

シナリオ記述言語 *Q*

言語仕様書

*Ver*3.0

2004.9.30

京都大学情報学研究科石田研究室

連絡先 : q@lab7.kuis.kyoto-u.ac.jp

シナリオ記述言語Q 言語仕様書

1. エージェント, アバター	3
2. 変数と値	5
3. シナリオと関数	6
4. キュー	9
5. アクション	10
6. 制御構造	11
7. 環境	13

1. エージェント, アバター

エージェント, アバターは以下の記法に従って定義される.¹

Syntax 1 – エージェント, アバターの定義 –

(defagent *name*

[:scenario *value*] ;; エージェントに割り当てたシナリオ

[:population *value*] ;; 同じ定義によって生成されるエージェントの個体数
{:*keyword value*}*)

(defavatar *name*

[:scenario *value*] ;; アバターに割り当てたシナリオ

[:population *value*] ;; 同じ定義によって生成されるアバターの個体数
{:*keyword value*}*)

{:*keyword value*}は, エージェントの生成に必要なデータを定義するために, エージェントシステムごとに任意に決定される. なお, Q ではこのようなキーワード引数が多用される. キーワード引数は `:` (コロン) で始まるキーワードシンボルに続いて値が指定される形式で利用される. 例えば, `:population 10` のように指定された場合, `:population` がキーワードシンボルで, そのキーワード値 `10` である事を意味する.

あらかじめ Q の言語仕様によって用意されているキーワード引数は:`:scenario`と`:population`の二つだけである. 前者はエージェント・アバターに割り当てるシナリオを指定するために用いられ, 後者は同じ定義によって生成されるエージェント・アバターの個体数を指定するために用いられる. ただし, これらのキーワード引数の利用はオプションである. `:scenario`が省略された場合は, 後で明示的にエージェントにシナリオを実行させる必要がある ([3 章シナリオと関数](#)を参照). 一方, `:population`が省略された場合は, 個体数 `1` を意味する.

例 1 : FreeWalk²におけるエージェントの定義

(defagent Taro

:scenario chat

:population 1

:location '(100.0 -10.0 5.0 45.0) ;; エージェントの初期座標

:shape '("body1.wrl", "body2.wrl")) ;; エージェントの描画用データ

¹以前の仕様書に存在した, 群集を定義するための`defcrowd`は, 本仕様書の`defagent`と等価である.

²京都大学石田研究室で開発されたマルチアバター・マルチエージェントプラットフォームである.

<http://www.lab7.kuis.kyoto-u.ac.jp/freewalk/>

```
(defagent evacuees
  :scenario fire-drill
  :population 10
  :location '((100.0 -10.0 5.0) (50.0 10.0 1.0)) ;;エージェントの配置される空間
  :shape '(("body1.wrl", "body2.wrl")) ;;エージェントの描画用データ
```

一つ目の例は、chat シナリオを実行するエージェント Taro を 1 体定義することを意味する。残りの :location, :shape は FreeWalk に特化したキーワード引数で、それぞれエージェントの初期座標、エージェントの描画用データを定義している。一方、二つ目の例では fire-drill シナリオを実行するエージェント evacuees を 10 体定義することを意味する。

defagent, defavatar で定義されたエージェント/アバターには ID が振られ name に割り当てられる。population が省略されるか、またはその値が 1 である場合は、エージェント ID がアトムで割り当てられ、値が 2 以上の場合はエージェント ID がリストで割り当てられる。なお、割り当てられたエージェント ID は name を評価することによって、その ID を得ることができる。さらに、list-ref を使うことによって、population の値が 2 以上で定義されたエージェント群の個々のエージェント ID を得ることができる。

例 2 : エージェントの ID リストの参照方法

```
(list-ref evacuees 3)
```

上記の例は、10 体のエージェント群を表す evacuees から 4 番目のエージェントの ID を参照している。

2. 変数と値

変数は予約語以外のアルファベットで始まるシンボルである。変数は宣言をしてから用いる。変数の宣言には次の2通りの方法がある。

- `define` による大域変数の宣言
- `let, let*`による局所変数の宣言

変数への値の代入は次の3通りの方法がある。

- `define, let, let*`による初期値の設定
- `set!`による代入
- キュー, アクションの実行による代入 ([4章キュー](#)を参照)

この中で、唯一キューやアクションの実行による代入が明示的でない。このようにキューやアクションの内部で代入される変数をパターン変数と呼ぶ。パターン変数の値は常に参照値であり、参照値を扱うために以下の操作が用意されている。

