HEWLETT
PACKARD

HP-SL Concrete Syntax

Patrick Goldsack
Software Engineering Department
HP Laboratories Bristol
HPL-91-74
August 1991

Formal specification

Abstract

The report defines the concrete and lexical syntax of the HP-SL specification language developed at the HP research laboratories in Bristol. The report provides the syntax in an extended BNF and the syntactic presentation is supported by some informal commentary.

Contents

1 Introduction

The purpose of this document is to present the complete HP-SL concrete syntax and lexical rules. Both of these are presented in an extended BNF notation which is described in section 2.1

The syntax other than that relating to identifiers is covered in section 2. The syntax for used and defining occurrences of identifiers is given in section 3. Lexical rules are given in section 4.

Following these main sections are a set of indices referencing into the syntax, an index of syntax clauses in section 7, an index of symbols in section 8 and an index of reserved words in section 9.

The index of symbols additionally contains the ASCII form of the mathematical symbols.

Each clause in the syntax is accompanied by a descriptive text. This text is not intended to form a reference manual, rather it is intended to highlight some pertinent information regarding the clause. This information varies considerably in detail from clause to clause, but will (where appropriate) refer the user to a more detailed description in another document.

The report [8] is a rapid overview of HP-SL, and includes an example use of most of the clauses of the grammar.

2 Concrete Syntax

This section defines the concrete syntax of HP-SL.

2.1 Meta-language

The following table defines the meta-language for describing the concrete syntax of HP-SL.

e1 $\wr$ e2	'e1' or 'e2'
$\prec$ e $\succ$	optional 'e'
$\langle e\rangle^*$	zero or more 'e'
$\langle e\rangle^*_s$	zero or more 'e' separated by 's'
$\langle e\rangle^+$	one or more 'e'
$\langle e\rangle^+_s$	one or more 'e' separated by 's'
N	non-terminal symbol 'n'
attr N	non-terminal symbol 'n' with attributes 'attr'
t	terminal symbol 't'
$\langle e\rangle$	grouping of syntactic components

In addition, all non-terminals are alphabetic, so any symbol sequence not defined above may be considered a terminal string. Occasionally, the syntactic description is reduced to English text. When this is done, the intention is always clear.

2.2 Modules

```
1    MODULE-DECL  ::=  module MODULE-ID ≙ MODULE-EXPR

2    MODULE-EXPR  ::=  ( MODULE-EXPR )
.2                 ≀   MODULE-ID
.3                 ≀   define ⟨DEFINITION⟩* enddefine
.4                 ≀   share MODULE-EXPR
.5                 ≀   local ⟨MODULE-DECL⟩* in MODULE-EXPR ⟨MODULE-DECL⟩* endlocal
.6                 ≀   enrich MODULE-EXPR with ⟨DEFINITION⟩* endenrich
.7                 ≀   MODULE-EXPR + MODULE-EXPR
.8                 ≀   sig ⟨DEFINITION⟩* for MODULE-EXPR ≺ fitting ⟨USED-ID → USED-ID⟩+, ≻ endsig
.9                 ≀   replace
                           MODULE-PREFIX by MODULE-EXPR
                           in MODULE-EXPR
                           ≺ fitting ⟨USED-ID → USED-ID⟩+, ≻
                           endreplace
```

Commentary:

1 A module declaration binds the name of a module (which must be unique) to the module denoted by the module expression. Circular dependencies are not allowed between two modules. For more details of the module system in HP-SL see [7].

2 A module expression is the way to construct modules; taking appropriate account of issues such as copying and sharing of types and values.

2.1 The bracketing is to ensure control precedences during parsing and has no other semantic effect.

2.2 The module identifier denotes another module.

2.3 The define module expression is basic building-block of module expressions, by defining a module from a set of definitions not dependent on other modules.

2.4 The share operator allows several modules to share the contents of the module denoted by the expression which follows. The shared module is not allowed to be anonymous.

2.5 The local syntax provides a scoping mechanism for module identifiers; the modules defined within the declaration part defines modules which are local to the following module expression.

2.6 Enrichment allows the addition of definitions to a module.

2.7 Summing two modules combines the definitions of the two modules to create a new one.

2.8 Giving a new signature to a module allows both renaming and hiding of definitions. The fitting morphism is from the new signature to the original signature. It must cover every entity in the new signature, though the language allows the omission of identity mappings.

2.9 The replacement operator allows users to substitute one module for another, thus creating a new module. The fitting morphism is from the original module (all entities whose owner disambiguator starts with the disambiguator provided) to the replacing module. The fitting morphism must cover every entity of the original module, though the language allows the omission of identity mappings.

