is checked if the indices x and y are in range, and the resulting type has exactly the size indicated by the values of x and y.

- A string has type **String**.
- An integer has type **Int**.
- A floating point number has type **Float**.
- A boolean has type **Bool**.
- $a \rightarrow b; c$ has type t_b, if $t_a =$ **Bool** and $t_b = t_c$.
- $a = b$ and $a \neq b$ have type **Bool**, if t_a and t_b are both **Int**, **Float**, **Bool** or **String**.
- $a + b$, $a - b$, $a \times b$, a **Mod** b and a^b have type t_a, if $t_a = t_b$, and these are **Int** or **Float**.
- a/b has type **Float**, if $t_a = t_b$, and these are **Int** or **Float**.
- a **Div** b has type **Int**, if $t_a = t_b$, and these are **Int** or **Float**.
- $a < b$, $a \leq b$, $a \geq b$ and $a > b$ have type **Bool**, if $t_a = t_b$, and these are **Int** or **Float**.
- $a \wedge b$ and $a \vee b$ have type **Bool**, if $t_a = t_b =$ **Bool**.
- $+a$ and $-a$ have type t_a, if t_a is **Int** or **Float**.
- $\neg a$ has type **Bool**, if t_a is **Bool**.
- $itof\ a$ has type **Float**, if t_a is **Int**.

Appendix C

Syntax Diagrams For Glass

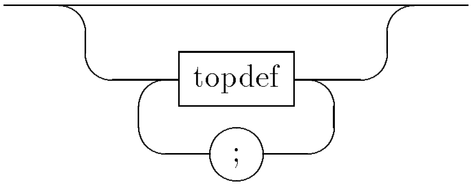
L.W.J. Rooijakkers, M. Seutter, KUN.

In this appendix the syntax of Glass will be represented using rail road diagrams, consisting of boxes, ovals and rail roads. However the names of the nonterminals used in the grammar of the language are too large to fit in most of the boxes. We will therefore abbreviate them or leave them out (rail road diagrams often express more than one grammar rule). The actual changes are documented in the following table:

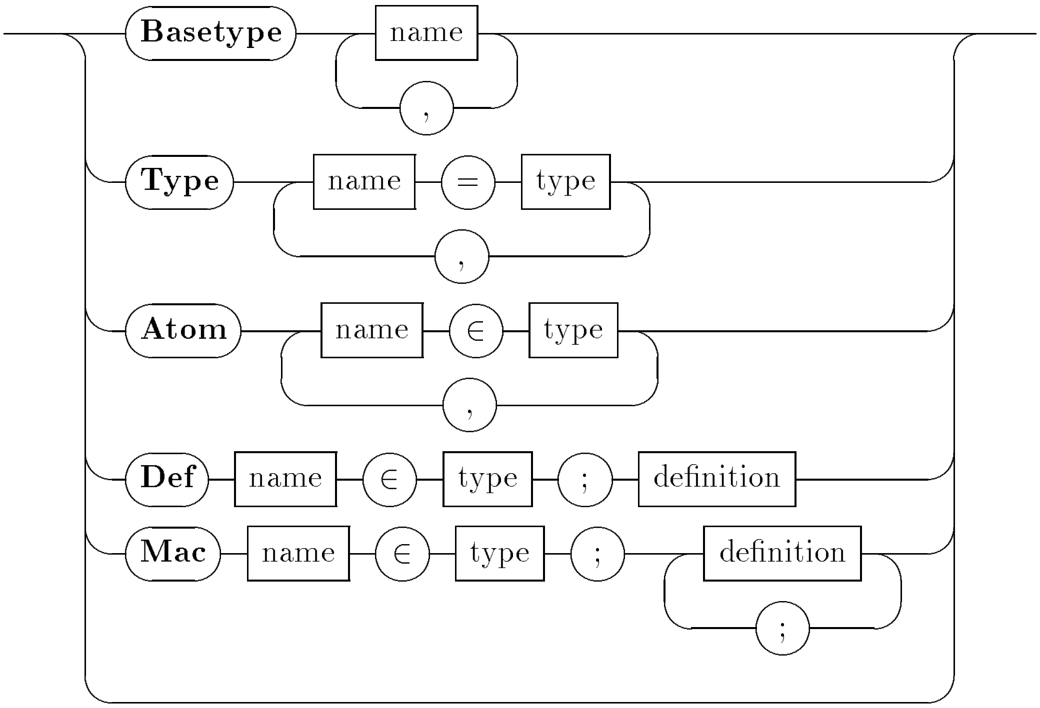
nonterminal	change
adding operator	*addop*
adirectional system type	into *type*
appset	*appset*
appset element	*element*
atomic system definitions	into *topdef*
base type declarations	into *topdef*
declaration	into *topdef*
definition part	*definition*
directional system type	into *type*
electrical scale	*scale*
equality operator	*eqop*
exponent	*exponent*
expr0	*expr 1–11*
expr1	*expr 2–11*
expr2	*expr 3–11*
expr3	*expr 4–11*
expr4	*expr 5–11*
expr5	*expr 6–11*
expr6	*expr 7–11*
expr7	*expr 8–11*
expr8	*expr 9–11*
expr9	*expr 10–11*
expr10	*expr 11*

