STATECRUNCHER Parsing

Graham G. Thomason

Report Relating to the Thesis "The Design and Construction of a State Machine System that Handles Nondeterminism"

Department of Computing
School of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, UK

July 2004

STATECRUNCHER Parsing

STATECRUNCHER is a state transition language which is used to as a test oracle to tests of state behaviour. This report is concerned with the language syntax and how it is compiled.

This report describes the STATECRUNCHER parser. The parser uses the GP4 tool [StCrGP4]. This is a reference manual rather than a user manual.

Contents

1. Introduction and overview

STATECRUNCHER is a state machine system, which is used as a test oracle to state-based tests for software systems. The figure below shows a tool chain for automatic generation and execution of tests, with STATECRUNCHER in the chain, and with the **parser** emphasized.

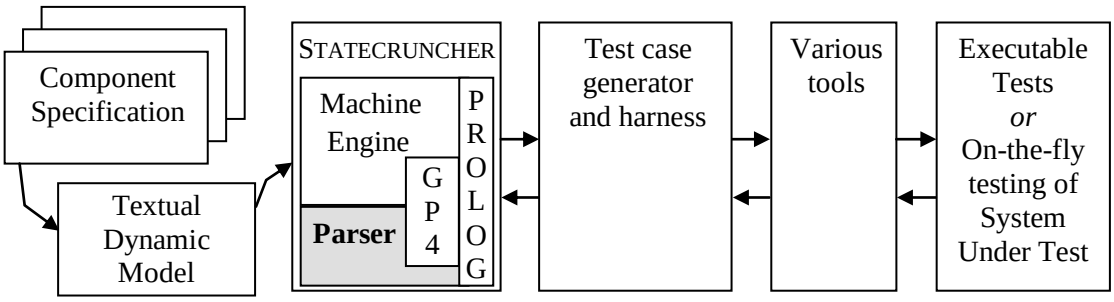


Figure 1. STATECRUNCHER in a testing tool chain

This report covers the detailed syntax of the language, and how it is parsed.

The STATECRUNCHER language is an extension to [ECHSM], which is close to the syntax of [CHSM]. ECHSM embeds in the state machine language many constructs that CHSM handles as calls to C++ code. STATECRUNCHER extends the language further syntactically with the following features:

- Declarations can be made at any hierarchical level; by default they have local scope.
- Scoping operators: Tagnames, enumerators, variables, events, PCOs and states can (in most cases) be defined and used employing scoping operations so as to refer to a scope other than the "current scope".
- Specification of multiple target states.
- Function calls have been added to expression syntax.

In order to concentrate on *parsing*, this report does not cover the detailed *semantics* of the constructs being parsed. STATECRUNCHER's handling of nondeterminism is essentially a semantic issue, and is not reflected in the syntax. The semantics are described in [StCrMain]. Neither does his report cover commands to a compiled and up-and-running STATECRUNCHER model. That is described in [StCrPrimer].

The parser for STATECRUNCHER uses Prolog and Definite Clause Grammars (DCG's). A good book covering this parsing technique is [Clocksin]. An underlying layer of parsing predicates is provided by the GP4 tool [StCrGP4]. GP4 provides in particular: tokenization, expression parsing, file I/O, and, for run-time use, evaluation of expressions.

The parser module as such does not check for context-sensitive errors such as type mismatches. For this purpose, a ***validator*** is run as a back end, also described in this report.

The output of the parser and validator is a set of Prolog-readable data structures consisting of nested lists. The implementation of Prolog used is WinProlog [WinPro] or SWI-Prolog [SwiPro].

The following figure shows the parsing stages:

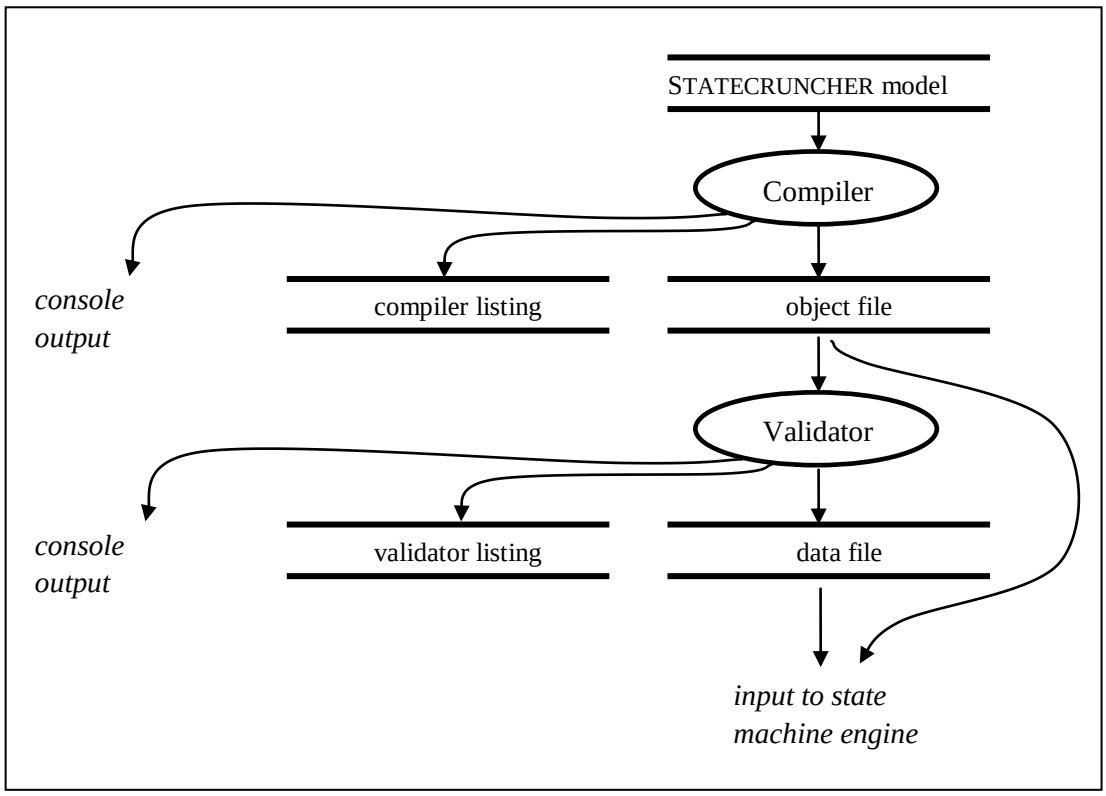


Figure 2. Data flow for the parser and validator

In the packaged STATECRUNCHER system, the user gives a single command to compile, and the validator is automatically called subsequently, unless there are compilation errors precluding it.

Abbreviations

CHSM	Concurrent Hierarchical finite State Machines
DCG	Definite Clause Grammar (Prolog grammar notation)
ECHSM	Extended Concurrent Hierarchical finite State Machines
GP4	Generic Prolog Parsing and Prototyping Package
GUI	Graphical User Interface
LHS	Left Hand Side
PCO	Point of Control and Observation
PRL	Philips Research Laboratories, Redhill, UK
RHS	Right Hand Side
TCL	Tool Command Language (scripting language)

2. Parsing conventions

The detailed syntax of STATECRUNCHER with parsing information is given in subsequent sections. The present section describes some conventions and recurring techniques used in parsing it.

2.1 Error status propagation

In order to inform the compiler control module about the success or failure in compiling each statement, status information is maintained as follows.

Every parse call at all levels of the grammar, (with a few exceptions mentioned below) is of the type:

```
parse_call(GSTATUS,[ITEM_NAME,LSTATUS,ITEM_PARSE])--> grammar_rhs
```

Exceptions to this structure:

- Some very low level parse calls which simply succeed or fail, returning the item(s) consumed only.
- The high-level "statement" parse call, which wraps in additional "machine path" information. This is described in a subsequent section.

`GSTATUS` is the global parsing status.

- It is set to `g_er` if the current parsing rule, or any deeper ones, contains an erroneous parse.
- It is set to `g_ok` if the current parsing rule, and all deeper rules, contain a valid parse.

The global parsing status is propagated to each higher level parse. Its purpose is to inform the compiler control module about whether the parse of a whole statement was successful or not.

`LSTATUS` is the local parsing status, applying to a fragment of syntax within a statement.

- It is set to `l_er` if the current parsing rule contains an erroneous parse.
- It is set to `l_ok` if the current parsing rule contains a valid parse. It should reflect whether the function of the current parsing level only could be performed. It should be set to `l_ok` even if there are errors in syntax fragments at a deeper level.

The local parsing status is not normally propagated upwards, except that a status of `l_er` implies that the global parsing status must be set to `g_er` at this and all higher levels in the parse tree. The purpose of the local parsing status is to provide information for error message generation.

Occasionally a local status may reflect the local parse status of a level just one level deeper. This may be appropriate when a list is being constructed from a head item parsed at one level, whilst the tail is parsed at a deeper level. If at the higher level the local status represents the validity of the list as a whole, and the local status information of individual elements is discarded in the higher level parse, then the per-element local statuses should be combined.

Note: the global parse status GSTATUS is ***not*** stored at each level in the parse tree. This is to help prevent the parse tree from becoming unnecessarily large. GSTATUS can, of course, be determined for any part of the parse tree by a recursive analysis.

Illustration of parsing statuses.
Suppose a particular kind of valid number consists of 6 digits $d_1d_2d_3d_4d_5d_6$ under the constraints that $d_1<d_2$ and $d_3<d_4$ and $d_5<d_6$. Figure 3 shows the grammar rules for such a number, including a semantic annotation to test the constraints. Suppose we attempt to parse the number 23654A. There are two reasons why this number is invalid:

- The constraint $d_3<d_4$ does not hold because 6 is not less than 5.
- The last digit is invalid.

A parse status propagation tree of such an attempted parse, with global and local status information, is shown in Figure 4.

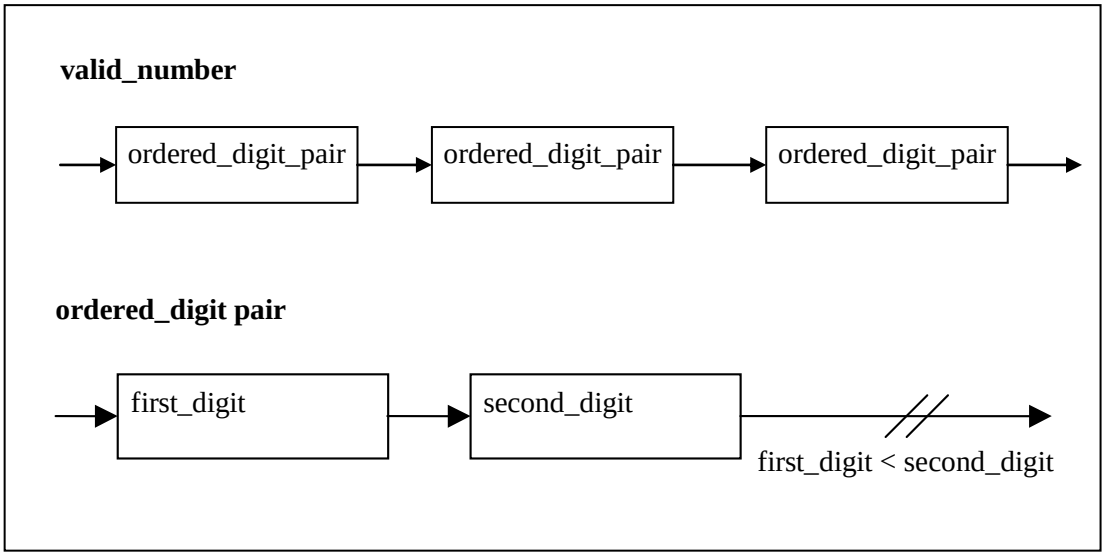


Figure 3. Parse status illustration: rules for a particular kind of *valid_number*

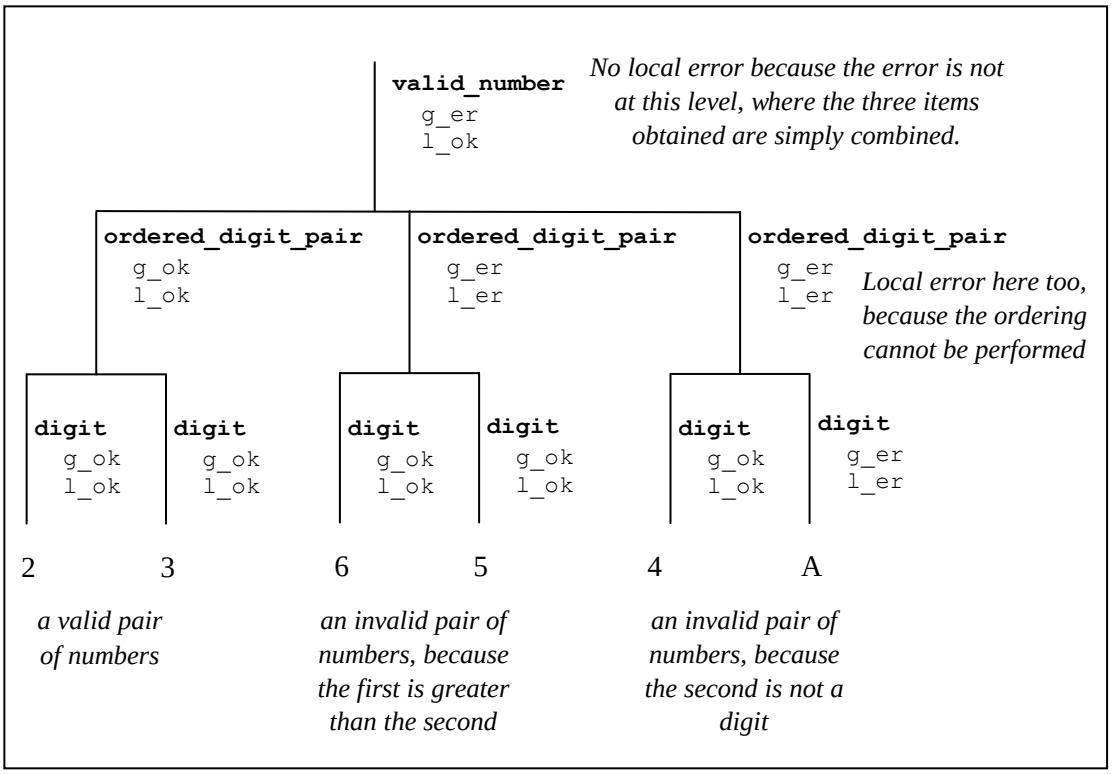


Figure 4. Error propagation in parsing the *valid_number* example

2.2 Generic aspects to parsing

2.2.1 Identifiers

Identifiers are GP4 style identifiers, but which are not keyword operators or keywords in STATECRUNCHER. The predicate `sy_identifier` provides a DCG grammar rule for a STATECRUNCHER identifier:

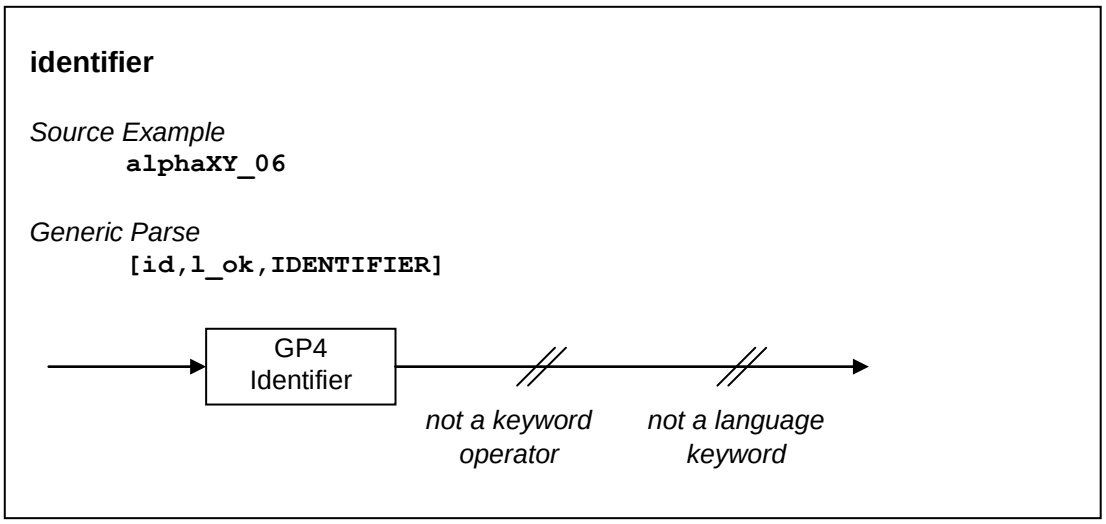


Figure 5. Identifier syntax

Items which are identifiers are indicated by an italic comment, e.g

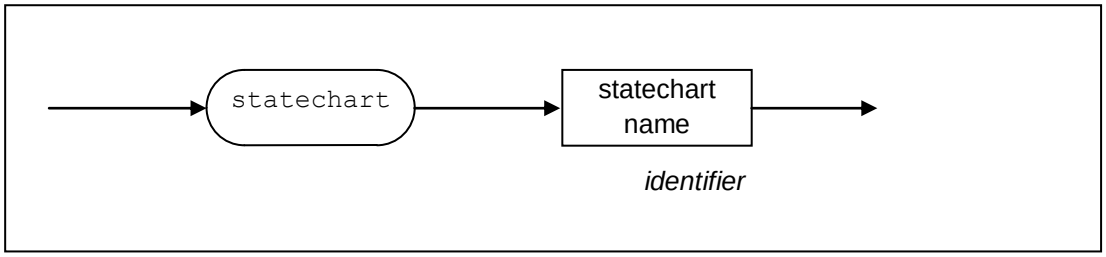


Figure 6. Syntax with an identifier

2.2.2 Repeating items

Two generic parsing techniques for repeating items will be frequently met with; they are typically handled as in the following figures:

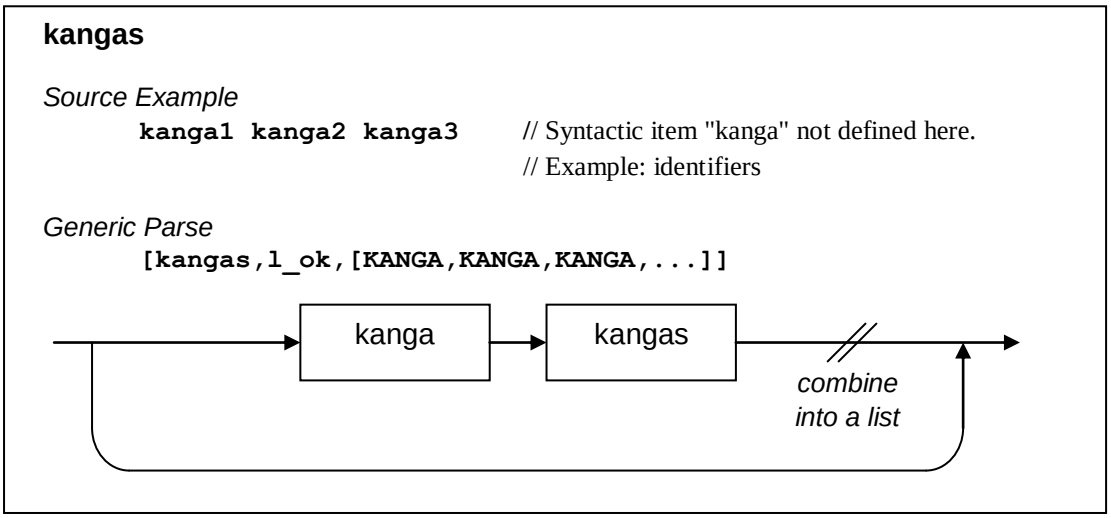


Figure 7. Zero or more of item *kanga*

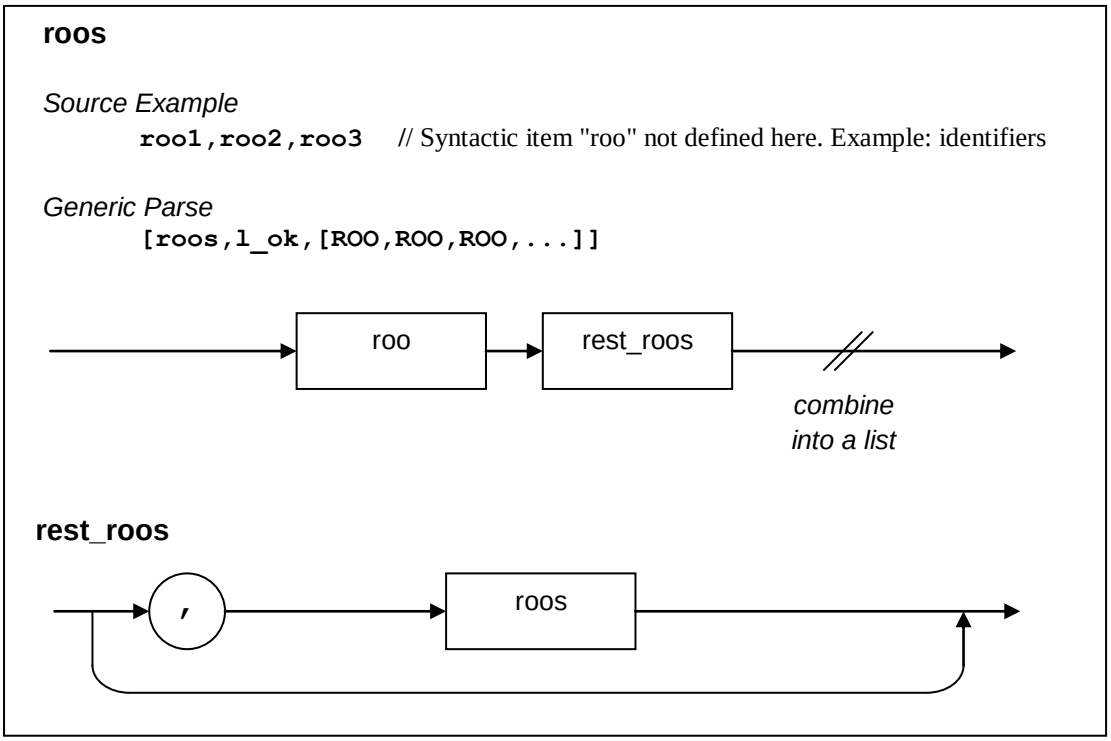


Figure 8. One or more of item *roo*, separated by commas

Resultant parse of lists
In the above examples, parsing information will typically be stripped and re-arranged so as to produce a list of items. A parse of *oofs* might be

```
[oofs,l_ok,[oof1,oof2,oof3]]
```

rather than a recursively nested structure.

2.2.3 Error messages

The following figure shows how an error message can be built into the parse tree. A minimal indication to the user is to present the parse tree containing the error message. A more sophisticated error message can be derived from the parse tree in a future release.

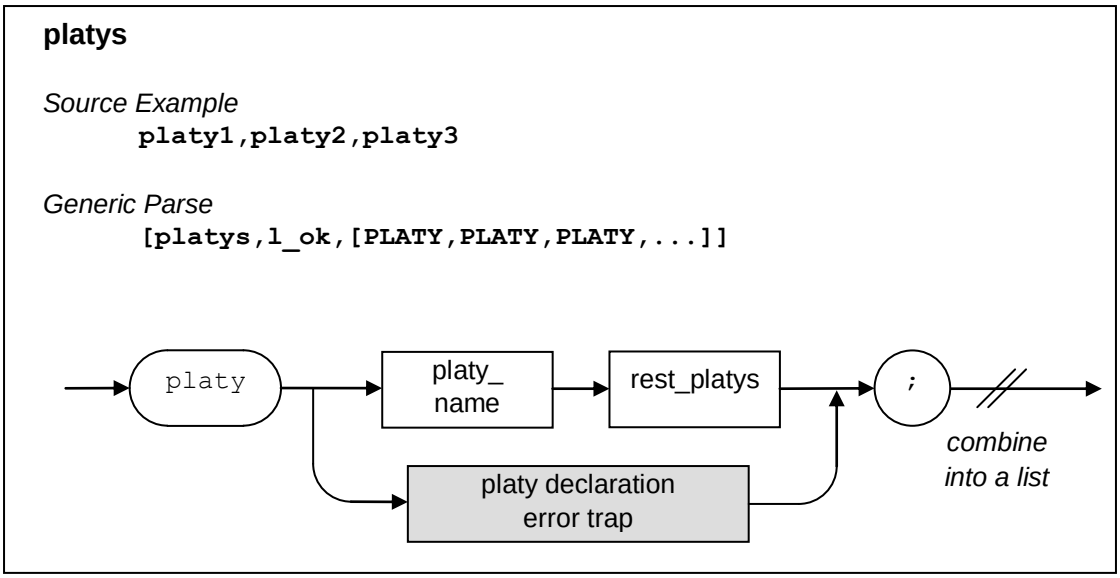


Figure 9. Declaration of *platy* items

In such an error trap, the syntax is *any text*, in a non-greedy mode.

Error message format
Error messages are inserted into the parse tree and are of the form:

```
**Error: parsing predicate: error-detail
```

The *parsing predicate* may be a slight variation on the actual predicate name in the source code (e.g. it may be less abbreviated; underscores may be removed).

2.2.4 Context sensitive absorption of "any text"

Error traps absorb any text, but sometimes they must ensure they absorb to the next *matching* bracket or brace. This is done by the technique illustrated in the following figure.

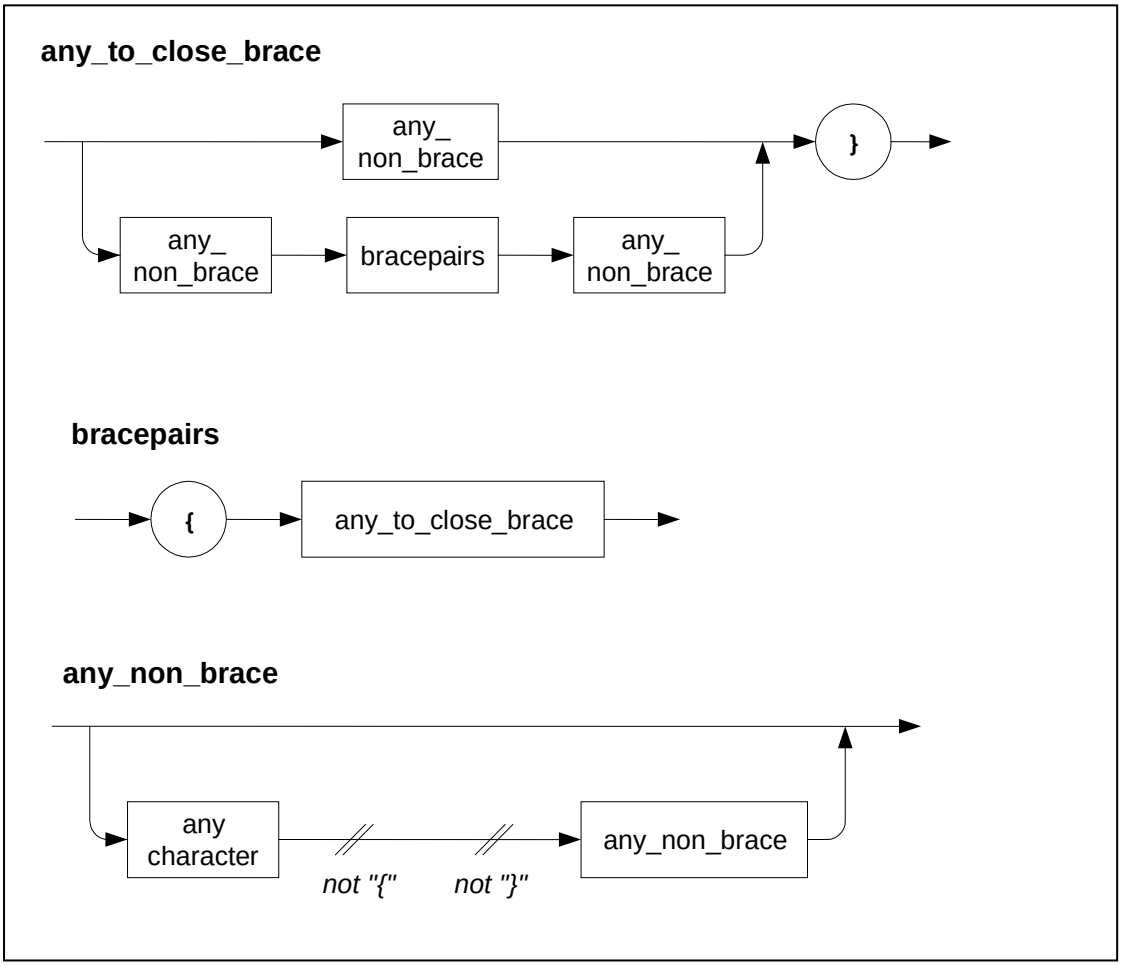


Figure 10. any_to_close_brace

The following productions are used to absorb any text up to a specific character (typically a semicolon), but *not* to the character within deeper braces.