2.3 Definitions

3	DEFINITION	::=	**val** $\prec$ TYPEVARS $\succ$ PATTERN $\prec \triangleq$ EXPR $\succ$
.2		$\wr$	**fn** $\prec$ TYPEVARS $\succ$ FN-SIG $\langle$**is** FN-BODY$\rangle^*$
.3		$\wr$	**reln** $\prec$ TYPEVARS $\succ$ RELN-SIG $\langle$**is** RELN-BODY$\rangle^*$
.4		$\wr$	**type** $\prec$ TYPEVARS $\succ$ DEFINING-TYPE-ID
			$\prec \triangleq \prec$ **overlapping** $\succ$ $\langle$CONSTRUCTOR$\rangle^+_\mid$ $\succ$
.5		$\wr$	**syntype** $\prec$ TYPEVARS $\succ$ DEFINING-TYPE-ID $\triangleq$ TYPE-EXPR
.6		$\wr$	**assert** $\prec$ TYPEVARS $\succ$ DEFINING-PREFIX-ID $\triangleq$ EXPR
.7		$\wr$	**provable** $\prec$ TYPEVARS $\succ$ DEFINING-PREFIX-ID $\triangleq$ EXPR

4	TYPEVARS	::=	$\llparenthesis$ $\langle$DEFINING-PREFIX-TID$\rangle^+_,$ $\rrparenthesis$

Commentary:

3 The definitions clause lists all the types of definition which may be used at the top level of a flat specification. It includes the declarations of types and values (including functions and relations) as well as the assertions. In all the declarations, the type variables which occur must be pre-declared in the 'typevars'.

3.1 The 'val' definitions declare the identifiers which occur in the pattern. They are bound to the appropriate parts of the expression which must be of a compatible type.

3.2 The 'fn' definition provides a derived syntax for the declaration of functions. The function syntax is further described in [1] and [2].

3.3 The 'reln' definition provides a derived syntax for boolean functions for use in relational modelling. Relations are further described in [1].

3.4 A type definition declares a new type or new type constructor, different from all other existing types or type constructors. The definition also provides a way of including constructor and projector functions by supplying a list of 'constructor'. The 'overlapping' keyword indicates that the 'no confusion' axiom should not be automatically provided. Types are further described in [3] and [6].

3.5 Syntypes define synonyms for existing types and type constructors. Synonym types are further described in [6].

3.6 Assertions are statements that the specification requires to be true. These assertions are used to define abstract behaviour.

3.7 The predicates given as provable are statements of properties of the specification and the assertion must be a consequence of the other definitions and assertions.

4 The definitions of type constructors, polymorphic functions, values or assertions may be generically defined for all types. The type-variable declarations at the start of every definitional form introduces the generic types for use within the definition. For more details on the polymorphism, see [4].

2.4 Constructors

```
5     CONSTRUCTOR  ::=  ...
.2                  ≀   ⟦ DEFINING-PREFIX-ID ⟧
.3                  ≀   ⟦ DEFINING-PREFIX-ID : TYPE-EXPR ⟧
.4                  ≀   ⟦ DEFINING-PREFIX-ID ⟨▷ ⟨PATTERN ≀ ...⟩⟩⁺ ⟧
```

Commentary:

5 A constructor definition is used within a type definition to give the name, signature and some properties of functions and values of the type. These functions are the constructor and projector functions; whilst the properties are those of 'no junk' and 'no confusion'. Associated with each constructor is an 'is_' function to test whether or not a value has been constructed using that constructor. Associated with each projector is a 'set_' function to create a new value which differs only in the value of that projector.

5.1 The three dots are used to indicate that other constructors may be present; it eliminates the 'no junk' axiom.

5.2 A simple identifier indicates a constant value of the type.

5.3 A typed identifier indicates a function which takes a value of the type given with the identifier to the type to whose declaration the constructor contributes.

5.4 The full form of the constructor definition gives both a constructor function and a set of projector functions — one for each of the pattern variables. If more than one of the patterns is used, then the constructor may not be used as a function (its type is not known). It may, however, be used in constructor expressions. The three dot projector indicates that there are more projectors that will be defined later.