nonterminal	change
expression	*expr*
formal parameter	*formal*
formal parameter1	*formal 1*
formal tuple parameter	into *formal*
glass text	*text*
glass text element	into *topdef*
lambda abstraction	into *expr*
local definition	*localdef*
local type specification	into *type*
macro definition	into *topdef*
monadic operator	*monop*
multiplicative operator	*mulop*
name tuple	*tuple*
name tuple1	*tuple 1*
parametrized integer type	into *type*
relational operator	*relop*
rhs	*rhs*
sigma abstraction	into *expr*
synonym term	into *element*
system definition	into *topdef*
tuple	into *expr*
type	*type*
type0	*type 1–6*
type1	*type 2–6*
type2	*type 3–6*
type3	*type 4–6*
type4	*type 5–6*
type5	*type 6*
type naming	into *topdef*
type namings	into *topdef*
where clause	into *appset*, *expr*

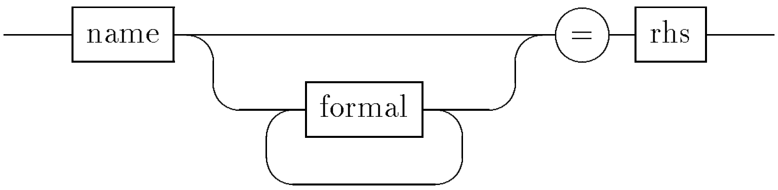
text



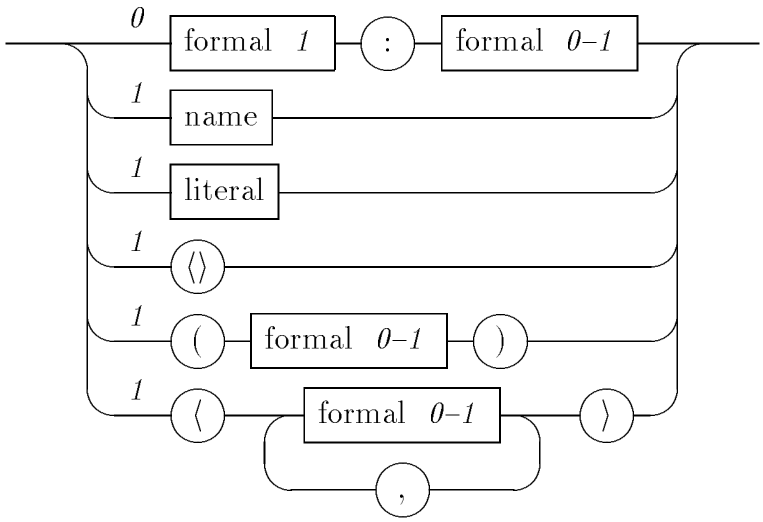
topdef



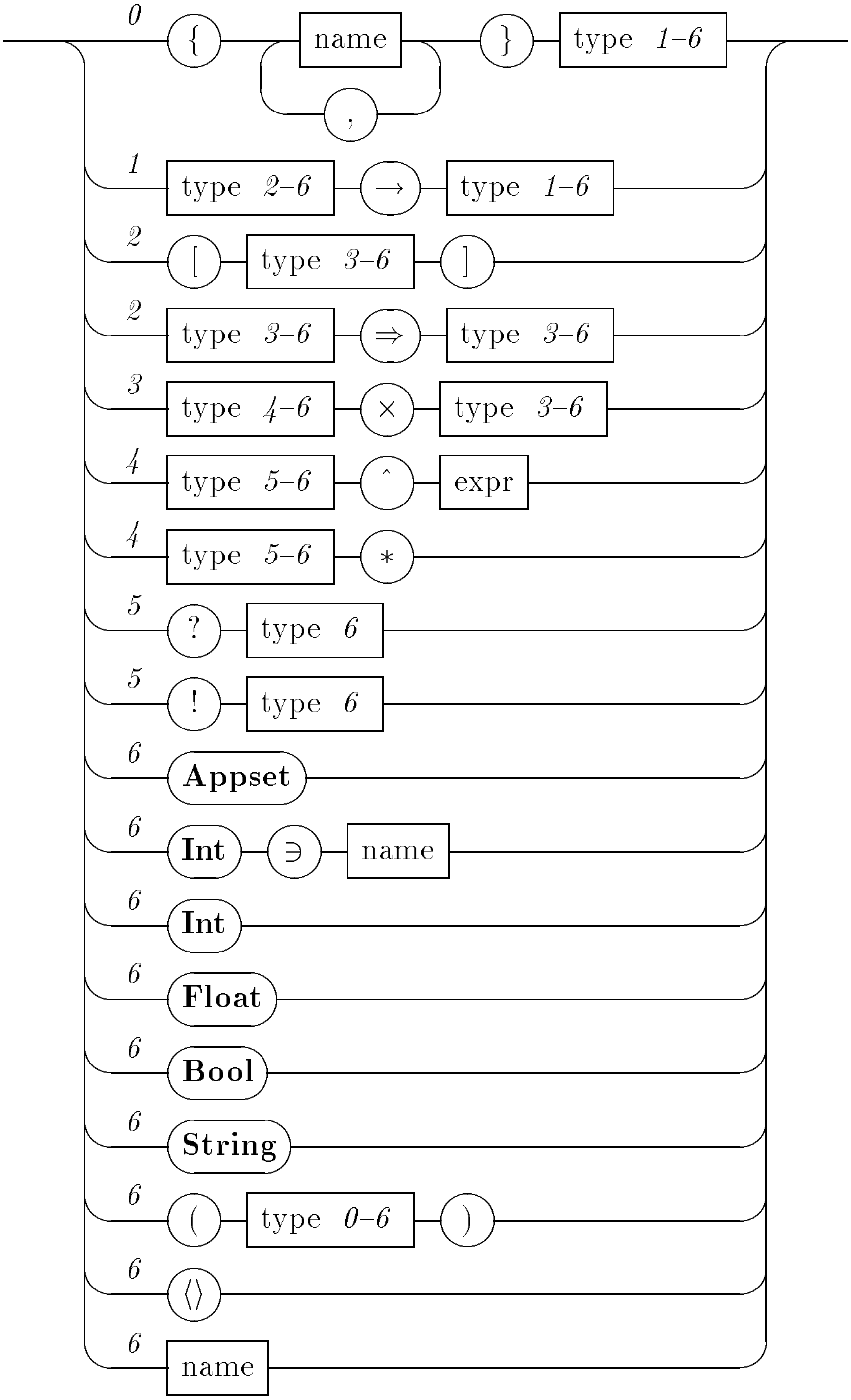
definition



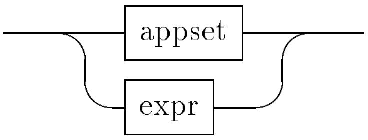
formal



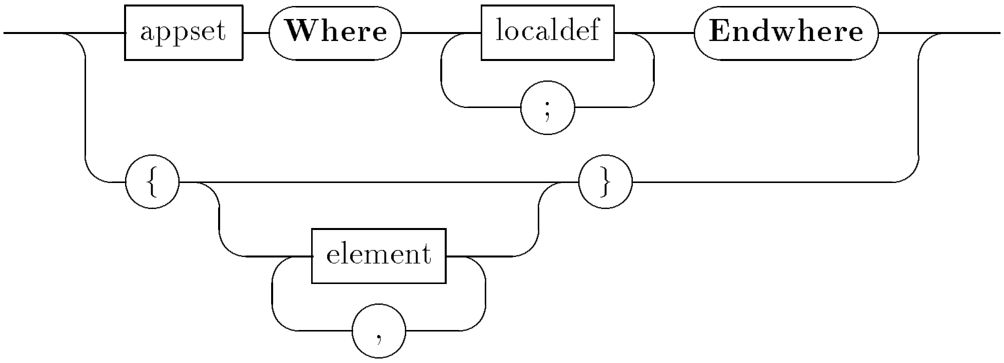
type


0
{
name
,
}
type 1–6
1
type 2–6
→
type 1–6
2
[
type 3–6
]
2
type 3–6
⇒
type 3–6
3
type 4–6
×
type 3–6
4
type 5–6
ˆ
expr
4
type 5–6
*
5
?
type 6
5
!
type 6
6
Appset
6
Int
∋
name
6
Int
6
Float
6
Bool
6
String
6
(
type 0–6
)
6
⟨⟩
6
name