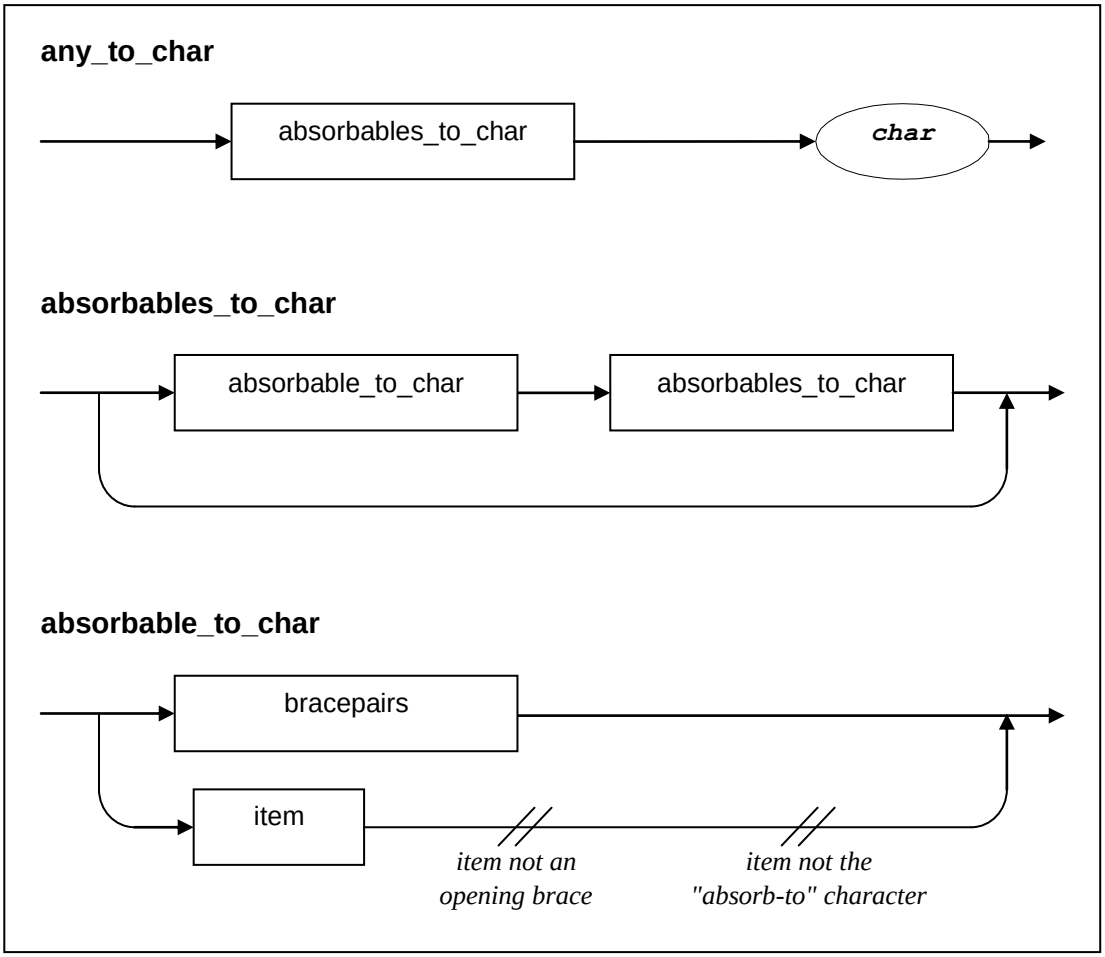


Figure 11. absorb_to_char

2.2.5 Cut-fail combinations in parsing

A cut-fail combination means that the current grammar rule is to fail, even if there are some untried productions (drawn lower down in the railroad diagrams) for this item.

The reasons for using a cut-fail combination are:

- as an optimisation, to eliminate an invalid parse quickly
- to reject a null-item-parse of an optional item if the next token is the token that introduces a non-null parse of that optional item.

If there is an error trap, this may be redundant. If there is *another* production that starts with the same next token, then it may be essential to *not* have the error trap at this level, but to provide it a level higher.

If there is another production that starts with the same next token, an alternative is to factor out the leading token in the representation of the grammar, so that it only occurs in one production rule. This is efficient, but it may mean defining new terms with respect to the terminology of an accepted specification document.

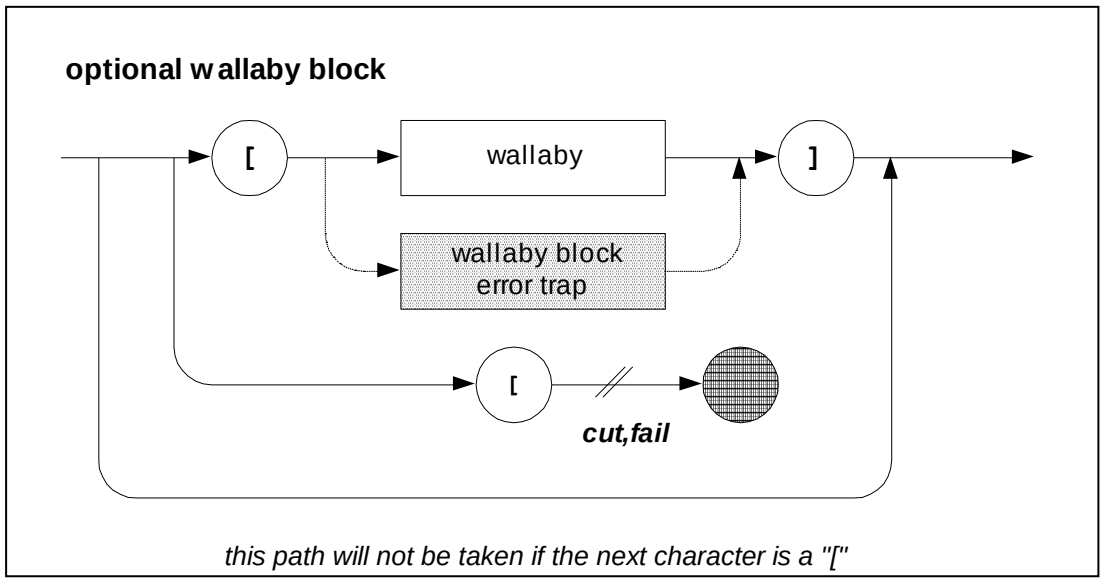


Figure 12. cut-fail combination

2.2.6 Transporting a parse up a level

At any given parsing level, there are two options as to how to handle a lower level parse result:

- nest it in a current level parse result
- transport the lower level parse body into the current level parse body.

For example, suppose a lower level parse is as follows:

```
[possum,l_ok,[POS,SUM]]
```

Now suppose we have an *optional possum* item:

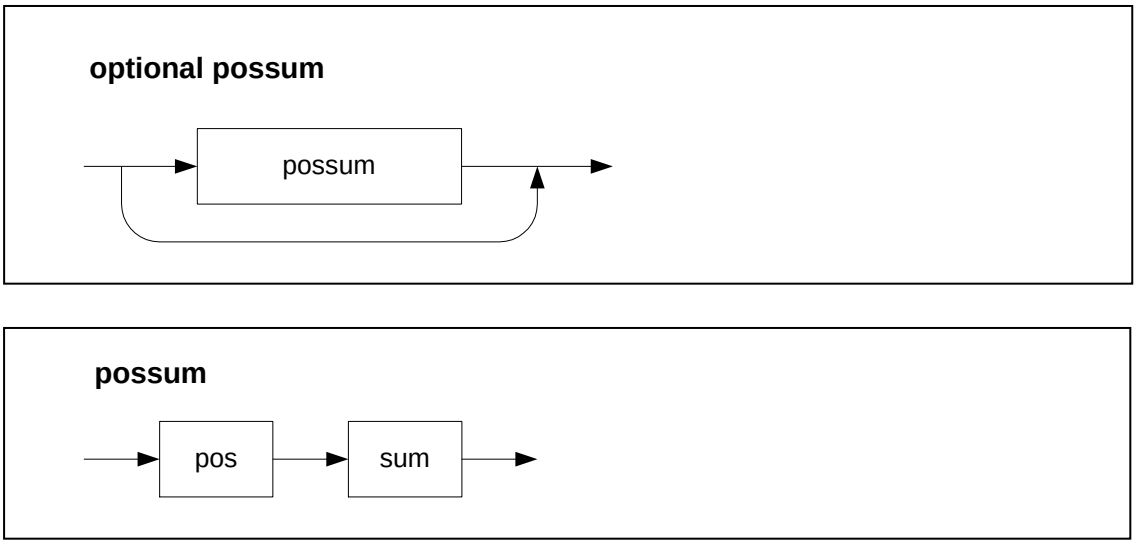


Figure 13. optional possum

The two representations could be

- nested:

```
[opt_possum,l_ok,[possum,l_ok,[POS,SUM]]]
[opt_possum,l_ok,[]]
```

- transported up

```
[opt_possum,l_ok,[POS,SUM]]
[opt_possum,l_ok,[]]
```

Transporting a parse up a level reduces the size of the parse tree, which is often convenient in reducing apparent complexity. The technique is appropriate where relatively trivial information is added by the extra parse level.

In cases where a parse is transported up a level, the local status (LSTATUS) indicator is transported up as well.

2.3 Interfacing with GP4

This section describes the inter-relationship between STATECRUNCHER-specific code and the underlying base parsing layer GP4 [StCrGP4].

2.3.1 Removal of white space

GP4 tokenization does not eliminate white space. White space is eliminated is as follows:

- Null parses terminate with optional white space
- All terminals are accessed by a generic predicate which eliminates leading and trailing white space.

Generic terminals that have been defined are as follows:

name	usage
`ex_opt_delim`	null parse
`sy_identifier`	identifiers (excludes keywords)
`sy_keyword/4`	language keywords, (excludes label keys)
`sy_labelkey`	label key
`sy_int`	an integer
`sy_literal`	any literal text as provided in the parameter

Table 1. Generic terminals

2.3.2 The interface with the compiler module (cp.pl)

The compiler module makes the following calls to the syntax predicates:

- an initial call: `sy_initialize`
- per tokenized statement:
  `sy_statement(WORLD,STATUS,OBJECT_STATEMENT,P1_STATEMENT,[])`
- a final call: `sy_finalize(MESSAGE)`

STATECRUNCHER uses the `sy_initialize` call to initalize machine path data and to output object and listing file headers.

The `sy_statement` call is provided with a tokenized statement (based on one line of input, with continuation lines if present) in parameter `P1_STATEMENT`. The `sy_statement` predicate must return

- in `STATUS`: a status of `g_er` or `g_ok`
- in `OBJECT_STATEMENT`: an atom or nested list representing the parse

Note that the call is compatible with DCG (Definite Clause Grammar) representation.

In STATECRUNCHER the `sy_statement` predicate provides a DCG style parse for

- statements with a structural bracket match error (handled without DCG's)
- null statements
- statechart statements
- type declaration statements
- variable declaration statements
- PCO declaration statements
- event declaration statements
- state (cluster, set and leaf-state) statements

Parsing continues from here in DCG format.

The `sy_finalize` call allows the syntax module to return an error message if an end-of-file has been reached unexpectedly. Otherwise, 'ok' is returned. STATECRUNCHER checks that there are no missing state definitions.

2.4 Conversion from list to predicate

A parsed statement of the type

```
[oc_statechart,LSTATUS, MACHINEPATH, DETAILEDPARSE]
```

is turned into a Prolog readable clause in the "object code" file, with a WORLD term added, of the type

```
oc_statechart(WORLD,LSTATUS,MACHINEPATH,DETAILEDPARSE).
```

e.g.

```
oc_statechart(1,l_ok,[mpath,l_ok,[sc]],
  [sc,[statnamsblk,l_ok,[s]]]).
```

Worlds are used to hold several statechart configurations when produced by nondeterminism (see [StCrMain]). They have been introduced not only for data constituents, but also for statechart elements, in preparation for a future enhancement, where the statechart may be dynamically altered per world.