- 1 `refval ref`
参照 *ref* の参照先の値を取得する。
- 2 `set-ref! ref value`
参照 *ref* の参照先の値を *value* に更新する。
- 3 `make-ref value`
value を参照する参照値を作成する。

パターン変数の記法として\$で始まるシンボルの利用を特に勧めている。パターン変数も変数であるので、他の変数と同様に宣言が必要である。

3. シナリオと関数

インタラクションのシナリオは `defscenario` を用いて定義する。シナリオは他のシナリオあるいは関数から呼び出すことができる。シナリオは関数とは異なり状態遷移を記述するためのものである。

Syntax 2 – シナリオの定義 –

```
(defscenario name ({var}* {&key (var val)}* {&pattern (var val)}* {&aux (var val)}*)
  (state1 {(testform {form})*}*
    {(cue {form})*}*
    [(otherwise {form}*)])
  (state2 {(testform {form})*}*
    {(cue {form})*}*
    [(otherwise {form}*)])
  ...))
```

キーワード引数は `&key` を用いて定義する。パターン変数を受け取るキーワード引数は `&pattern` を用いて定義する。パターン変数宣言を用いて定義された引数は、キューやアクションの `out`, `inout` 属性の引数として利用する事が出来る。シナリオローカルな変数は `&aux` を用いて定義する。

各状態では、`testform`と`cue`を記述することができる。`testform`にはSchemeの式が書かれ、`cond`節（条件分岐）として処理される。一方、`cue`と`otherwise`は`guard`付きコマンド([6章制御構造](#)を参照)として処理される。したがって、`testform`が真であれば後続する `{form}*` を実行し、偽であれば観測された `cue` に後続する `{form}*` を実行する。`{form}*` の中では状態の遷移を記述するために `(go state)` を用いることができる。また、シナリオの実行主体のエージェントIDを `self` で参照することもできる。なお、`state1`, `state2`... は状態を表すシンボルであり、状態を的確に示す名前をつけることを薦める。明確な状態遷移の指定なく `{form}*` の実行が終われば、シナリオを終了する。シナリオの返却値は最後に実行された `form` の返却値となる。

例 3 : シナリオの定義

```
(defscenario chat (&key (message "Oh! Jiro, good boy!") &pattern ($x #f))
  (greeting ((?hear :name $x :word "Hello") (go identification))
    ((?hear :name $x :word "Bye") (go farewell)))
  (identification ((?equal (refval $x) Jiro) (!speak :sentence message :to Jiro))
    (otherwise (!speak :sentence "Hello" :to (refval $x))))
  (farewell (otherwise (!speak :sentence "Bye" :to (refval $x)))))
chat シナリオは greeting, identification, farewell の 3 つの状態から構成され、greeting 状態
```

から開始される. `greeting` 状態では誰かから `Hello` もしくは `Bye` と言われるのを待ち続ける. `Hello` と言われれば, `identification` 状態に遷移し, `Bye` と言われれば, `farewell` 状態に遷移する. `identification` 状態では, `Hello` と声を掛けてきたのが `Jiro` であれば, `Jiro` にキーワード変数 `message` に束縛されたメッセージを話し, `Jiro` でなければ, その相手に `Hello` と返事をする. 一方, `farewell` 状態では声を掛けてきた相手に `Bye` と返事をする.

定義したシナリオをあるエージェントに実行させるには, エージェントの定義時に実行させるシナリオを `:scenario` キーワード引数を用いて指定するか, 次のように実行させるかどうかである.

Syntax 3 – シナリオの実行 –

```
(scenario_name AgentID arguments)
```

`scenario_name` は呼び出すシナリオの名前であり, `AgentID` はシナリオの動作主体である. また, `arguments` はシナリオの実行に必要な引数である.

例 4 : シナリオの実行

```
(chat Taro :message "Hi Jiro")
```

上記の例は, エージェント `Taro` に `chat` シナリオを実行させることを意味する.

また同様な記法で, シナリオ内もしくは関数内でも他のシナリオを呼び出すことができる.