2.5 Functions and Relations

6 FN-SIG ::= DEFINING-PREFIX-ID $\prec$: *function* TYPE-EXPR $\succ$

7 FN-BODY ::= HEAD $\prec$ **pre** EXPR $\succ$ $\prec$ **return** PATTERN $\prec$ **post** EXPR $\succ$ $\succ$$\prec$ $\triangleq$ EXPR $\succ$

8 RELN-SIG ::= DEFINING-PREFIX-ID$\prec$: TYPE-EXPR $\succ$

9 RELN-BODY ::= HEAD $\prec$ **pre** EXPR $\succ$ $\prec$ $\triangleq$ EXPR $\succ$

10 HEAD ::= DEFINING-PREFIX-ID $\langle$PARAM-PATTERN$\rangle^+$
.2 $\wr$ PARAM-PATTERN DEFINING-INFIX-ID PARAM-PATTERN
.3 $\wr$ DEFINING-LBRACKET-ID SEQ-PATTERN-BODY DEFINING-RBRACKET-ID

11 PARAM-PATTERN ::= *untyped at outer level* PATTERN

Commentary:

6 A function signature defines the identifier and the type of the function. The type is optional. If it not provided it must be possible to deduce it from the body and procedure parameters. The most general type will be deduced.

7 The body of a function consists of the head — that part of the definition which defines the formal parameters — the pre condition (which is optional) and the definition part. This last part may be left out or may be given explicitly or implicitly, or indeed both. Several bodies may be given for a function each with their own formal parameter patterns and pre-conditions. This gives a clausal definition form for functions.

8 The relation signature gives the name and the types of the various parameters to the relation. Unlike a function, though, the result type (always Bool) must not be given.

9 The body of a relation is constructed in a similar way to that for functions — though the implicit form is not provided.

10 The head of a function or relation is that part of the definition which provides the formal names for the parameters. They are provided in the form of a dummy application of the function or relation with patterns replacing expressions and hence defining the names required for the rest of the function.

10.1 For prefix operators, the head takes on the appearance of a (possibly curried) function application. Thus there is a pattern allowed for each of the curried parameters — though they need not all be supplied (by leaving out the right-most parameters).

10.2 An infix operator is defined by the infix application of the operator between two patterns matching the two expression to which such an operator will be applied.

10.3 A bracket head consists of the left and the right bracket symbols surrounding a sequence pattern matching the sequence to which the brackets will be applied.

11 Each formal parameter of a function is given as a pattern. These patterns may not be typed unless both the pattern and the associated typing are within parentheses.

2.6 Type Expressions

```
12    TYPE-EXPR  ::=  let ⟨DEFINITION⟩* in TYPE-EXPR ⟨DEFINITION⟩* endlet
 .2              ≀    ( TYPE-EXPR )
 .3              ≀    USED-TYPE-ID
 .4              ≀    [ TYPE-EXPR ]
 .5              ≀    TYPE-EXPR inv PATTERN . EXPR
 .6              ≀    CONSTRUCTOR-INSTANTIATION
 .7              ≀    ...
```

```
13    CONSTRUCTOR-INSTANTIATION  ::=  TYPE-EXPR TYPE-EXPR
 .2                              ≀    TYPE-EXPR USED-INFIX-TYPE-ID TYPE-EXPR
```

Commentary:

12 Type expressions are used to denote types. They are 'referentially transparent'; thus two identical expressions denote the same type if they occur in the same environment. For more information on type expressions, see [6].

12.1 A let construction creates a local scope within which new definitions may be given and then used within the type expression between the 'in' and 'endlet'.

12.2 Parentheses are used to control the binding and precedence rules. They have no effect on the meaning of type expression within the parentheses.

12.3 The identifier must be the identifier of a synonym type, a type variable or a type which is within scope. If it is a synonym identifier, the semantics may be seen as that of the definition of the synonym being 'unfolded' at the point of use in an environment identical to that at the point of definition. Thus names in a synonym refer to their meanings at the time of the definition, not at the time of use.

12.4 An optional type expression denotes the type of the internal expression extended by the value 'nil'. There is such a 'nil' for every type.

12.5 A subtype expression creates a type whose values are members of the parent type, but only those that satisfy the predicate expression are members of the subtype. The pattern is used to provide names for the components of a typical member of the type for use in the predicate.

12.6 Type constructors are type functions which define types from other types. The instantiation of such a constructor is the way in which a type constructor is applied to a type (represented by a type expression).

12.7 The three dot notation may be used to indicate that the type has yet to be written. This is useful in giving partial specifications and examples.

13 Constructor instantiations are the way in which type constructors are applied to their actual type parameters.

13.1 Most type constructors, including all user defined constructors, are prefix constructors. These are applied to their parameters by juxtaposition; by placing the constructor infront of the type expression. Type constructors with more than one type parameter are curried.