3. STATECRUNCHER syntax

3.1 Statements

statement

Source examples:

```
statechart newtv(STATENAME)
enum tag1 {2,..,12};
tag1 v1,v2=6;
PCO pco1,pco2;
event event1,event2@pco1;
cluster q(e,f) deep history
```

Generic parse:

```
[statechart,LSTATUS, MACHINEPATH, DETAILEDPARSE]
[typedecl,  LSTATUS, MACHINEPATH, DETAILEDPARSE]
[vardecl,   LSTATUS, MACHINEPATH, DETAILEDPARSE]
[pcodecls,  LSTATUS, MACHINEPATH, DETAILEDPARSE]
[eventdecl, LSTATUS, MACHINEPATH, DETAILEDPARSE]
[state,     LSTATUS, MACHINEPATH, DETAILEDPARSE]
```

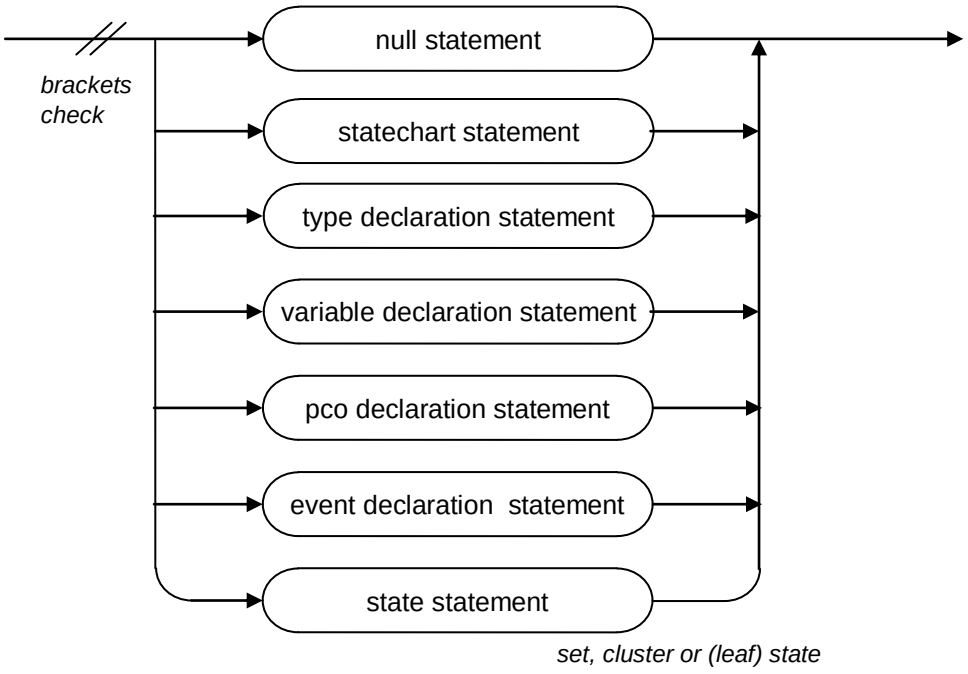


Figure 14. Statement

3.2 Machine path processing

The machine path is a list of the hierarchical state machines that lead to the machine in question.

Consider the following statechart structure (declarations/transitions excluded):

```
statechart s(a)
  set a(bb,cc,dd,ee,ff)
    cluster bb(bb1,bb2,bb3)
      state bb1
      state bb2
      state bb3
    cluster cc(cc1,cc2)
      cluster cc1(kkk)
        state kkk
      state cc2
    state dd
    cluster ee(ee1)
      cluster ee1(jjj)
        state jjj
    state ff
```

The machine path starts with statechart **s**, and is represented as **[s]**. At set **a** the path is represented in deep-first order as **[a,s]**. At cluster **bb** the path becomes **[bb,a,s]**, then in its states it becomes **[bb1,bb,a,s]**, **[bb2,bb,a,s]**, **[bb2,bb,a,s]**. Then the path shortens to **[cc,a,s]**. And so on.

In order to maintain the correct path during parsing, the following algorithm is used. In addition to the current machine path, an "expectlist" is maintained, consisting of states (sets, clusters and leaf-states) that are to be expected. States that are expected at a shallower level are put in nested lists.

The following table illustrates this.

	machine path	expectlist	un-wrap	con-sume	wrap rest	in-sert
`statechart s(a)`	[s]	[a]	no	no	no	YES
`set a(bb,cc,dd,ee,ff)`	[a,s]	[bb,cc,dd,ee,ff]	no	YES	CANT	YES
`cluster bb(bb1,bb2,bb3)`	[bb,a,s]	[bb1,bb2,bb3,[cc,dd,ee,ff]]	no	YES	YES	YES
`state bb1`	[bb1,bb,a,s]	[bb2,bb3,[cc,dd,ee,ff]]	no	YES	no	no
`state bb2`	[bb2,bb,a,s]	[bb3,[cc,dd,ee,ff]]	no	YES	no	no
`state bb3`	[bb3,bb,a,s]	[[cc,dd,ee,ff]]	no	YES	no	no
`cluster cc(cc1,cc2)`	[cc,a,s]	[cc1,cc2,[dd,ee,ff]]	YES1	YES	YES	YES
`cluster cc1(kkk)`	[cc1,cc,a,s]	[kkk,[cc2,[dd,ee,ff]]]	no	YES	YES	YES
`state kkk`	[kkk,cc1,cc,a,s]	[[cc2,[dd,ee,ff]]]	no	YES	no	no
`state cc2`	[cc2,cc,a,s]	[[dd,ee,ff]]	YES1	YES	no	no
`state dd`	[dd,a,s]	[ee,ff]	YES1	YES	no	no
`cluster ee(ee1)`	[ee,a,s]	[ee1,[ff]]	no	YES	YES	YES
`cluster ee1(jjj)`	[ee1,ee,a,s]	[jjj,[[ff]]]	no	YES	YES	YES
`state jjj`	[jjj,ee1,ee,a,s]	[[[ff]]]	no	YES	no	no
`state ff`	[ff,a,s]	[]	YES2	YES	no	no

Table 2. Machine path processing example

The latter columns indicate something of the actions taken on the expect list as each *state* statement is encountered. An unwrap operation may be required if the expected item is at a shallower level; unwrapping could be required repeatedly. The unwrapping count indicates how much of the machine path is to be removed. An item of the expectlist is consumed as the expected state is found, except for the initial statechart statement. Sets and clusters, because they introduce a deeper level, cause the existing expected items to be wrapped into a nested list before appending of the new items.

3.3 Syntax of declarations

The figures in this section show the syntax in a form that is close to the implementation. Typical source examples are used to illustrate the parse.
The Prolog clause names correspond, but have a prefix "sy_".

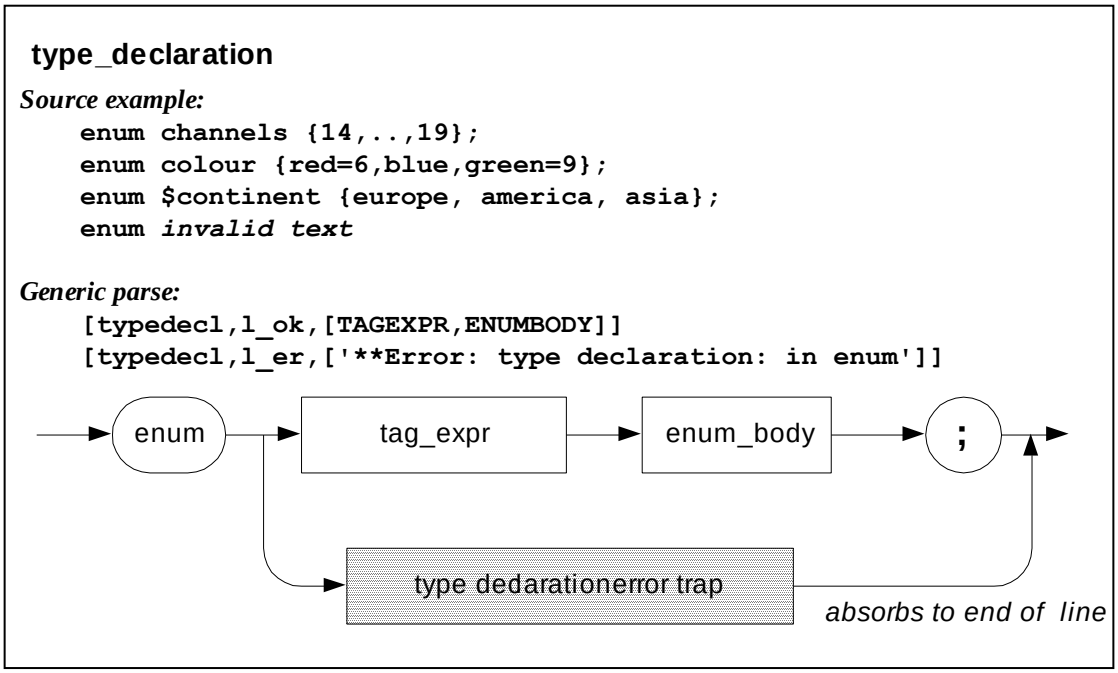


Figure 15. type_declaration

`enum channels {14,..,19};`	`[typedecl,l_ok,` `  [[ex_tag_expr,[ex_id,channels]],` `  [enumbody,l_ok,[range,l_ok,[14,19]]]]]`
`enum colour` `  {red=6,blue,green=9};`	`[typedecl,l_ok,` `  [[ex_tag_expr,[ex_id,colour]],` `  [enumbody,l_ok,` `  [valnams,l_ok,` `  [[red,6],[blue],[green,9]]]]]]`
`enum $continent` `  {europe, america,` `asia};`	`[typedecl,l_ok,` `  [[ex_tag_expr,[[ex_monadic,mback],` `  [ex_id,continent]]],` `  [enumbody,l_ok,` `  [valnams,l_ok,` `  [[europe],[america],[asia]]]]]]`
`enum #rubbish`	`[typedecl,l_er,` `  [**Error: type declaration: in enum]]`

Table 3. Examples of parsing “type_declaration”

enum_body

Source example:

```
{14 .. 19}
{red=6,blue,green=9}
{erroneous text}
```

Generic parse:

```
[enumbody,l_ok,RANGE]
[enumbody,l_ok,VALUENAMES]
[enumbody,l_er,['**Error: enum body: between braces']]
```

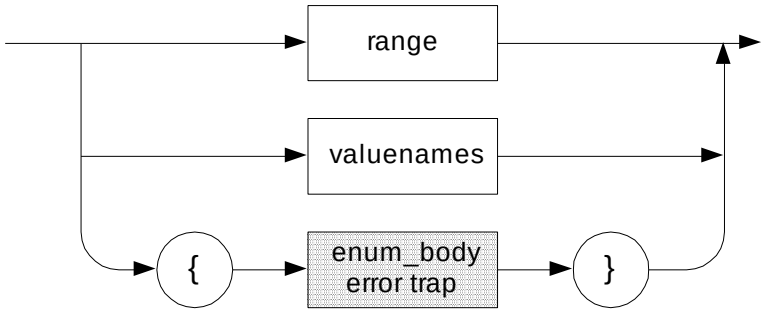


Figure 16. enum_body

{14,..,19};	[enumbody,l_ok,[range,l_ok,[14,19]]]
{red=6,blue,green=9}	[enumbody,l_ok, [valnams,l_ok,[[red,6],[blue],[green,9]]]]
{#rubbish}	[enumbody,l_er, [**Error: enum body: between braces]]

Table 4. Examples of parsing "enum_body"

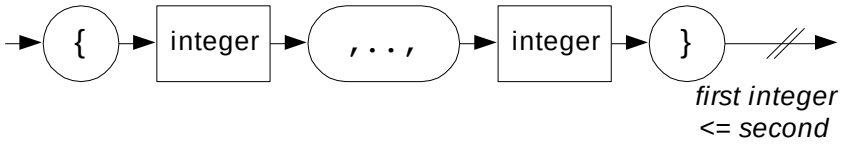
range

Source example:

```
{14,..,19}
{24,..,19}   /* error */
```

Generic parse:

```
[range,l_ok,[VAL1,VAL2]]
[range,l_er,[VAL1,VAL2,'**Error: range: lower .. higher']]
```



first integer
<= second

Figure 17. range

`{14,..,19};`	`[range,l_ok,[14,19]]`
`{24,..,19};`	`[range,l_er,` `  [24,19,**Error: range: lower .. higher]]`

Table 5. Examples of parsing "range"