例 5 : シナリオ内におけるシナリオ呼び出し

```
(defscenario TestScenario-1 ()  
  (smalltalk (otherwise (chat self :message "Hi Jiro. "))))
```

例 6 : 関数内におけるシナリオ呼び出し

```
(defscenario TestScenario-2 ()  
  (smalltalk (otherwise (func-chat self "Hi Taro. "))))
```

```
(define (func-chat agent message)  
  (chat agent message))
```

ただし, 関数内でシナリオを呼び出す場合には注意が必要である. シナリオの呼び出しにはそのシナリオの実行主体を指定しなければならないが, シナリオと異なり関数にはその関数の実行主体を参照する方法が存在しない. そこで, 関数の実行時に引数として, 関

数の実行主体のエージェント ID を渡しておくとも便利である。例えば `chat` シナリオを実行する関数は、上記のように引数として実行主体のエージェント ID が渡されている。なお、上記の両方の例に現れる `self` は、各シナリオを実行しているエージェント ID を表す。

一方、関数内にはシナリオだけでなく、キューやアクションを記述することも可能である。ただし、キューやアクションは暗黙のうちに `self` を参照するので、関数内で `self` にエージェント ID を束縛する必要がある。そのため関数の引数に実行主体となるエージェントの ID を渡さなければならない。

例 7：関数内におけるキュー・アクションの利用

```
(defscenario TestScenario-3 ()
  (greeting (func-position-and-speak self 3)))

(define (func-position-and-speak agent dist)
  (let ((self agent)
        ($agent (make-ref #f)))
    (?position :name $agent :distance dist)
    (!speak :sentence "Hello" :to (ref-val $agent))))
```

上記の例では、アクションやキューが暗黙の内に `self` を参照するので、引数 `agent` で渡されたエージェント ID を `let` 文により `self` に束縛している。

4. キュー

インタラクションのきっかけはキューと呼ばれ、エージェントに観測の依頼を行うものである。キューは以下の `defcue` で定義される。`defcue` はキューのインターフェースを定義するものであり、実体はエージェントシステムで定義される。なお `cue` の記法として `cue` の前には?をつける。

Syntax 4 – キューの定義 –

```
(defcue name  
  {(:keyword in|out|inout)}*)
```

`in`, `out`, `inout` はキーワード引数の属性を指定する。`in`, `out` 属性はエージェントから見た表現である。`in` は、キーワード引数の値がシナリオ側からエージェント側に送られる向き(エージェントへ渡す), `out` はエージェント側からシナリオ側に送られる向き(エージェントから受け取る)を意味する。`inout` は `in`, `out` 両方向(エージェントに渡した後, 受け取る)を意味する。

キューは観測が成功するまでその事象を待ちつづける。`defcue` の使い方の例をあげる。

例 8 : キューの定義

```
(defcue ?position (:name in) (:distance in) (:angle in))  
(defcue ?observe (:name in) (:gesture in))  
(defcue ?hear (:name out) (:word in))
```

キューを依頼した時のエージェントの振舞いは、各エージェントシステムで定義される。上記の `?hear` は指定された言葉を聞き取った時に成立し、聞き取った言葉を発したエージェントのエージェント ID を返却するものとする。その場合、シナリオ内で

```
(?hear :name $agent :word "Hello! ")
```

と、キューが呼び出されると、`Hello!` という言葉を聞き取った時に成立し、その時、パターン変数 `$agent` には、その言葉を発したエージェントのエージェント ID が設定される。

5. アクション

外界への作用はアクションと呼ばれ、エージェントに実行の依頼を行うものである。アクションには実行の終了を待たない非同期型アクションと、実行の終了を待つ同期型アクションがある。アクションは以下の **defaction** で定義される。**defaction** はアクションのインターフェースを定義するものであり、実体はエージェントシステムで定義される。なお、アクションの記法として、同期型アクションには!をつけ、非同期型アクションには!!をつける。両方の特性を備えたアクションについては、同期型と非同期型の両方の定義が必要である。

Syntax 5 –アクションの定義–

```
(defaction name
  {(:keyword in|out|inout)*})
```

in, *out*, *inout* はキーワード引数の属性を指定する。*in*, *out* 属性はエージェントの立場で考えている。*in* は、キーワード引数の値がシナリオ側からエージェント側に送られる向き(エージェントに渡す), *out* はエージェント側からシナリオ側に送られる向き(エージェントから貰う)を意味する。*inout* は *in*, *out* 両方向(エージェントに渡した後、貰う)を意味する。また、引数の属性が *out* もしくは *inout* の場合は、参照型を指定しなくてはならない。

