Links Grammar

Contents

1 Grammar for patterns

pattern	*::=*	*as-pattern* ⟨ `:` *primary-datatype* ⟩
as-pattern	*::=*	*cons-pattern* ⟨ `as VARIABLE` ⟩
cons-pattern	*::=*	*constructor-pattern* ⟨ `::` *cons-pattern* ⟩
constructor-pattern	*::=*	*primary-pattern* `CONSTRUCTOR` ⟨ *parenthesized-pattern* ⟩
parenthesized-pattern	*::=*	`(` ⟨ *pattern* ⟨ `,` *patterns* ⟩⟩ `)` `(` *labeled-patterns* ⟨ `|` *pattern* ⟩ `)`

primary-pattern	*::=*	`VARIABLE`
		`_`
		special-constant
		`[` ⟨ *patterns* ⟩ `]`
		parenthesized-pattern

patterns	*::=*	*pattern* ⟨ `,` *patterns* ⟩
labeled-patterns	*::=*	*record-label* = *pattern* ⟨ , *labeled-patterns* ⟩

2 Grammar for types

datatype	*::=*	*mu-datatype* ⟨ `->` *datatype* ⟩
		mu-datatype ⟨ `-{` *datatype* `}->` *datatype* ⟩
mu-datatype	*::=*	`mu` `VARIABLE` `.` *mu-datatype*
		primary-datatype
primary-datatype	*::=*	`(` *datatype* `)`
		`(` ⟨ *datatype* `,` *datatypes* ⟩ `)`
		`(` *fields* `)`
		`{` `VARIABLE` `}`
		TableHandle `(` *datatype* `,` *datatype* `)`
		`[[` ⟨ *vfields* ⟩ `]]`
		`[` *datatype* `]`
		`VARIABLE`
		`QUOTE` `VARIABLE`
		`CONSTRUCTOR` (⟨ *primary-datatype-list* ⟩)
primary-datatype-list	*::=*	*primary-datatype* ⟨ `,` *primary-datatype-list* ⟩
vfields	*::=*	`VARIABLE`
		`CONSTRUCTOR` ⟨ `:` *spec* ⟩ ⟨ `|` *vfields* ⟩
datatypes	*::=*	*datatype* ⟨ `,` *datatypes* ⟩
tfields	*::=*	*fields*
		`VARIABLE`
fields	*::=*	*field* ⟨ `|` `VARIABLE` ⟩
		field `,` *fields*
spec	*::=*	`-`
		datatype
field	*::=*	*record-label* `:` *spec*

3 Grammar for regular expressions

regex	*::=*	`/` ⟨ *regex-pattern-sequence* ⟩ `/`
regex-pattern	*::=*	`[` `REGEXCHAR` `-` `REGEXCHAR` `]`
		`.`
		`REGEXCHAR`
		`(` *regex-pattern-sequence* `)`
		regex-pattern repeat-op
		block

repeat-op ::= `*`
`+`
`?`

regex-pattern-sequence ::= *regex-pattern* ⟨ *regex-pattern-sequence* ⟩

4 Grammar for expressions

location ::= `server`
`client`
`native`

special-constant ::= `UINTEGER`
`UFLOAT`
`true`
`false`
`STRING`
`CHAR`

atomic-expression ::= `VARIABLE`
special-constant
parenthesized-expression

primary-expression ::= *atomic-expression*
`[` ⟨ *exps* ⟩ `]`
xml-forest
`fun` *arg-list block*

constructor-expression ::= `CONSTRUCTOR` ⟨ *parenthesized-expression* ⟩

parenthesized-expression ::= `(` *binop* `)`
`(` `.` *record-label* `)`
`(` *exps* `)`
`(` *exp* `with` *labeled-exps* `)`
`(` *labeled-exps* ⟨ | *exp* ⟩ `)`

binop ::= `-`
`-.`
infix-op

infix-op ::= `` ` `` `VARIABLE` `` ` ``
`SYMBOL`

postfix-expression ::= *primary-expression*
block
`query` *block*

		`spawn` *block*
		`spawnWait` *block*
		postfix-expression `.` *record-label*
		postfix-expression `(` ⟨ *exps* ⟩ `)`
exps	::=	*exp* ⟨ `,` *exps* ⟩
unary-expression	::=	`-` *unary-expression*
		`-.` *unary-expression*
		postfix-expression
		constructor-expression
infix-expression	::=	*unary-expression*
		infix-expression infix-op infix-expression
		infix-expression `~` *regex*
logical-expression	::=	*infix-expression*
		logical-expression `||` *infix-expression*
		logical-expression `&&` *infix-expression*
typed-expression	::=	*logical-expression* ⟨ `:` *datatype* ⟩
db-expression	::=	*typed-expression*
		`insert` *exp* `values` *exp*
		`delete (` *table-generator* `)` ⟨ *where* ⟩
		`update (` *table-generator* `)` ⟨ *where* ⟩ `set (` *labeled-exps* `)`
where	::=	`where (` *exp* `)`
xml-forest	::=	*xml-tree*
		xml-tree xml-forest
xmlid	::=	`VARIABLE`
attr-list	::=	*xmlid* `="` ⟨ *attr-val* ⟩ `"` ⟨ *attr-list* ⟩
attr-val	::=	*block* ⟨ *attr-val* ⟩
		string ⟨ *attr-val* ⟩
xml-tree	::=	`<` `QNAME` ⟨ *attr-list* ⟩ `/>`
		`<` `QNAME` ⟨ *attr-list* ⟩ `>` ⟨ *xml-contents-list* ⟩ `</` `QNAME` `>`
xml-contents-list	::=	*xml-contents* ⟨ *xml-contents-list* ⟩
xml-contents	::=	*block*
		`CDATA`
		`{` *logical-expression* `->` *pattern* `}`
		xml-tree