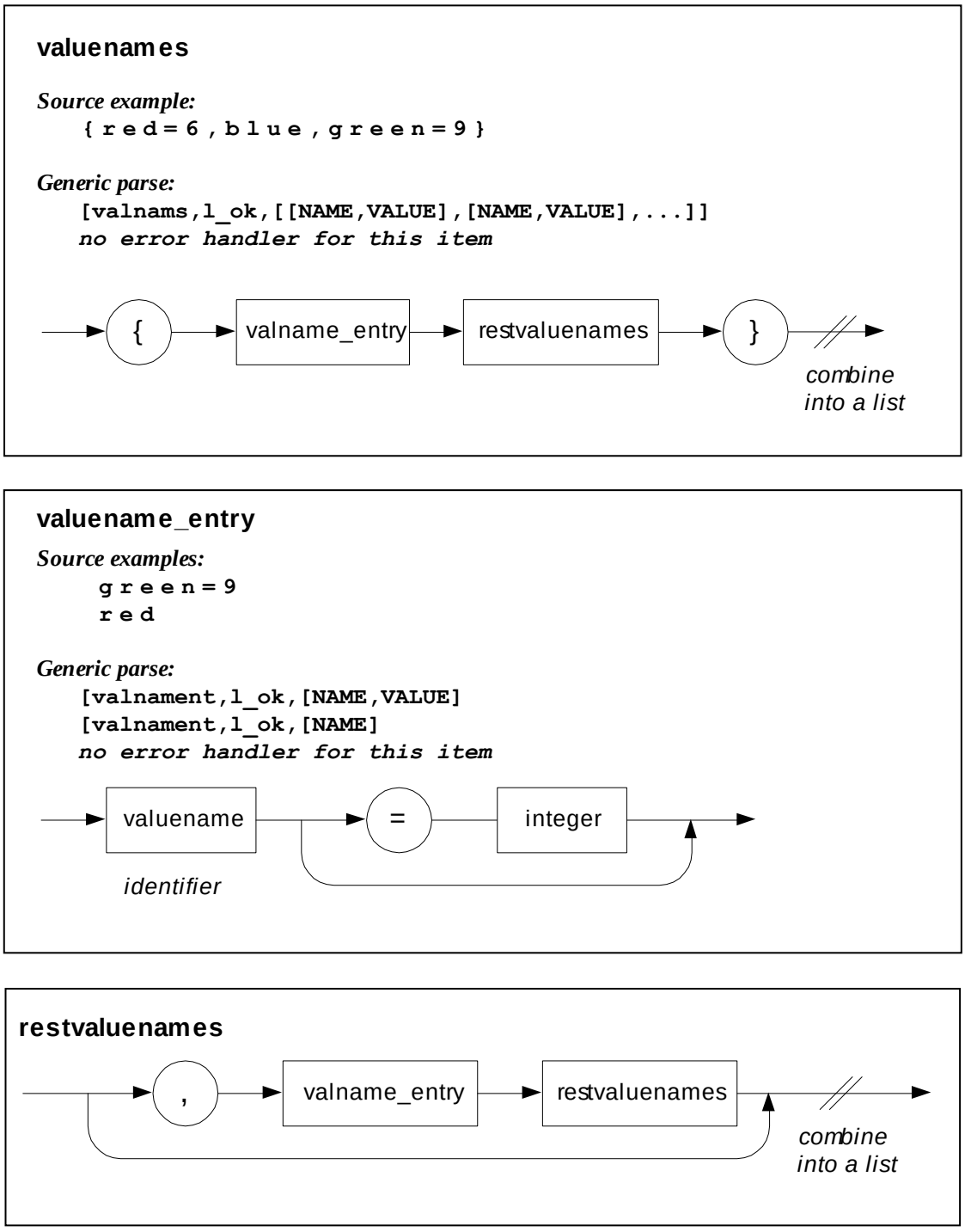


Figure 18. valuenames

{red=6,blue,green=9}	[valnams,l_ok,[[red,6],[blue],[green,9]]]

Table 6. Example of parsing "valuenames"

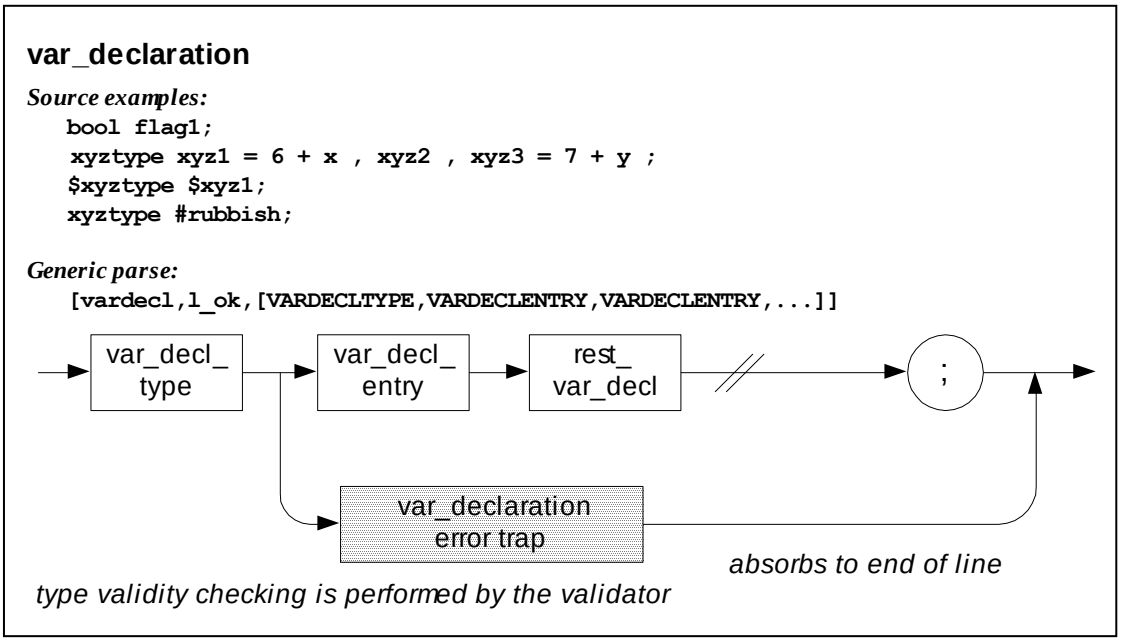


Figure 19. var_declaration

`bool flag1;`	`[vardecl,l_ok,` `  [` `  [vardecltype,l_ok,[bool]],` `  [vardeclent,l_ok,` `  [[ex_var_expr,[ex_id,flag1]]]]` `] ]`
`xyztype` `  xyz1 = 6 + x ,` `  xyz2 ,` `  xyz3 = 7 + y ;`	`[vardecl,l_ok,` `  [` `  [vardecltype,l_ok,` `  [enumtype,[ex_tag_expr,[ex_id,xyztype]]]],` `  [vardeclent,l_ok,` `  [[ex_var_expr,[ex_id,xyz1]],` `  [ex_expr,[[ex_dyadic,dplus],` `  [ex_co,int,6],[ex_id,x]]]]],` `  [vardeclent,l_ok,` `  [[ex_var_expr,[ex_id,xyz2]]]],` `  [vardeclent,l_ok,` `  [[ex_var_expr,[ex_id,xyz3]],` `  [ex_expr,[[ex_dyadic,dplus],` `  [ex_co,int,7],[ex_id,y]]]]]` `] ]`
`$xyztype $xyz1;`	`[vardecl,l_ok,` `  [` `  [vardecltype,l_ok,` `  [enumtype,[ex_tag_expr,` `  [[ex_monadic,mback],[ex_id,xyztype]]]]],` `  [vardeclent,l_ok,[[ex_var_expr,` `  [[ex_monadic,mback],[ex_id,xyz1]]]]]` `] ]`
`xyztype #rubbish;`	`[vardecl,l_er,[**Error: var declaration]]`

Table 7. Examples of parsing "var_declaration"

var_decl_type

Source examples:

```
bool
$colour
```

Parse example (1):

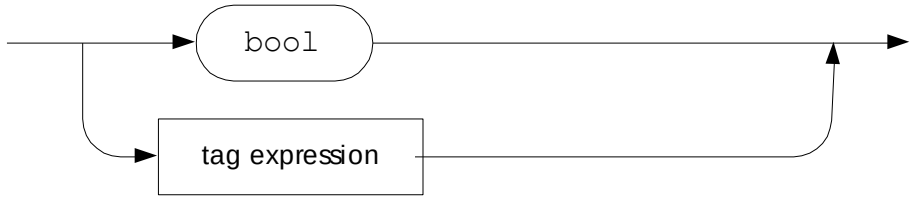
```
[vardecltype,l_ok,[bool]]
```

Parse example (2):

```
[vardecltype,l_ok,[enumtype,TAGEXPR]]
```

where

```
TAGEXPR=[ex_tag_expr,[[ex_monadic,mback],[ex_id,colour]]]
```



checking that the resultant tagname (enumtype) is known is performed by the validat

Figure 20. var_decl_type

`bool`	`[vardecltype,l_ok,[bool]]`
`$colour`	`[vardecltype,l_ok,` `  [enumtype,[ex_tag_expr,` `  [[ex_monadic,mback],[ex_id,colour]]]]]`

Table 8. Examples of parsing "var_decl_type"

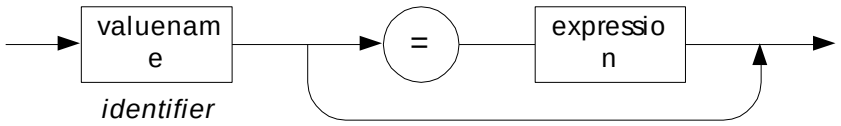
var_decl_entry

Source examples:

```
flag1
xyz1 = 6+x
$xyz1
```

Generic parse:

```
[vardeclent,l_ok,[ENTRY,EXPR]]
[vardeclent,l_ok,[ENTRY]]
```



rest_var_decl

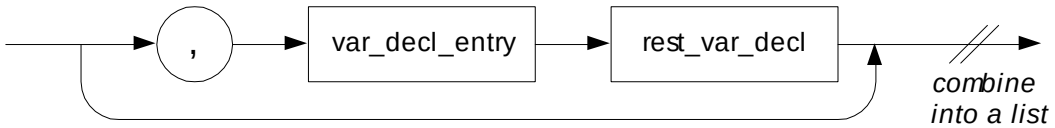


Figure 21. var_decl_entry

`flag1`	`[vardeclent,l_ok,` `  [[ex_var_expr,[ex_id,flag1]]]]`
`xyz1 = 6+x`	`[vardeclent,l_ok,` `  [[ex_var_expr,[ex_id,xyz1]],` `  [ex_expr,[[ex_dyadic,dplus],` `  [ex_co,int,6],[ex_id,x]]]]]`
`$xyz1`	`[vardeclent,l_ok,` `  [[ex_var_expr,` `  [[ex_monadic,mback],[ex_id,xyz1]]]]]`

Table 9. Examples of parsing "var_cecl_entry"

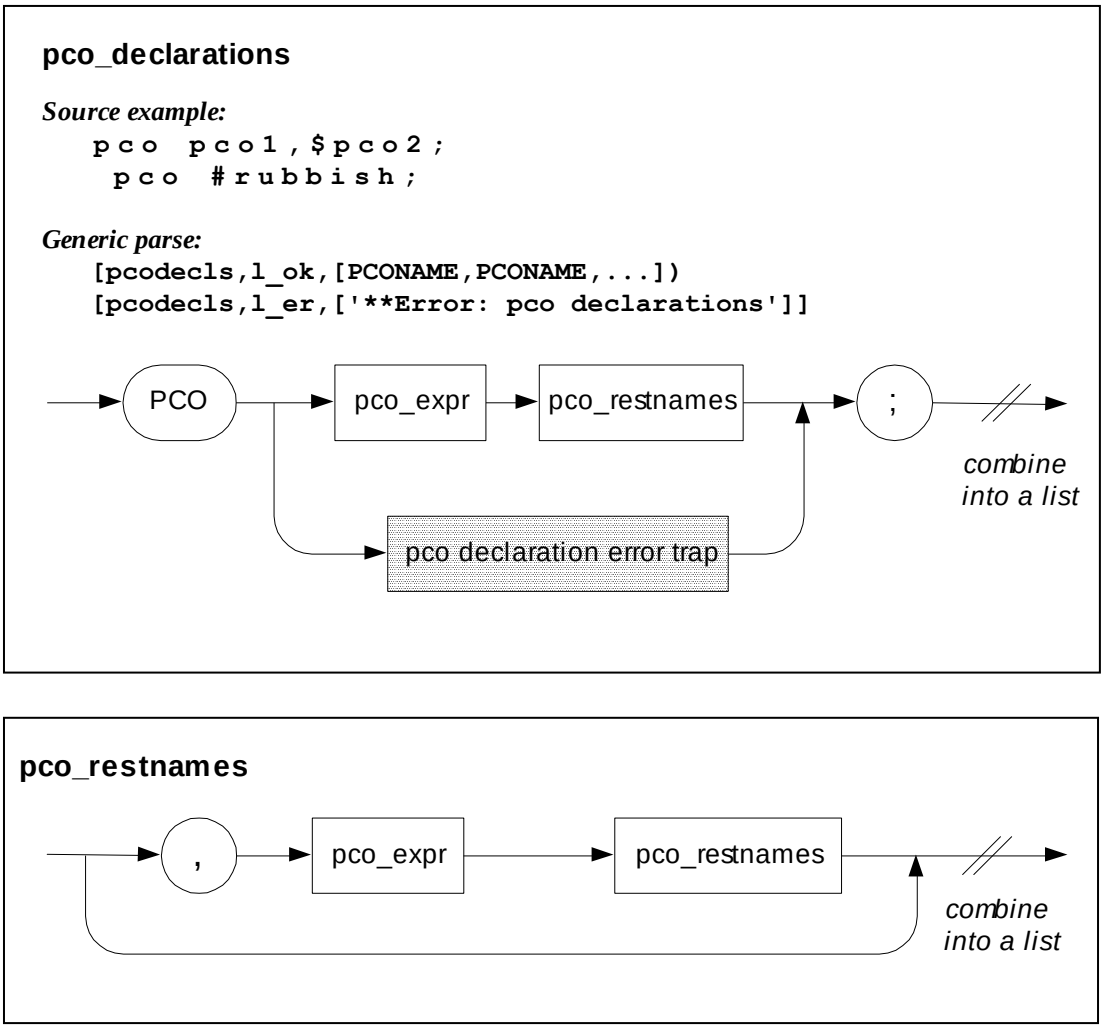


Figure 22. pco_declarations

`PCO pco1,$pco2;`	`[pcodecls,l_ok,` `  [` `  [ex_pco_expr,[ex_id,pco1]],` `  [ex_pco_expr,[[ex_monadic,mback],` `  [ex_id,pco2]]]` `] ]`
`PCO pco1,#rubbish;`	`[pcodecls,l_er,[**Error: pco declarations]]`

Table 10. Examples of parsing "pco_declarations"

event_decl

Source examples:

```
event alpha;
event alpha,beta;
event alpha,beta@pco1;
event alpha,$beta@$pco1;
event #rubbish ;
```