13.2 A few of the predefined constructors are written using an infix notation. The application is written by placing the constructor between the two type parameter expressions.

2.7 Patterns

```
14   PATTERN  ::=  _
 .2             ≀  DEFINING-PREFIX-ID
 .3             ≀  PATTERN as PATTERN
 .4             ≀  TYPED-PATTERN
 .5             ≀  ( PATTERN )
 .6             ≀  ( PATTERN , ⟨PATTERN⟩+, )
 .7             ≀  ⟦ USED-PREFIX-ID ⟨▷ PATTERN ≺ ≜ USED-PREFIX-ID ≻⟩* ⟧
 .8             ≀  ≪ SEQ-PATTERN-BODY ≫
```

Commentary:

14 A pattern is used to name 'typical' members of a type inside a local scope. They are also used to simultaneously name sub-components without the use of projector functions. This enables very compact and readable definitions. In most cases, patterns closely follow the structure of the expressions that build the values. Patterns may be nested to arbitrary depths.

14.1 A don't care pattern matches any value, but does not supply a name for it. It is most useful when only one sub-component of a compound value (such as a tuple) is required and it seems unnecessary to invent names for the other parts.

14.2 An identifier is used to name the whole expression to which it is matched.

14.3 Sometimes it is useful to name components of values in different ways; for example when a name is required for the whole of a value and for sub-components. This is achieved by layering the pattern. Both the left and the right pattern are matched to the same value and both sets of name bindings which occur are visible.

14.4 At all times it is necessary to know the type of the variables within a pattern. This may be done either by typing the whole of the pattern or by allowing types to occur on the component sub-patterns.

14.5 Parentheses are used to control precedence during parsing and have no effect on the pattern matching.

14.6 A tuple pattern has a sub-pattern for each component of the tuple value against which it is matched. The matching is carried out pairwise — the first pattern matched against the component of the tuple value and so on. A tuple pattern only matches a tuple value with the same number of components.

14.7 A constructor pattern may be used to match a value of a type declared using a union of constructor definitions. The pattern will only match values constructed using the constructor mentioned as the first identifier. Subsequent patterns may be given, and associated with the identifier of one of the projectors for that constructor function. The component pattern is then matched against the value returned by that projector when applied to the whole value to be matched. If no projector is mentioned, the whole constructed value is matched. If the keyword 'overlapping' was used in the type definition then the pattern matching is not allowed. For more information on type patterns see [3].

14.8 A sequence pattern matches both elements from the front of the list and elements from the back of the list. This is explained more with the sequence pattern body clause (clause 16.1).

Patterns (cont.)

15	TYPED-PATTERN	::=	PATTERN : TYPE-EXPR
.2		≀	$\langle$PATTERN **and** $\langle$PATTERN$\rangle^{+}_{\textbf{and}}\rangle$
.3		≀	PATTERN **sat** EXPR
.4		≀	PATTERN USED-INFIX-ID EXPR
.5		≀	NUMERIC-LITERAL
.6		≀	CHAR-LITERAL
.7		≀	STRING-LITERAL

16 SEQ-PATTERN-BODY ::= $\langle$PATTERN ≀ < PATTERN >$\rangle^{*}_{,}$

Commentary:

15 Every pattern must be typed. However, sometimes it is convenient to give a set of 'short forms' for a simple typing. The following set of clauses are all of this nature — they all are explained in terms of a short-hand for a longer typed pattern.

15.1 The simple form of typing has the obvious meaning — the pattern may only be matched against a member of the type. For this to be valid, the pattern must also be of the right form — for example a tuple pattern may not be given a sequence type.

15.2 The and form of pattern definition is equivalent to a tuple where every element is of the same type.

15.3 The 'sat' form of pattern represents a sub-typing. The value to which the pattern is matched must satisfy the predicate expression.

15.4 The relational form of pattern is a special case of the 'sat' pattern where the predicate is the application of a binary relational operator. In this case the pattern must be in the appropriate relation to the value denoted by the expression.

15.5, 15.6, 15.7 It is possible to use the built in literal forms as patterns. They match only if equal to the value to which they are matched. The boolean literals should be treated as members of an enumerated type.

16 A sequence may be matched against a sequence pattern. A pattern consists of a possibly empty sequence of patterns matching elements from the front of the sequence value; a sequence of patterns matching elements from the back of the sequence; a pattern matching the rest of the sequence — itself a sequence. The syntax is more generous than is in fact allowed as it appears that any number of such sub-sequence patterns is allowed. A sequence pattern only matches with sequences that are sufficiently long to match every element mentioned in the pattern — though the 'rest of the sequence' part may be matched against an empty sequence.

