Syntax of the Perspectives Language

Joop Ringelberg 29-10-19 Version: 3

Introduction

This text gives the syntax of the Perspectives Language. It is an appendix to *Introduction to the Perspectives Language*.

Identifiers

```
identifier = qualifiedName | prefixedName | segmentedName
qualfiedName = 'model:' segmentedName
prefixedName = prefix ':' segmentedName
prefix = lowerCase+
segmentedName = segment ['$' segmentedName]*
segment = upperCaseCharacter | [alphaNum | '_']+
```

On character classes

Perspectives is programmed in the Purescript language. We use the definitions of various character classes as defined in the Unicode package. These definitions are quite extensive; we do not repeat them here but refer the interested reader to the documentation of that package.

`upperCaseCharacter`. From the documentation of `isUpper`: upper-case or title-case alphabetic Unicode characters (letters). Title case is used by a small number of letter ligatures like the single-character form of /Lj/.

`alphaNum`. Alphabetic or numeric Unicode characters.

`lowerCase`. Lower-case alphabetic Unicode characters (letters).

`stringLetter`. Any character that is not a double quote, a backslash or an ampersand. Escape character sequences are allowed, too, but they are not documented here. We refer to the source code of Text.Parsing.Parser.Token

Type declarations

A Perspectives model is a nested hierarchy of type declarations. The syntax we give here is quite permissive in the sense that many syntactically correct expressions are semantical nonsense. Indentation is meaningful in PL. The syntax below suggests the indentation, without being precise.

BNF-like meta grammar

The Perspectives Language grammar is given in a format that derives from Backus-Naur Format (BNF). These are the conventions used:

`<production>`	refers to a production rule.
`[...]`	whatever is between brackets, zero or one time (optional)
`<production>*`	zero or more times the production.
`<production>+`	one or more times the production.

Grouping constructs:

`{...}`	whatever is between brackets, exactly once.
`{...}*`	whatever is between brackets, zero or more times
`{...}+`	whatever is between brackets, one or more times
`{<p> | <q>}`	`<p>` or `<q>` (grouped for convenience or to disambiguate)
`(<p> <q>)`	`<p>` and `<q>` between parentheses, i.e. the parenthesis are part of the production.

NOTE: parenthesis have no special meaning in this grammar syntax. Hence '(' and ')' are perfectly ordinary parts of a production.

`<p> | <q>`	`<p>` or alternatively `<q>`
`'{'`	the literal character "{"
`state`	the literal keyword `state`. We don't write keywords between double quotes.

NOTE: double quotes have no special meaning in this grammar syntax. Hence " is a perfectly ordinary part of a production.