Generic Parse

```
[eventdecl,l_ok,[EVENTNAMES,PCO]]
```

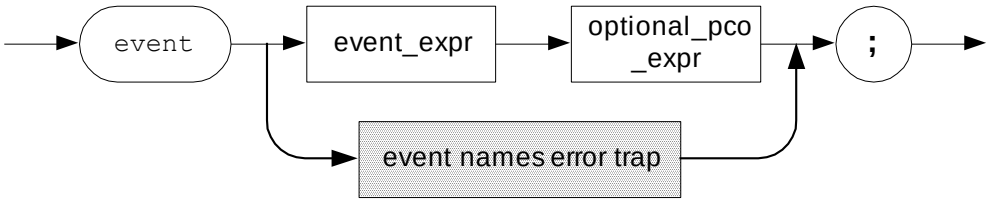


Figure 23. event_decl

`event alpha;`	`[eventdecl,l_ok,` `  [[[ex_evt_expr,[ex_id,alpha]]],[]]]`
`event alpha,beta;`	`[eventdecl,l_ok,` `  [` `  [ [ex_evt_expr,[ex_id,alpha]],` `  [ex_evt_expr,[ex_id,beta]]` `  ],` `  []` `] ]`
`event alpha,beta@pco1;`	`[eventdecl,l_ok,` `  [` `  [ [ex_evt_expr,[ex_id,alpha]],` `  [ex_evt_expr,[ex_id,beta]]` `  ],` `  [ex_pco_expr,[ex_id,pco1]]` `] ]`
`event alpha,$beta@$pco1;`	`[eventdecl,l_ok,` `  [` `  [ [ex_evt_expr,[ex_id,alpha]],` `  [ex_evt_expr,` `  [[ex_monadic,mback],[ex_id,beta]]]` `  ],` `  [ex_pco_expr,` `  [[ex_monadic,mback],[ex_id,pco1]]]` `] ]`
`event alpha,#rubbish;`	`[eventdecl,l_er,[**Error: event declaration]]`

Table 11. Examples of parsing "event_decl"

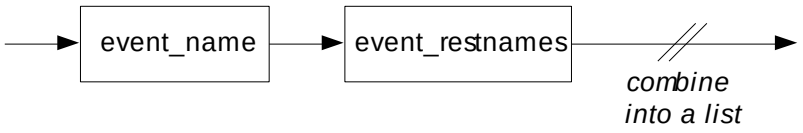
event_names

Source example:

```
ev1,ev2,$ev3
```

Generic parse:

```
[eventnames,l_ok,[EVENTEXPR,EVENTEXPR,...]]
```



event_restnames

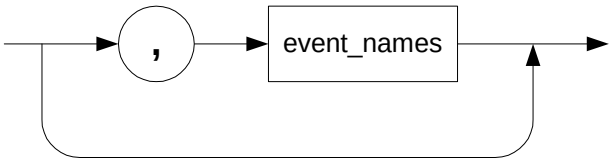


Figure 24. event_names (actually event scoping-expressions now)

opt_pco

Source examples:

```
@ pco1
@$pco2
null
```

Generic:

```
[opt_pco,l_ok,[]]
[opt_pco,l_ok,PCOEXPR]
```

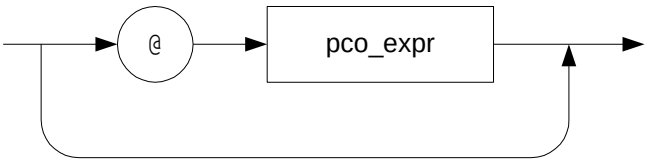


Figure 25. opt_pco

3.4 Syntax of state and transition blocks

First we give the syntax in a *non-feed-forward* way, i.e. expressing repetitions by backwards flow, as that is more compact and gives a better overview. Following this we give a description of the constituent items in the order of their description. Finally the whole syntax is described item by item in a purely feed-forward way.

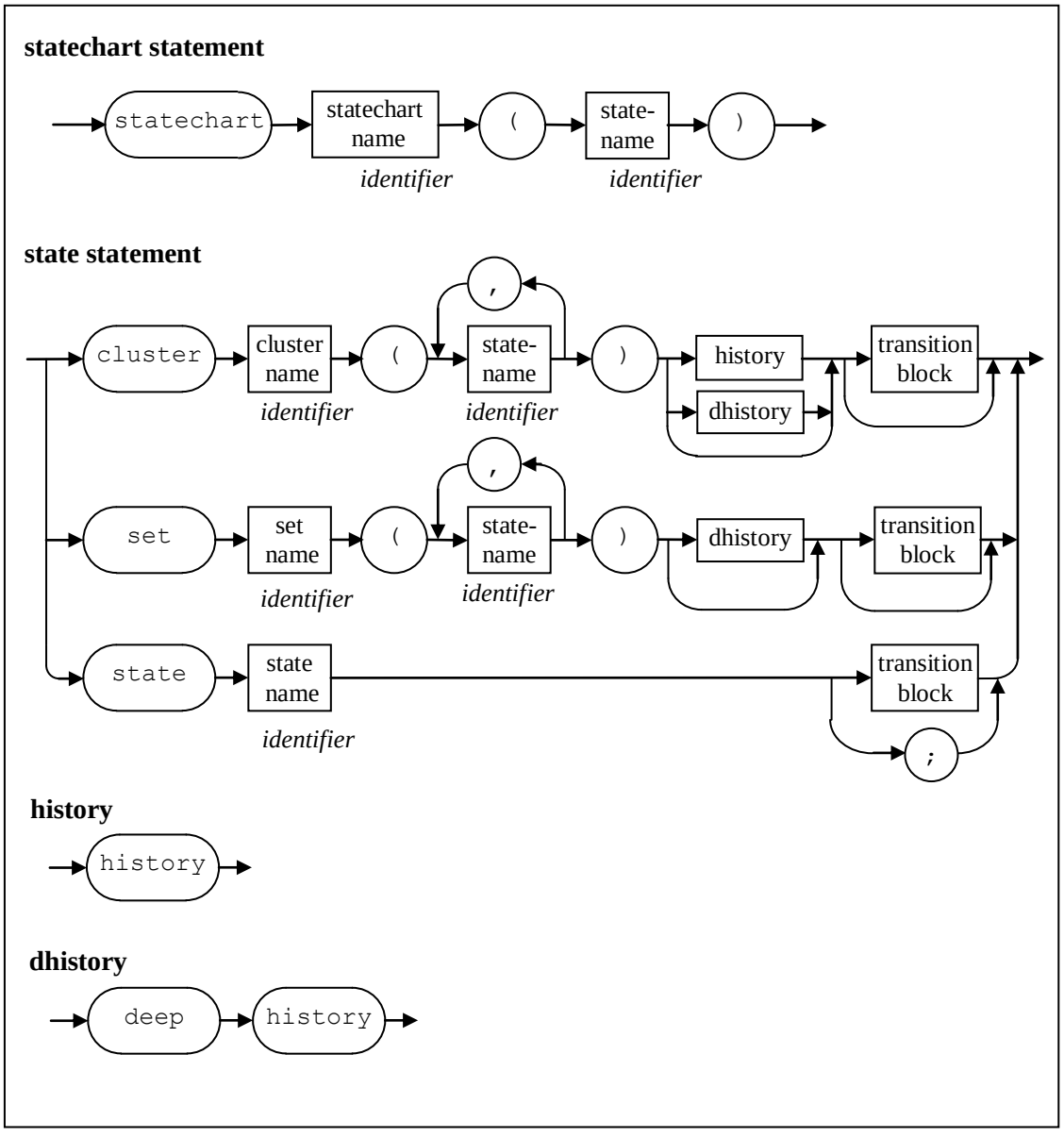


Figure 26. Overview (not feed forward) of statechart/cluster/set/state syntax

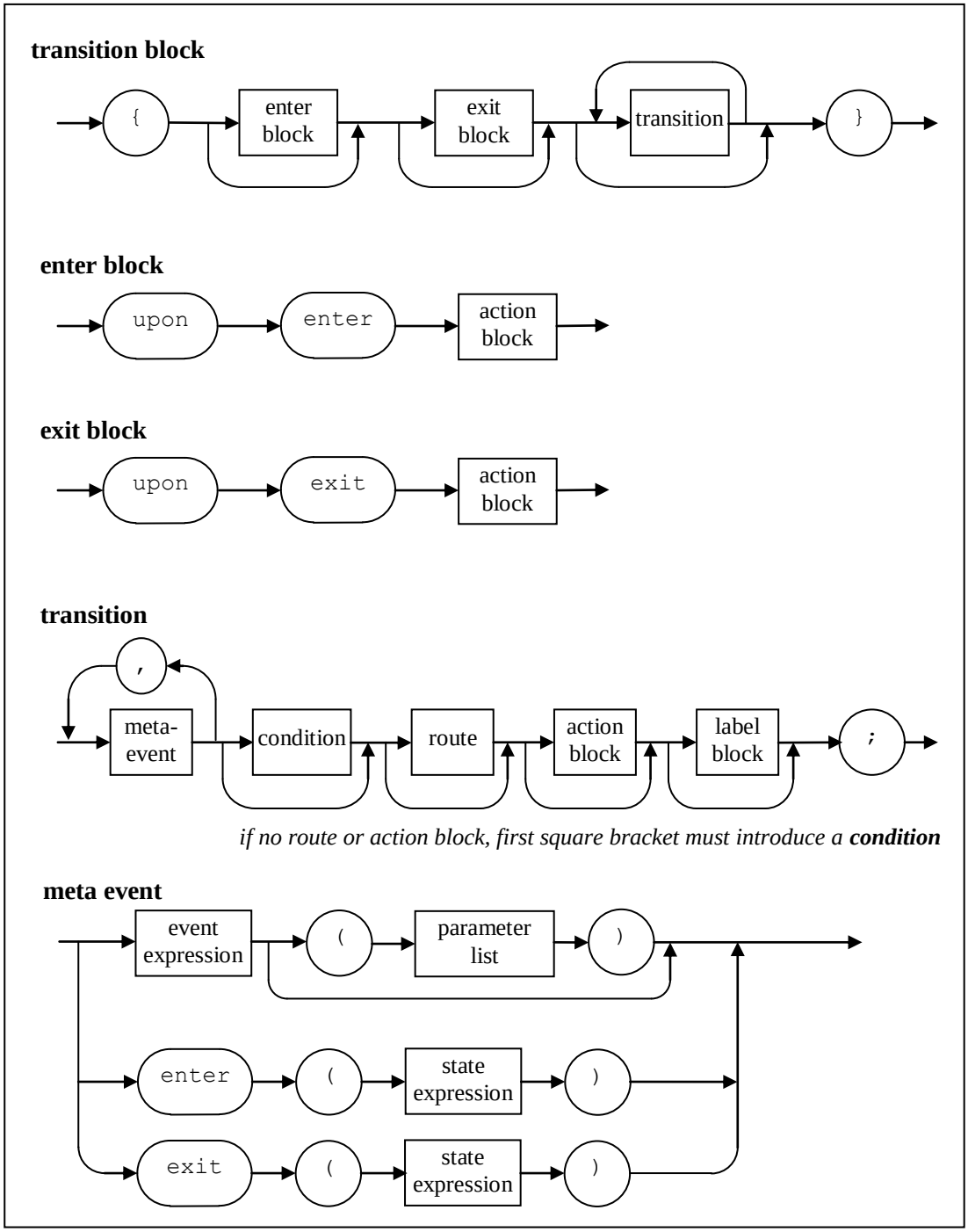


Figure 27. Overview (not feed forward) transition block syntax

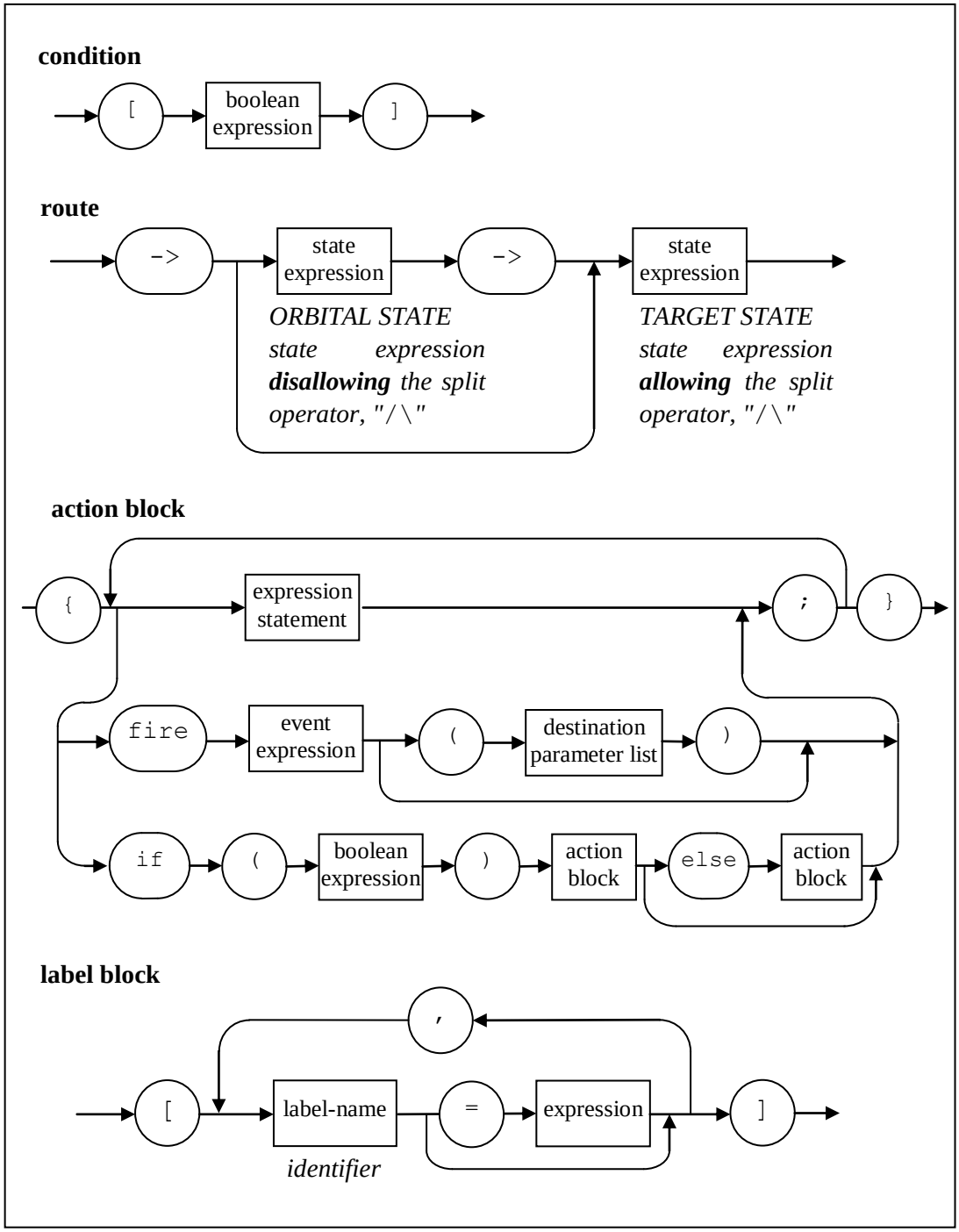


Figure 28. Overview (not feed forward) of transition block syntax (continued)