2.8 Expressions

17	EXPR	::=	**let** $\langle$DEFINITION$\rangle^*$ **in** EXPR $\langle$DEFINITION$\rangle^*$ **endlet**
.2		≀	(EXPR $\prec$: TYPE-EXPR $\succ$)
.3		≀	NUMERIC-LITERAL
.4		≀	CHAR-LITERAL
.5		≀	STRING-LITERAL
.6		≀	USED-PREFIX-ID
.7		≀	APPLICATION
.8		≀	($\langle$EXPR$\rangle^+_,$)
.9		≀	CONSTRUCTOR-EXPR
.10		≀	**if** EXPR **then** EXPR $\langle$**elseif** EXPR **then** EXPR$\rangle^*$ **else** EXPR **endif**
.11		≀	**cases** EXPR **of** $\langle$**case** PATTERN **then** EXPR$\rangle^+$ $\prec$ **else** EXPR $\succ$ **endcases**
.12		≀	($\forall$ PATTERN . EXPR $\prec$ **end**$_\forall$ $\succ$)
.13		≀	($\exists$ PATTERN . EXPR $\prec$ **end**$_\exists$ $\succ$)
.14		≀	($\exists!$ PATTERN . EXPR $\prec$ **end**$_{\exists!}$ $\succ$)
.15		≀	(λ PATTERN . EXPR $\prec$ **end**$_\lambda$ $\succ$)
.16		≀	...

Commentary:

17 Expressions are used to denote values of particular types. Expressions fall into several classes. The majority are those which construct values directly, others control scope and parsing, yet more are used in predicates to give properties of the values thus defining them indirectly. This richness makes for a large expression language in comparison to programming languages.

17.1 Let expressions are used to control scope, with the definitions provided between the 'let' and 'endlet' being only visible within the main expression between 'in' and 'endlet' or a following definition. The result of the 'let' expression is that of the expression following the 'in'. It is illegal for this expression to be of a type which is defined within the local definitions.

17.2 Parentheses are used only for controlling parsing and have no other effect upon the values of the expressions which they contain. To avoid parsing problems with the range of a type expression, both in terms of the expression it is typing and with respect to the end of any invariant that may be involved, typings may only be included within a parenthesised expression. The typing operates on the whole expression occurring within the parentheses. The result of the parenthesised expression is the result of the expression within the parentheses.

17.3 - 17.5 Numeric, character and string literals are, along with identifiers, the basic building blocks of expression

17.6 Identifiers denote the values to which they were bound in their declaration.

17.7 There are three types of operator whose application needs to be considered. These are described further in the commentary for clause 18.

17.8 Cartesian expressions build values of the tuple types, they are distinguished from the parenthesised expressions by the occurrence of one or more comma symbols.

17.9 Constructor expressions are used to define a value of an abstract type indirectly by stating what the projectors for that type should produce for the value. More detail of the precise syntax is given in the description for clause 20. See also [3].

17.10 'If' expressions are provided to give the usual conditional control in expressions. An arbitrary number of 'elseif' clauses may be provided. As the conditional is an expression, an 'else' must always be present. All of the condition expressions (before a 'then') must be boolean. All of the result expressions (following and 'then' or 'else') must have the same type. The 'if' expression has the same type as the result expressions, and its value is that of the first alternative with a valid guard.

17.11 The 'cases' expression provides control based on pattern-matching, particularly useful in conjunction with enumerations, sequences and the like. The variables defined in a particular case pattern are visible only within the expression following the 'then' of that case. When a value is matched against the patterns, the resultant case-expression value is that of the case alternative which matches. In the instance that more than one case matches, the results should be identical otherwise the specification is ill-formed. Each of the alternatives must have the same type.

17.12 The universal quantifier expression is used to test all values of the pattern variables in the predicate expression following the '·'. The pattern variables are visible within that predicate which forms an inner scope. The ending keyword is optional to allow compact one-line quantifications and yet to allow a greater degree of readability when the quantification runs over several lines. The parentheses are necessary to limit the extent of the quantified expression.

17.13 The existential quantifier, used to test whether one of the legal pattern variables satisfies the following predicate expression. In all other respects, the syntax is identical to that for the universal quantifier, clause 17.12.

17.14 The quantifier which is used to test whether exactly one of the legal pattern variables satisfies the following predicate expression. In all other respects, the syntax is identical to that for the universal quantifier, clause 17.12.