例 9 : アクションの定義

```
(defaction !walk (:forward in) (:backward in) (:left in) (:right in) (:movement in))
(defaction !!walk (:forward in) (:backward in) (:left in) (:right in) (:movement in))
(defaction !speak (:sentence in) (:to in))
(defaction !!approach (:to in) (:distance in) (:angle in))
```

defcue, defaction の例であげたキューとアクションを用いたシナリオの例を以下に示す。

例 10 : シナリオの記述

```
(defscenario go-to-station (&pattern ($x #f))
  (waiting ((?hear :name $x :word "Hello!")
    (!speak :sentence "Hi!" :to (refval $x))
    (go waiting))
    ((?observe :name Taro)
    (!!approach :to Taro :distance "near")
    (go moving)))
  (moving ((?position :name Kyoto_Station :distance "near"))))
```

6. 制御構造

制御構造は Scheme に従う。それに加えて、以下の `guard` 付きコマンドの制御構造も提供されている。

Syntax 6 – ガード付きコマンド –

```
(guard {(cue {form})*}*  
      [(otherwise {form}*)])
```

Guard 形式は、キューとフォームからなる節の並びである。すべてのキューは一括してエージェントに送られ、エージェントは全てのキューを並行して観測する。観測しているキューのうちいずれかのキューが真になると、そのキューに後続するフォームを順に評価する。キューがひとつでも真になった時点で、キューの並行観測を終了する。いずれのキューも真とならない場合、`otherwise` 節が指定されていれば、後続するフォームが評価される。ただし、`otherwise` 節の実行のタイミングは、各エージェントシステムに任される。Guard 形式の返却値は最後のフォームの評価結果となる。もし複数のキューが成功した場合、どれを選択するかはエージェントシステムに任される。

例 11 : `guard` 文の記述

```
(guard ((?hear :name $who :word "Hello")  
      (!speak :sentence "Hello" :to (refval $who)))  
      ((?hear :name $who :word "Fire")  
      (!!approach :to Kyoto_station :distance "near")  
      ((?position :name Bus_terminal :distance "far")  
      (!!approach :to Bus_terminal :distance "near")  
      (!speak :sentence "Here is a bus stop!" :to Taro))))  
  
(guard ((?hear :name $x :word "What's that?")  
      (guard  
        ((?position :name Kyoto_station :distance "near")  
         (!speak :sentence "That's Kyoto station" :to (refval $x)))  
        ((?position :name Kyoto_University :distance "near")  
         (!speak :sentence "That's Kyoto University" :to (refval $x))))  
      (otherwise  
        (!speak :sentence "I don't know" :to (refval $x))))))
```

一つ目の例は、Hello と聞こえるか、Fire と聞こえるか、もしくは Bus_terminal が遠くに

あるかという3つの観測を同時並行に行うことを意味している。一方二つ目の例では **guard** を入れ子にして用いており、**What's that?**と聞こえれば、**Kyoto_station** が近くにあるのか、**Kyoto_University** が近くにあるのかという2つの観測を同時並行に行うことを意味し、仮にどちらも観測されなかった場合は、**I don't know** と返事することを意味する。

7. 環境

エージェント以外の環境について定義したい場合は、以下のように `defenv` を用いて定義することができる。ただし、`defenv` の利用はオプションである。

Syntax 7 – 環境の定義 –

```
(defenv name
  {:keyword value}*)
```

`{:keyword value}` は、環境の設定に必要な情報を得るために、エージェントシステムごとに決定できる。

例 12 : FreeWalk における環境の定義

```
(defenv space
  :max_entry 100                ;; 存在可能なエージェント等の最大数(integer)
  :ip_address "comm.-vaio.Stanford.edu" ;; 接続先 IP アドレス(string)
  :port_number 12345            ;; 接続先ポート番号(integer)
  :vrml_url "http://www.lab7.kuis.kyoto-u.ac.jp/shijo_street/" ;; 参照先URL(string)
  :vrml_path "data¥shijo_street¥") ;; リソースフォルダへの相対パス(string)
```

上記の例では、`space` という環境を定義している。`space` という環境ではエージェントを最大 100 体まで存在させることができ、この環境へアクセスするために必要な IP アドレス、ポート番号、またこの環境で利用できる描画データの URL やローカルディレクトリへの相対パスが定義されている。

謝辞

株式会社数理システム, パリ第六大学情報学研究所, 上海交通大学計算機科学科, イメージ情報科学研究所, 野村総合研究所に様々な御協力を頂いたことを記して感謝します. また, Q の開発は科学技術振興機構(JST)戦略的創造研究推進事業(CREST): デジタルシティのユニバーサルデザインプロジェクトの一環として行われました.