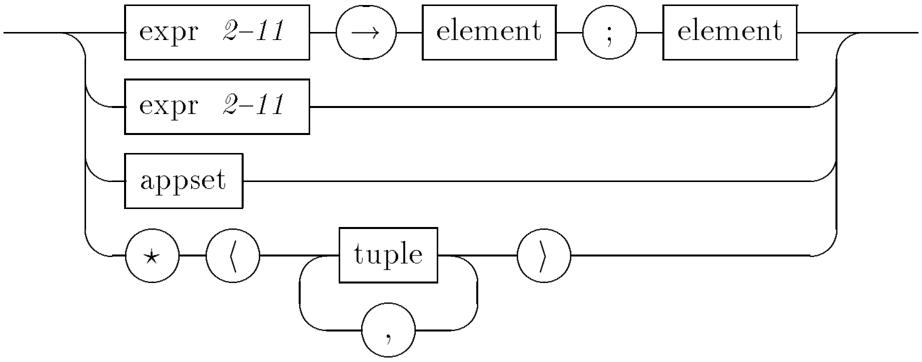
rhs


appset
expr

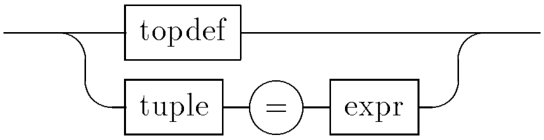
appset



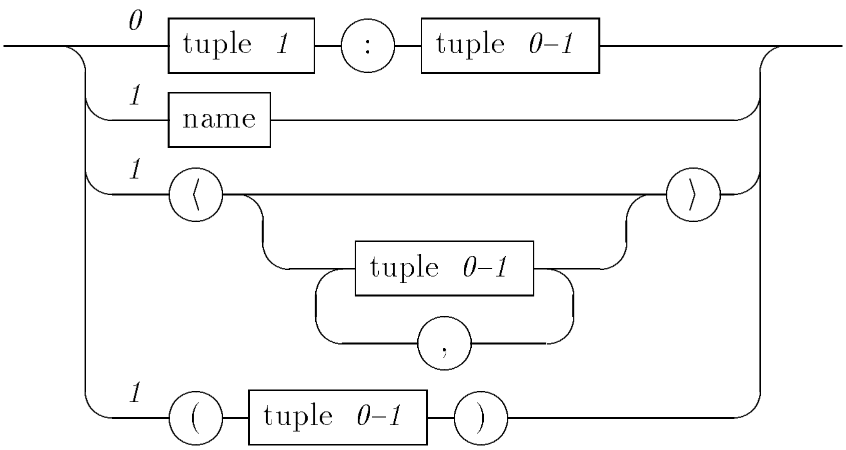
element



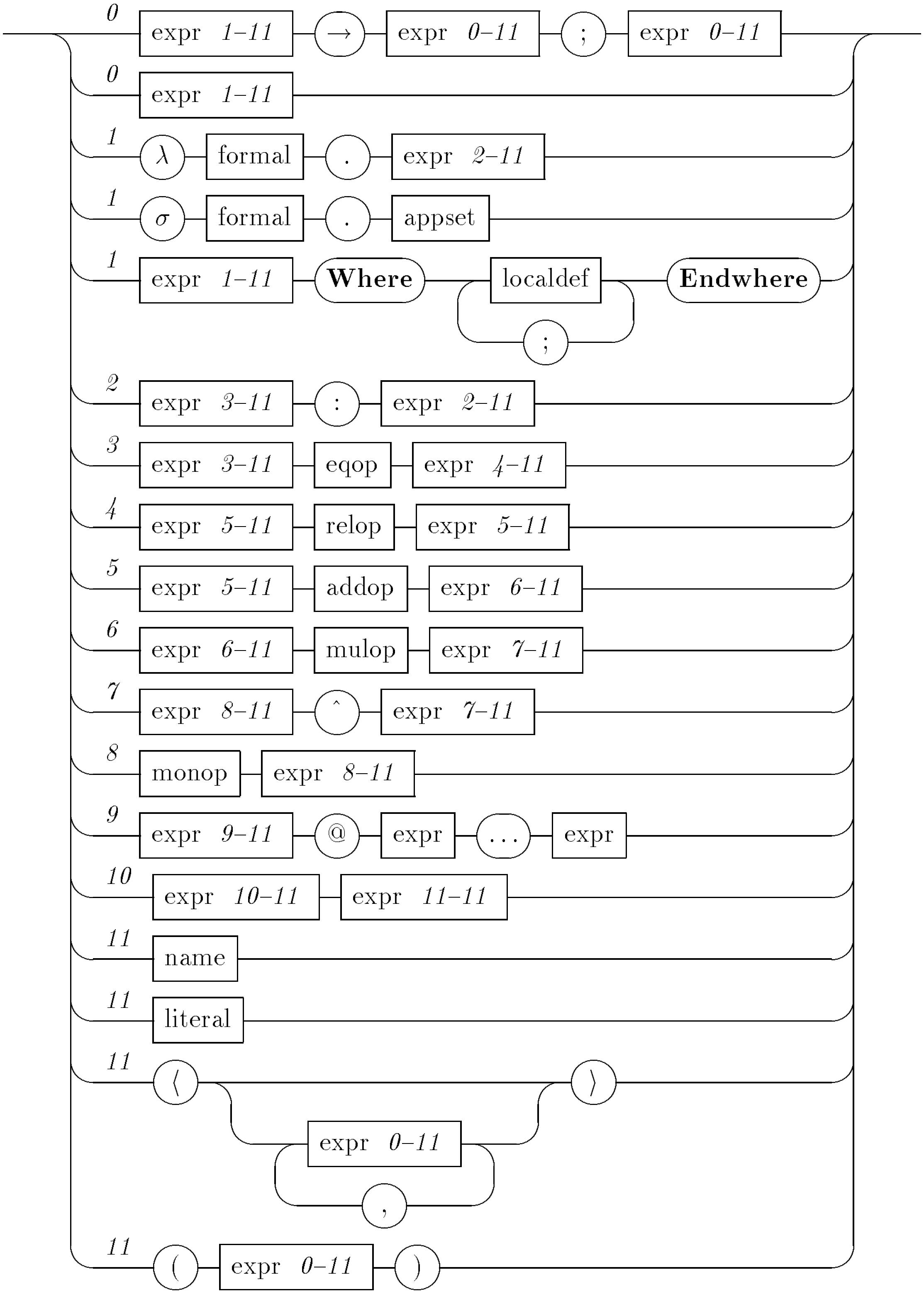
localdef



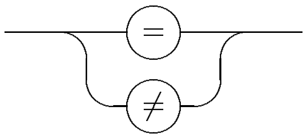
tuple



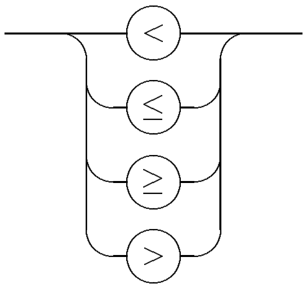
expr


0
expr 1–11
→
expr 0–11
;
expr 0–11
0
expr 1–11
1
λ
formal
.
expr 2–11
1
σ
formal
.
appset
1
expr 1–11
Where
localdef
;
Endwhere
2
expr 3–11
:
expr 2–11
3
expr 3–11
eqop
expr 4–11
4
expr 5–11
relop
expr 5–11
5
expr 5–11
addop
expr 6–11
6
expr 6–11
mulop
expr 7–11
7
expr 8–11
ˆ
expr 7–11
8
monop
expr 8–11
9
expr 9–11
@
expr
...
expr
10
expr 10–11
expr 11–11
11
name
11
literal
11
⟨
expr 0–11
,
⟩
11
(
expr 0–11
)