17.15 The lambda-expression is used to build values of function type. The pattern immediately following the lambda represents the formal parameters. These may then be used within the body of the function, the expression which follows the '·'. On application of the function, the pattern is matched against the actual parameter and the pattern variables are bound to the appropriate values. The resultant value of the application is the value of the function body under the appropriate variable bindings. The parentheses are necessary to limit the extent of the body of the function and the ending keyword is optional to allow compact one-line expressions and to aid readability when the function spreads over several lines.

17.16 The undefined expression indicates an expression which has not yet been completed.

Expressions (cont.)

```
18      APPLICATION  ::=  EXPR EXPR
 .2                  ≀    EXPR USED-INFIX-ID EXPR
 .3                  ≀    USED-LBRACKET-ID BRACKET-BODY USED-RBRACKET-ID

19      BRACKET-BODY  ::=  EXPR .. EXPR
 .2                   ≀    ⟨EXPR ≀ ⩿ EXPR ⪀⟩*, ≺ ⟨| PATTERN ≺ ϵ EXPR ≻⟩+ ≺ . EXPR ≻ ≻

20      CONSTRUCTOR-EXPR  ::=  ⟦ USED-PREFIX-ID ⟨▷ USED-PREFIX-ID ≜ EXPR⟩* ⟧
```

Commentary:

18 Applications come in three flavours, prefix application, infix application and bracket application — this latter form being further described in the clause 19.

18.1 Prefix application is achieved by juxtaposition of the function and the parameter, function first. Application binds to the left, so curried functions may be applied to several parameters by repeated juxtaposition without parentheses.

18.2 Infix application is achieved by placing the operator between the first and second values of the pair on which the operators are defined.

18.3 Bracket application is achieved by placing the opening (left) bracket and the closing (right) bracket around the collection to which they will be applied.

19 A bracket body is the way in which collections of elements are constructed for the bracket operators. The syntax allows enumerations, comprehensions, and integer ranges. The syntax is very rich and hence somewhat complicated.

19.1 This is one of the two main forms for applying the brackets to collections of integers, the integers in order between the two integers in the clause. If the first integer value is greater than the second, then the collection is empty.

19.2 The syntax for enumerations and comprehensions is described in [2].

20 These expressions are used to give values of abstract types defined in terms of the constructor/projector syntax an indirect form of definition via the projected values. In this form of the expression, the constructor name is the first identifer after the opening bracket. It is not called directly with ordered parameters, but rather each of the subsequent '▷' symbols is followed by a projector names bound to the value which it should project. The projectors need not occur in any particular order. It is a language feature very like a record syntax, with field selection being achieved by the use of the projector functions. The value denoted by the expression is one of the values of the associated abstract type, such that the projectors applied to that value return the values to which they are bound.

3 Identifiers

There are several kinds of identifier

1. module identifiers
2. type and type constructor identifiers
3. value, function, relation, assertion and provable identifiers, known collectively as prefix identifiers
4. infix operator identifiers
5. infix type constructor identifiers, though these are all built-in
6. bracket operator identifiers, both left and right bracket identifiers

Each of these has a different lexical form (except for modules and types which share their form). Furthermore, each (other than the module and right bracket identifiers) come in two flavours, used and defined occurrences. Syntactically, this difference is reflected in the fact that a module disambiguator is allowed with the used occurrences. Modules do not have a disambiguator, their names must be unique. Right bracket identifiers are disambiguated on the use of the matching left bracket identifier.

A further difference between used and defining occurrences, this time including the right bracket identifiers, is that the HP-SL predefined identifiers are only allowed in the used identifier positions and need never be disambiguated.

A module disambiguator consists of a sequence of module names referring to the owning module of that identifier. Each identifier must be unique within that owning module — HP-SL does not support overloading as overloading conflicts with the parametric polymorphism.

```
21      MODULE-ID  ::=  TID


22      MODULE-PREFIX  ::=  ⟨TID⟩*⁀


23      USED-PREFIX-ID  ::=  ID ≺ ⁀ MODULE-PREFIX ≻
 .2                      ≀    ′USED-INFIX-ID
 .3                      ≀    ′USED-LBRACKET-ID
 .4                      ≀    ‘predefined prefix operator’


24      DEFINING-PREFIX-ID  ::=  ID
```