conditional-expression ::= *db-expression*
`if (` *exp* `)` *exp* `else` *exp*

cases ::= `case` *pattern* `->` *block-contents* ⟨ *cases* ⟩

case-expression ::= *conditional-expression*
`switch (` *exp* `) {` ⟨ *cases* ⟩ `}`
`receive {` ⟨ *cases* ⟩ `}`

iteration-expression ::= *case-expression*
`for (` *generator* `)` ⟨ *where* ⟩ ⟨ `orderby (` *exp* `)` ⟩ *exp*

generator ::= *list-generator*
table-generator

list-generator ::= *pattern* `<-` *exp*
table-generator ::= *pattern* `<--` *exp*

escape-expression ::= *iteration-expression*
`escape VARIABLE in` *postfix-expression*

formlet-expression ::= *escape-expression*
`formlet` *xml-contents-list* `yields` *exp*

table-expression ::= *formlet-expression*
`table` *exp* `with` *datatype* ⟨ `where` *constraints* ⟩ `from` *exp*

constraints ::= *record-label* `readonly` ⟨ `,` *constraints* ⟩

arg-list ::= *parenthesized-pattern* ⟨ *arg-list* ⟩

binding ::= `var` *pattern* `=` *exp* `;`
exp `;`
`fun VARIABLE` *arg-list block*

bindings ::= ⟨ *bindings* ⟩ *binding*

block ::= `{` *block-contents* `}`

block-contents ::= ⟨ *bindings* ⟩ ⟨ *exp* ⟩ ⟨ `;` ⟩

labeled-exps ::= *record-label* `=` *exp* ⟨ `,` *labeled-exps* ⟩

exp ::= *table-expression*
`database` *atomic-expression* ⟨ *atomic-expression* ⟩ ⟨ *atomic-expression* ⟩

5 Grammar for files and declarations

file	::=	⟨ *declarations* ⟩ ⟨ *exp* ⟩
declarations	::=	⟨ *declarations* ⟩ *declaration*
declaration	::=	*fixity* `UINTEGER` *infix-op* `;` `typename CONSTRUCTOR` ⟨ `(` *typevars* `)` ⟩ `=` *datatype* `;` ⟨ *signature* ⟩ *toplevel-binding*
typevars	::=	`VARIABLE` ⟨ `,` *typevars* ⟩
toplevel-binding	::=	`var` *pattern perhaps-location* `=` *exp* `;` `fun VARIABLE` *arg-list* ⟨ *location* ⟩ *block* ⟨ `;` ⟩ `fun` *pattern infix-op pattern* ⟨ *location* ⟩ *block* ⟨ `;` ⟩

fixity	::=	`infix`
		`infixl`
		`infixr`

signature	::=	`sig VARIABLE :` *datatype* `sig` *infix-op* `:` *datatype*

6 Terminals

The meaning of non-literal terminals, which occur in uppercase in the grammar, is as follows:

Identifier character an uppercase or lowercase letter, a digit, or underscore (`_`). Then a `CONSTRUCTOR` is a non-empty sequence of identifiers starting with an uppercase letter and a `VARIABLE` is a non-empty sequence of identifiers starting with a lowercase letter. A `UINTEGER` is either `0` or a non-empty sequence of digits starting with a non-zero digit. A `SYMBOL` is a non-empty sequence of non-identifier characters (e.g. `++` or `$@`). A `REGEXCHAR` is a character not otherwise employed in the grammar for regular expressions. A `QNAME` is a non-empty sequence of identifier characters and colons; the first character may not be a colon. A `UFLOAT` consists of a `UINTEGER`, a dot (`.`), a possibly-empty sequence of digits and an optional exponent part consisting of the letter `e`, an optional minus (`-`), and a `UINTEGER`. `QUOTE` represents the single quote character, `'`.

An escape character is a backslash (`\`) followed by three octal digits, or by an uppercase or lowercase `x` followed by two hexadecimal digits. A `STRING` consists of a possibly-empty sequence of characters (other than `"`), escape characters and the sequence `\"` between double quotes (`"`). A `CHAR` consists of a character other than an escape character or the character `'` between single quotes (`'`). `CDATA` is a non-empty sequence of characters other than `{`,`}`,`<` or `&`.