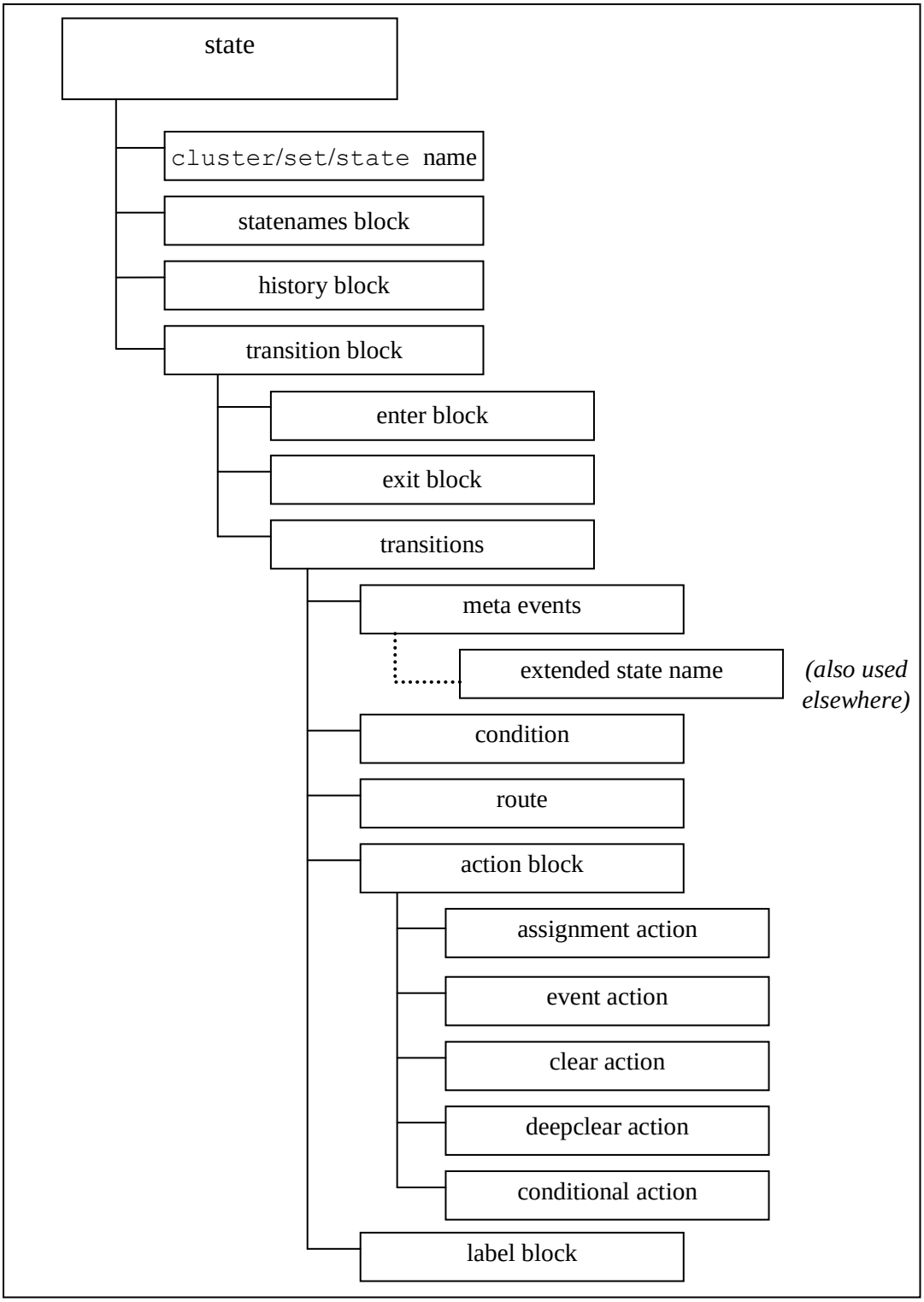


Figure 29. Overview of "state" in order of description

state

Source examples:

```
cluster p(x,d) deep history {alpha->d;beta->e;}
set     p(x,d)              {alpha->d;beta->e;}
state   x                   {alpha->d;beta->e;}
state   x ;
cluster p #rubbish
```

Generic parse:

```
[state,l_ok,[cluster,  CLUSTERNAME, STATENAMES, HISTORY, O1TRBLK]
[state,l_ok,[set,      SETNAME,     STATENAMES,          O1TRBLK]
[state,l_ok,[leafstate,STATENAME,                        O2TRBLK]]
[state,l_er,['**Error: state: cluster: NAME']]       // also applies to set
```

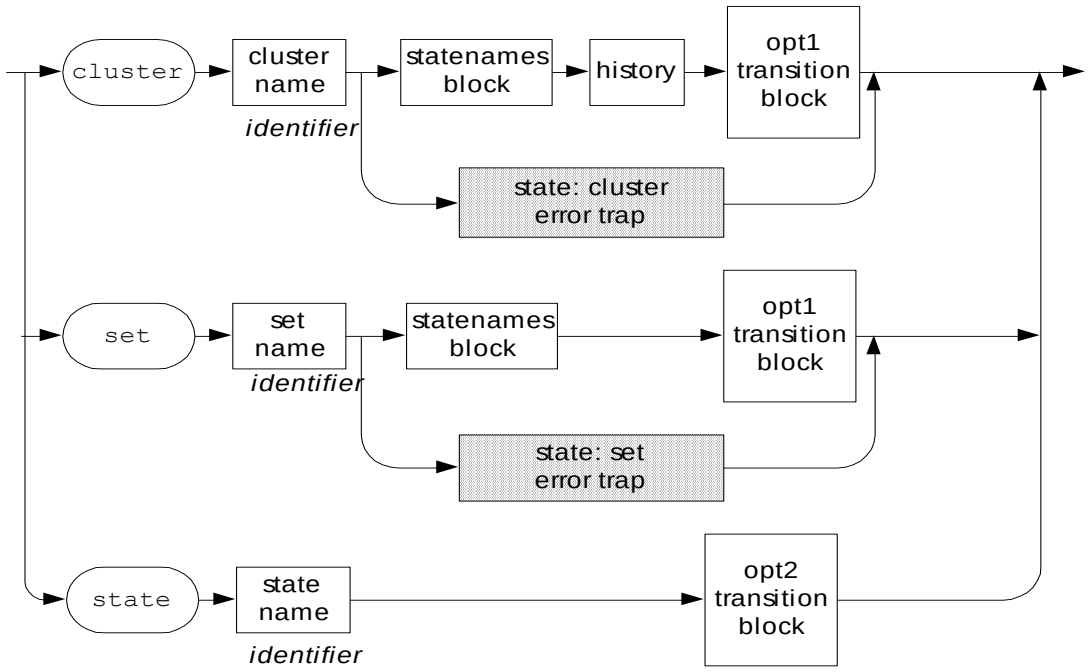


Figure 30. State

From test *[sc,sy,state,ssta,6]* cluster a (b,c) \ { alp->bet; }	[oc_state,l_ok, [cluster,a,CHILDREN,HISTORY,TRBLK]]], CHILDREN=[statnamsblk,l_ok,[b,c]], HISTORY=[history,l_ok,[]], TRBLK=[opt1trblk,l_ok,[EN,EX,TR]], TR=[transitions,l_ok,[TR1]
cluster p(x,d) deep history	[oc_state,l_ok,[cluster,p, [statnamsblk,l_ok,[x,d]], [history,l_ok,[deephistory]], [opt1trblk,l_ok,[]]]]
state x;	[oc_state,l_ok,[leafstate,x, [opt2trblk,l_ok,[]]]]
cluster p #rubbish	[oc_state,l_er,[**Error: state: cluster p]]

Table 12. Examples of parsing "state"

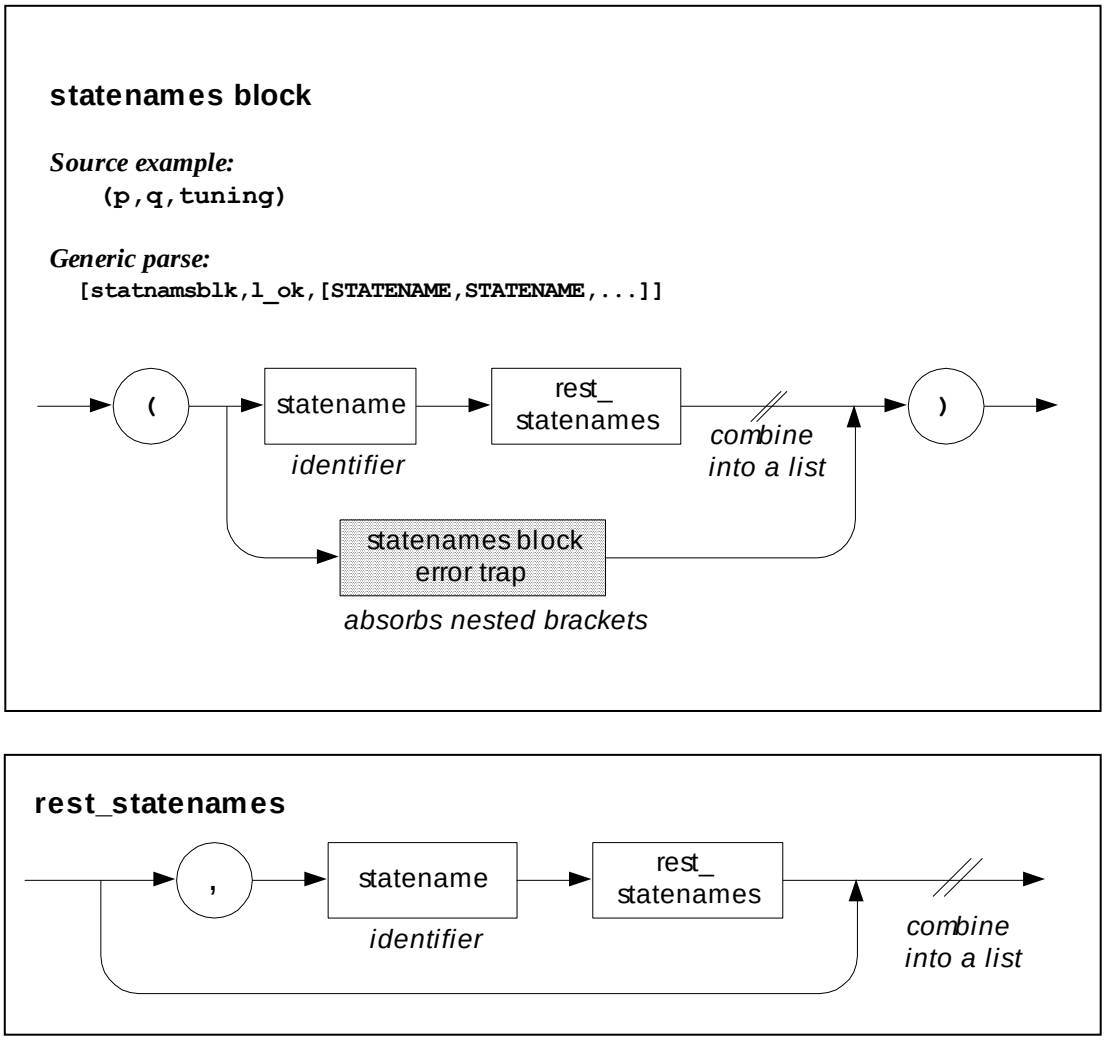


Figure 31. statenames block

(a,bb)	[statnamsblk,l_ok,[a,bb]]

Table 13. Example of parsing "statenames"

history

Source examples:

```
deep history
history
null
```

Generic parse:

```
[history,l_ok,[deephistory]]
[history,l_ok,[history]]
[history,l_ok,[]]
```

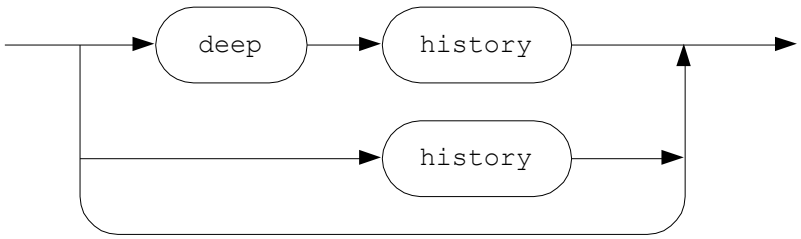


Figure 32. history

opt1_transition_block

Source example:

```
see "transition_block"
null
```

Generic parse:

```
[opt1trblk,l_ok,[ENTERBLOCK,EXITBLOCK,TRANSITIONS]]
[opt1trblk,l_ok,[]]
```

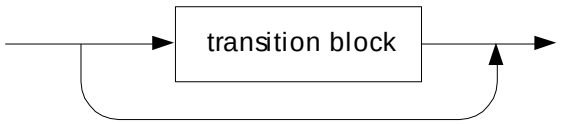


Figure 33. opt1_transition_block

Note: the above item

- transports the parse body of a transition block up one level

null	[opt1trblk,l_ok,[]]
From test *[sc,sy,state,o1tb,2]* {upon enter {x=y;} \ upon exit{x=y;} \ alp->e; \ alp[c==d]->e.f{x=y;}\ [lk_cost=6]; }	[opt1trblk,l_ok,[ENTERBLK,EXITBLK,TRANSITIONS]]], ENTERBLK=[enterblk,l_ok,ENTERACBLK], EXITBLK=[exitblk,l_ok,EXITACBLK], TRANSITIONS=[transitions,l_ok,[TR1,TR2].