.2 ≀ ′ DEFINING-INFIX-ID
.3 ≀ ′ DEFINING-LBRACKET-ID

25 USED-INFIX-ID ::= INID ≺ ⌢ MODULE-PREFIX ≻
.2 ≀ ′ USED-PREFIX-ID
.3 ≀ **‘predefined infix operator’**

26 DEFINING-INFIX-ID ::= INID

27 USED-TYPE-ID ::= TID ≺ ⌢ MODULE-PREFIX ≻

28 DEFINING-TYPE-ID ::= TID
.2 ≀ ′ DEFINING-INFIX-TYPE-ID

29 USED-INFIX-TYPE-ID ::= **‘pre-defined infix type id’**

30 DEFINING-INFIX-TYPE-ID ::= **‘pre-defined infix type id’**

31 USED-LBRACKET-ID ::= LBRACKET ≺ ⌢ MODULE-PREFIX ≻
.2 ≀ **‘pre-defined left bracket id’**

32 DEFINING-LBRACKET-ID ::= LBRACKET

33 USED-RBRACKET-ID ::= RBRACKET
.2 ≀ **‘pre-defined right bracket id’**

```
34      DEFINING-RBRACKET-ID  ::=  RBRACKET


35      USED-ID  ::=  USED-PREFIX-ID
 .2                |  USED-TYPE-ID
```

4 Lexical Rules

In the concrete syntax above, the syntax is described in terms of the mathematical syntax and not the ASCII syntax. The lexical rules, however, are never applied to input in the mathematical syntax; all input is naturally in the ASCII syntax. It makes sense, therefore, to only give rules for ASCII input.

The lexical analysis attempts to build the longest possible tokens, reading left to right — lexical analysis never breaks a token unnecessarily.

The lexical classes are divided into two main groups — the sign characters and the alphanumerics (which include the $'$ character). Characters of the same class will stick together to form groups.

The '_' character will link groups of characters together. If the characters in both groups are from the same class, then the class of the combined group is that of the components. If one of the groups is sign and the other is alphanumeric, the combined class is that of the first group.

Parenthesis characters never stick to anything unless part of a reserved word.

In addition to the general rules, there are a set of reserved words and pre-defined operator symbols. These override the standard rules *if* the standard rules would result in a shorter token.

With these meta-rules in mind, here is an informal set of lexical rules; the formal ones are given in the following sub-sections.

- *reserved words* — if a token matches a reserved word, then this takes preference over any other type of token
- *identifiers* — any linked set of groups such that the first (or only) group starts with a lower-case character

  For example

  a
  $d4$
  $john's\_book$
  xyz'
  x_**

  are all identifiers
- *numeric literals* — any sequence of characters starting with a numeric character and containing only numeric and upper-case letters characters and at most a single occurrence of 'b', '.' and 'e' in that order.

  For example

  1
  2b1101
  1.34334
  0.01e20
  16b4F.FeA0

are all numeric literals

- *operators* — there are two possibilities

  1. *infix operators* Any linked set of character groups, such that the first (or only) group consists of sign characters. Bracket characters must be balanced.
     For example

     ++
     +_*complex*
     {*}

     are all infix operators

  2. *bracket operators* – bracket operators are considered either 'left' or 'right' operators. There is the concept of matching of a left bracket operator with a right bracket operator; this being defined at the lexical level to ensure that parsing may continue without the declaration of the operator.
     A bracket operator is an infix operator with at least one non-matching bracket character — {, }, [,] — included in the first (or only) group. All the unmatched bracket symbols must be of the same directionality (i.e. both left or right symbols in the same token is illegal). The directionality of the bracket operator is the same as that of the included bracket symbols.
     For example

     {*
     [{

     are both left brackets, and

     ]

     is a right bracket.
     To understand left/right mirroring, consider the following examples

     {* *}
     [_*history*]_*history*

     These are all matching bracket operators. The rules are as follows:

     (a) the first group is mirrored, with bracket characters being swapped for their respective matching pair
     (b) the remaining groups are left untouched
     (c) the groups remain in the same order

     In addition, the following two tokens are considered matching bracket operators

     << >>

- *type identifiers* — any linked set of groups such that the first (or only) group starts with an upper-case character

  For example

Colour
New_type
Type_10
*T_***

are all type identifiers

4.1 Character classes

36 LOWER-CASE ::= a–z

37 UPPER-CASE ::= A–Z

38 DIGIT ::= 0–9

39 CHARACTER ::= any printable character except a " ' or \
.2 ≀ \ [LOWER-CASE ≀ UPPER-CASE ≀ DIGIT]

40 ALPHANUM ::= ⟨' ≀ LOWER-CASE ≀ UPPER-CASE ≀ DIGIT⟩$^{+}$

41 SIGN-CHAR ::= any of ! @ # $ % & * − = + \ { } [ ]| ? / > <

42 SIGN ::= ⟨SIGN-CHAR⟩$^{+}$

43 SIGN-AND-ALPHA ::= _ ⟨*bracket-balanced* SIGN ≀ ALPHANUM⟩$^{+}_{\_}$

4.2 Tokens

The following disambiguation rules apply to the tokenisation

- the longest token is taken in preference
- given two productions which produce tokens of the same length, the earlier production is preferred