The syntax

```
context =
      <contextKind> <ident>
            use <lowercaseName> for <ident>
            indexed <ident>
            aspect <ident>+
            state <state>
            {<context> | <role>}+
role = <calculatedRole> | <enumeratedRole> | <externalRole>
calculatedRole =
      <roleKind> <ident> = <expression>
            <rolePart>*
```

```
enumeratedRole =
      <roleKind> <ident> [(roleAttribute [, roleAttribute])] [filledBy
<ident>]
            <rolePart>*
roleKind = user | thing | context | external
roleAttribute = mandatory | relational | unlinked
externalRole =
      external
            <rolePart>*
rolePart =
      <perspectiveOn> |
      <perspectiveOf> |
      <inState> |
      <onEntry> |
      <onExit> |
      <state> |
      <aspect> |
      <indexed> |
      <property> |
      <view>
state =
      state <ident> = <expression>
            <state>*
            <onEntry>
            <onExit>
            <perspectiveOn>
            <perspectiveOf>
property = <calculatedProperty> | <enumeratedProperty>
calculatedProperty =
      property <ident> = <expression>
enumeratedProperty =
      property <ident> [(propertyAttribute [, propertyAttribute])]
roleAttribute = mandatory | relational
view =
      view <ident> (<ident>+)
perspectiveOn =
      perspective on <expression>
            <roleVerbs>
            <perspectivePart>*
perspectiveOf =
      perspective of <ident>
```

```
            <perspectivePart>*
perspectivePart =
      defaults
      <propertyVerbs> |
      <roleVerbs> |
      <inState> |
      <onEntry> |
      <onExit> |
      <action> |
      <perspectiveOn> |
      <perspectiveOf>
aspect =
      aspect <ident>
inState =
      in state <ident> [of {subject | object | context} state]
            defaults
            <propertyVerbs>
            <roleVerbs>
            <perspectiveOn>
            <perspectiveOf>
            <action>
onEntry =
      on entry [of {subject | object | context} state <ident>]
            {<notification> | <automaticEffect>}*
onExit =
      on exit [ of {subject | object | context} state <ident>]
            {<notification> | <automaticEffect>}*
action =
      action <ident>
            <statement>*
            |
            letA
                  {<lowercaseName> <- <expression>}*
            in
                  <statement>*
automaticEffect =
      do
            <statement>*
            |
            letA
                  {<lowercaseName> <- <expression>}*
            in
```

```
                  <statement>*
notification =
      notify [<ident>] sentence
sentence = " { <charString> | '{' <expression> '}' }* "
charString = a string of any character that is not { or ".
roleVerbs =
      only ( <RoleVerb> {, <roleVerb>}+ )
      |
      excluding ( <RoleVerb> {, <roleVerb>}+ )
      |
      all roleverbs
propertyVerbs =
      view <ident> [ ( <propertyVerb>{, <propertyVerb>}+ )]
      |
      props [(<ident> {, <ident>}+) ] [ verbs ( <propertyVerb>{,
<propertyVerb>}+ )]
roleVerb = Remove | Delete | Create | CreateAndFill | Fill | Unbind |
RemoveFiller | Move
propertyVerb = RemovePropertyValue | DeleteProperty | AddPropertyValue |
SetPropertyValue | Consult
lowercaseName = an identifier consisting solely of lowercase characters.
ident =  {<name>$}*<name>
name = an identifier consisting of alphanumerical characters, upper and
lowercase, starting on an uppercase character.
```

Range types

The range types are defined operationally as follows:

`Number`: not yet decided.

Boolean: this type is inhabited by just the values true and false, designated by the same symbols.

String: a sequence of any character type, between double quotes

Date: all formats that can be parsed by Javascript Date.parse(), enclosed in single quotes. See Date Time String Format.

These same types govern the forms of the terminals in the expression syntax below: `number`, `boolean`, `string`, `date`.

Expressions

Expressions form a significant subpart of the Perspectives Language. They occur in several places in the type grammar:

- after the equals-sign in calculated roles and properties
- after the ‘if’ in the condition of an Action
- after the `assignmentOperator` in an assignment
- after the ‘if’ in a `botAction`
- after the ‘then’ in a `botAction`.

Again, the syntax allows for more expressions than that are semantically valid in PL. For example, we do not distinguish between expressions that yield a Boolean value and those that yield other values, even though there is a clear case for such Boolean expressions.

```
expression = simpleExpr | unaryExpr | binaryExpr | let*
simpleExpr = identifier | value | variable | >>= sequenceFunction | ‘this’
| ‘binding’ | ‘binder’ | ‘context’ | ‘extern’ |
value = number | Boolean | string | date
variable = lowerCase+
sequenceFunction = ‘sum’ | ‘product’ | ‘count’ | ‘minimum’ | ‘maximum’
unaryExpr = ‘not’ expr | ‘createRole’ identifier | ‘createContext’
identifier | ‘exists’ identifier
binaryExpr = ‘filter’ expression ‘with’ expression | expression operator
expression | ‘bind’ expression ‘in’ expression | ‘unbind’ expression from
expression | ‘unbind’ expression
operator = '>>' | '==' | '<' | '>' | '<=' | '>=' | 'and' | 'or' | '+' | '-'
| '*' | '/'
let* = ‘let*’ binding+ ‘in’ body
binding = variable ‘<-‘ expression
body = expr | assignment+
```

Operator precedence

Operator precedence is ruled by weights for each operator. Parentheses can be used to override these precedences.

Operator	Precedence
>>	8
==	0
/=	0
>	1
<	1
<=	1
>=	1
and	1
or	2
+	3
-	2
*	5
/	4

>>=	8
filter	9