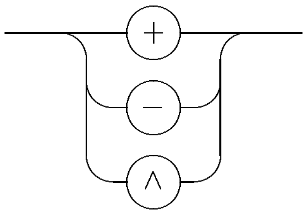
eqop



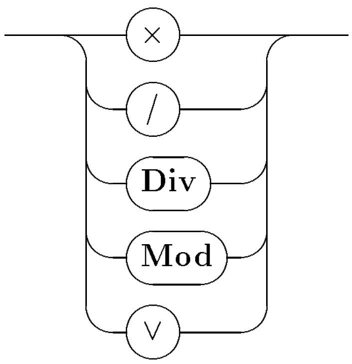
relop



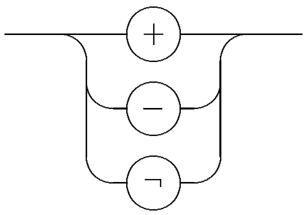
addop



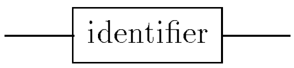
mulop



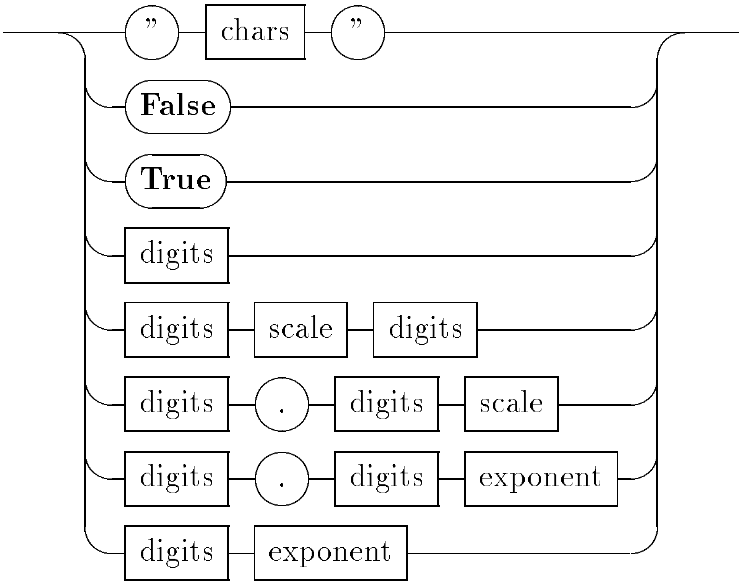
monop



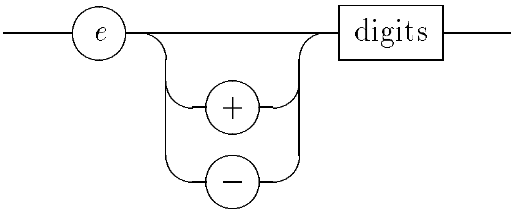
name



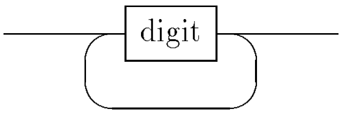
literal



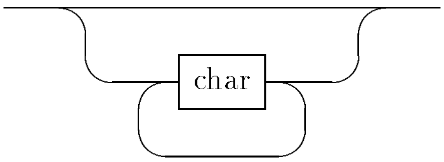
exponent



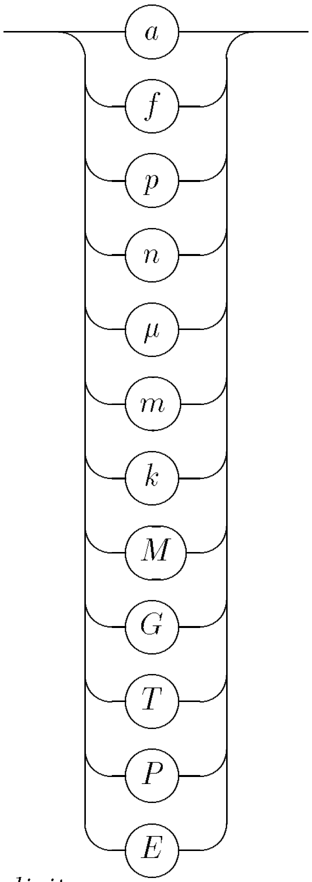
digits



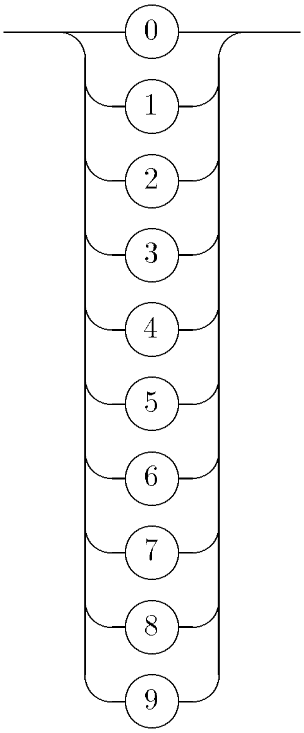
chars



scale



digit



Bibliography

[Bou88] R.T. Boute,
"Systems Semantics: Principles, Applications, and Implementation."
ACM Transactions on Programmimg Languages and Systems,
Vol. 10, No 1, Jan 1988, pp 118-155

[Bou90] R.T. Boute,
Course notes on functional and declarative formalisms,
Nijmegen, Autumn 1990

[Lin80] C.H. Lindsey, S.G. van der Meulen,
Informal introduction to ALGOL 68",
Revised ed., North Holland, Amsterdam, 1980

[Ree90] C. van Reeuwijk (ed.),
"Glass: A system description language and its environment:
Implementation and Maintenance"
Delft, May 1990

[Seu90] M. Seutter (ed.),
"Glass: A system description language and its environment:
Introduction and User manuals"
Nijmegen, May 1990

[Tur79] D.A. Turner,
Sasl language Manual (rev. ed.)
University of Kent, Kent, England, 1979.

[Wij65] A. van Wijngaarden,
"Orthogonal Design and Description of a Formal Language",
Mathematisch Centrum MR 76
Amsterdam, October 1965.

[Wij76] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker,
Revised Report on the Algorithmic Language ALGOL 68,
Springer-Verlag, Berlin, 1976.

Contents