44 NUMERIC-LITERAL ::= $\prec \langle \text{DIGIT} \rangle^{*}\text{b} \succ \langle \text{DIGIT} \rangle^{+} \prec .\langle \text{DIGIT} \rangle^{+} \succ \prec \text{e}\langle \text{DIGIT} \rangle^{+} \succ$

45 CHAR-LITERAL ::= ‘CHARACTER’

46 STRING-LITERAL ::= “$\langle$CHARACTER$\rangle^{*}$”

47 ID ::= *first character lower-case* ALPHANUM SIGN-AND-ALPHA

48 TID ::= *first character upper-case* ALPHANUM SIGN-AND-ALPHA

49 INID ::= *bracket-balanced* SIGN SIGN-AND-ALPHA

50 LBRACKET ::= *at least one left and no right bracket characters* SIGN SIGN-AND-ALPHA

51 RBRACKET ::= *at least one right and no left bracket characters* SIGN SIGN-AND-ALPHA

4.3 Comments

The dialect supports comments of the ‘C’ style, by delimiting the comment text between ‘/∗’ and ‘∗/’. It is required that the comment start token normally be surrounded by white-space characters. Comment start tokens which are immediately followed by a character other than white space are reserved for use by the tools and unexpected behaviour might occur if users do not follow these guidelines. Nesting of comments is not supported.

5 ASCII equivalents

The following table describes the ASCII equivalents to the symbols included in this syntax. ASCII based tools will be defined to use these equivalents — one of these tools is intended to be a translator from the ASCII form to LaTeX symbolic for printing within documents.

symbol	ASCII
$\forall$, end$_{\forall}$	FORALL, ENDFORALL
$\exists$, end$_{\exists}$	EXISTS, ENDEXISTS
$\exists!$, end$_{\exists!}$	EXISTS!, ENDEXISTS!
λ, end$_{\lambda}$	LAMBDA, ENDLAMBDA
$\times$	X
$\ll, \gg$	<<, >>
$\lessdot$, $\gtrdot$	<., .>
$(\!\vert$, $\vert\!)$	(|, |)
$[\![$, $]\!]$	[[,]]
$\triangleq$	==
$\rightarrow$	->

6 Associativity and Precedence

This section describes the relative precedences and associativity of operators within the dialect. It includes more than just the built-in expression operators such as those for arithmetic, it also includes those for types, patterns and any other feature whose binding one must know.

For simplicity, the associativity of the individual expression operators are bundled into a single level. With respect to the rest of the language this is sufficient. However each has a separate associativity and precedence and for a full definition of these, see [5].

Tightly binding

operator	associativity
⁀	left
all unary prefix operators, application and indexing	left
all infix binary operators	see [5]
.	non
sat inv	left
as	left
and	flattening
:	left

Weakly binding

7 Index Of Clauses

8 Index Of Symbols

9 Index Of Reserved Words

10 References

[1] Patrick Goldsack. HP-SL functions and relations. Technical report, Hewlett-Packard Laboratories. In production.

[2] Patrick Goldsack. HP-SL operator syntax. Technical report, Hewlett-Packard Laboratories. In production.

[3] Patrick Goldsack. HP-SL abstract type syntax (records and unions). Technical Report HPL-91-69, Hewlett-Packard Laboratories, Bristol, June 1991.

[4] Patrick Goldsack. The HP-SL model of polymorphism — an informal description. Technical Report HPL-91-71, Hewlett-Packard Laboratories, Bristol, June 1991.

[5] Patrick Goldsack. HP-SL pre-defined types. Technical Report HPL-91-72, Hewlett-Packard Laboratories, Bristol, June 1991.

[6] Patrick Goldsack. The HP-SL type model. Technical Report HPL-91-73, Hewlett-Packard Laboratories, Bristol, June 1991.

[7] Patrick Goldsack. Module combinators for specification languages. Technical Report HPL-91-70, Hewlett-Packard Laboratories, Bristol, 1991. June.

[8] Patrick Goldsack. A tour of HP-SL. Technical Report HPL-91-68, Hewlett-Packard Laboratories, Bristol, June 1991.