Table 14. Examplesof parsing "opt1_transition_block"

opt2_transition_block

Source example:

```
see "transition_block"
null
```

Generic parse:

```
[opt2trblk,l_ok,[ENTERBLOCK,EXITBLOCK,TRANSITIONS]]]
[opt2trblk,l_ok,[]]
```

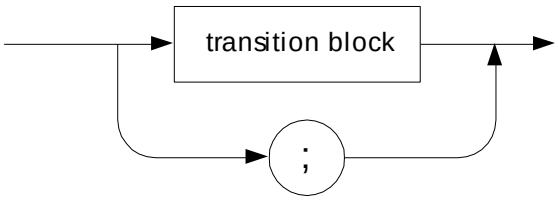


Figure 34. opt2_transition_block

Note: the above item

- transports the parse body of a transition block up one level

;	[opt2trblk,l_ok,[]]
From test *[sc,sy, state,o2tb,2]* {upon enter {x=y;} \ upon exit{x=y;} \ alp->e; \ alp[c==d]->e.f{x=y;} \ [lk_cost=6]; }	[opt2trblk,l_ok,[ENTERBLK,EXITBLK,TRANSITIONS]]], ENTERBLK=[enterblk,l_ok,ENTERACBLK], EXITBLK=[exitblk,l_ok,EXITACBLK], TRANSITIONS=[transitions,l_ok,[TR1,TR2].

Table 15. Examples of parsing "opt2_transition_block"

transition block

Source example:

```
{ upon enter ACTIONS,
  upon exit  ACTIONS
  TRANSITIONS }
```

Generic parse:

```
[trblk,l_ok,[ENTERBLOCK,EXITBLOCK,TRANSITIONS]
[trblk,l_ok,['Error: transition block']]
```

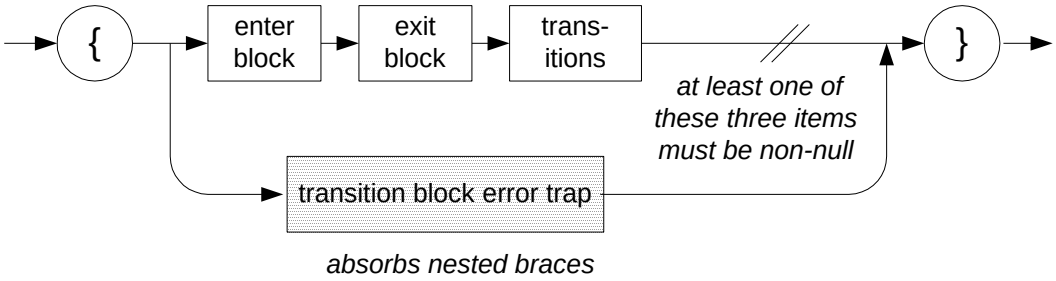


Figure 35. transition block

From test *[sc,sy,state,trb,5]* {upon enter {x=y;} \\ upon exit{x=y;} \\ alp->e; \\ alp[c==d]->e.f{x=y;} \\ [lk_cost=6]; }	[trblk,l_ok,[ENTERBLK,EXITBLK,TRANSITIONS]]], ENTERBLK=[enterblk,l_ok,ENTERACBLK], EXITBLK=[exitblk,l_ok,EXITACBLK], TRANSITIONS=[transitions,l_ok,[TR1,TR2].

Table 16. Examples of parsing "transition block"

enter block

Source examples:

upon enter *ACTIONS*

null

Generic parse:

[enterblk,l_ok,ACTIONS]

[enterblk,l_ok,[]]

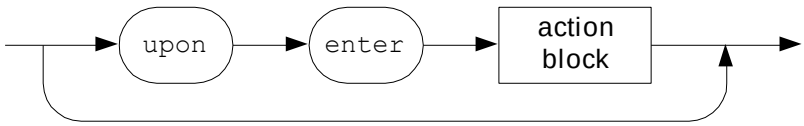


Figure 36. enter block

exit block

Source examples:

upon exit *ACTIONS*

null

Generic parse:

[exitblk,l_ok,ACTIONS]

[exitblk,l_ok,[]]

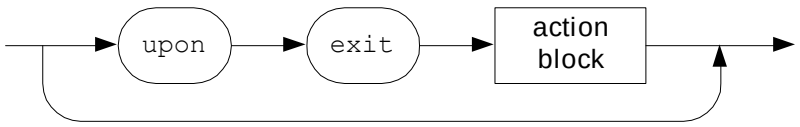


Figure 37. exit block

null	[enterblk,l_ok,[]]
From test *[sy,sc,enb,2]* upon enter {x=y;}	[enterblk,l_ok,ACTIONBLK]], ACTIONBLK=[actionblk,l_ok,[ACTION1].

Table 17. Examples of parsing "enter/exit block"

The parse of an exit block is analogous to that of an enter block.

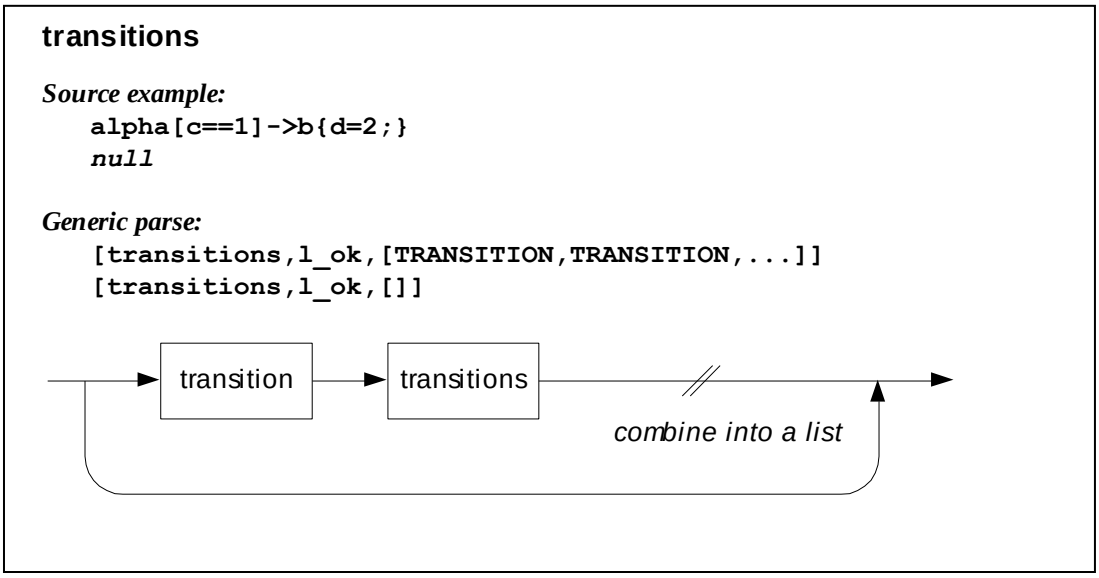


Figure 38. transitions (zero or more)

From test *[sc,sy, state,trns,2]* alpha -> e ; \ alpha,enter(a.b) \ [c==d] -> e.f \ {x=y;clear(b);} \ [lk_cost=6];	[transitions,l_ok,[TRN1,TRN2]]], TRN1=[transition,l_ok, [MEVS1,COND1,ROUTE1,ACBLK1,LABLK1]], MEVS1=[metaevents,l_ok,[EV1]], COND1=[condition,l_ok,[]], ROUTE1=[route,l_ok,[ORBIT1,DEST1]], ACBLK1=[optactblk,l_ok,[]], LABLK1=[labelblk,l_ok,[]], TRN2=[transition,l_ok, [MEVS2,COND2,ROUTE2,ACBLK2,LABLK2]], MEVS2=[metaevents,l_ok,[EV21,EV22]], COND2=[condition,l_ok,[ex_expr,_]], ROUTE2=[route,l_ok,[ORBIT2,DEST2]], ACBLK2=[optactblk,l_ok,[AC21,AC22]], LABLK2=[labelblk,l_ok,[LAB21].

Table 18. Example of parsing “transitions”

transition

Source example:

```
alpha[c==1]->b{d=2;};
#rubbish;
```

Generic parse:

```
[transition,l_ok,[EVENT,CONDITION,ROUTE,ACTIONS,LABELS]
[transition,l_er,['**Error: transition']  // various varieties
```

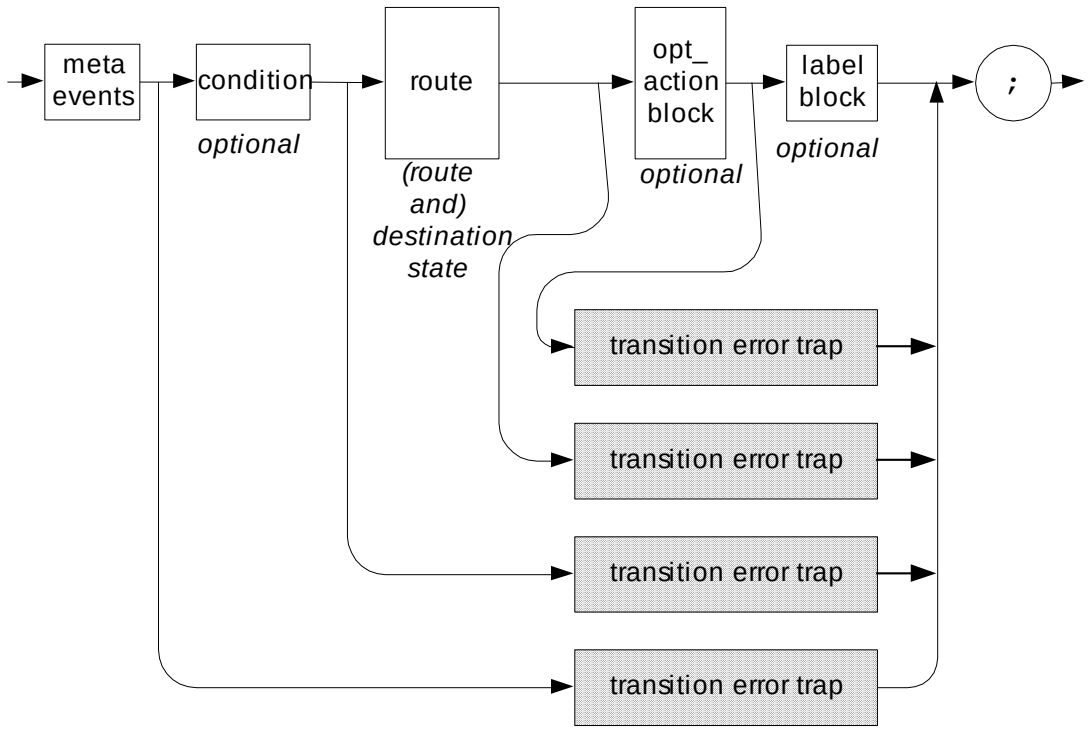


Figure 39. transition

From test *[sc,sy,state,tran,1]* alpha, \ enter(a.b) \ [c==d] ->e.f->g.h \ {x=y; clear (b); } \ [lk_cost=6, \ lk_time=-7];	[transition,l_ok,[MEVS,COND,ROUTE,ACBLK,LABLK]]], MEVS=[metaevents,l_ok,[EV1,EV2]], COND=[condition,l_ok,[ex_expr,EXPR]], ROUTE=[route,l_ok,[ORBIT,DEST]], ACBLK=[optactblk,l_ok,[AC1,AC2]], LABLK=[labelblk,l_ok,[LAB1,LAB2].
alpha,enter(a.b) [c==d]->e.f->g.h {x=y;clear(b);} [lk_cost=6,lk_time=-7] #trash;	[transition,l_er, [**Error: transition: invalid after label block]]
alpha,enter(a.b) [c==d]->e.f->g.h {x=y;clear(b);} [lk_cost=6, lk_time=-7/*]*/ ;	[transition,l_er, [**Error: transition: invalid after action block]]
alpha,enter(a.b) [c==d]->e.f->g.h #rubbish {x=y; clear(b);} [lk_cost=6,lk_time=-7];	[transition,l_er, [**Error: transition: invalid after extended state route]]
alpha,enter(a.b) [c==d] /*->*/ e.f->g.h {x=y;clear(b);} [lk_cost=6,lk_time=-7];	[transition,l_er, [**Error: transition: invalid after condition]]
alpha,enter(a.b) [c==d/*]*/ ->e.f->g.h {x=y;clear(b);} [lk_cost=6,lk_time=-7];	[transition,l_er, [**Error: transition: invalid after meta events]]

Table 19. Examples of parsing “transition”

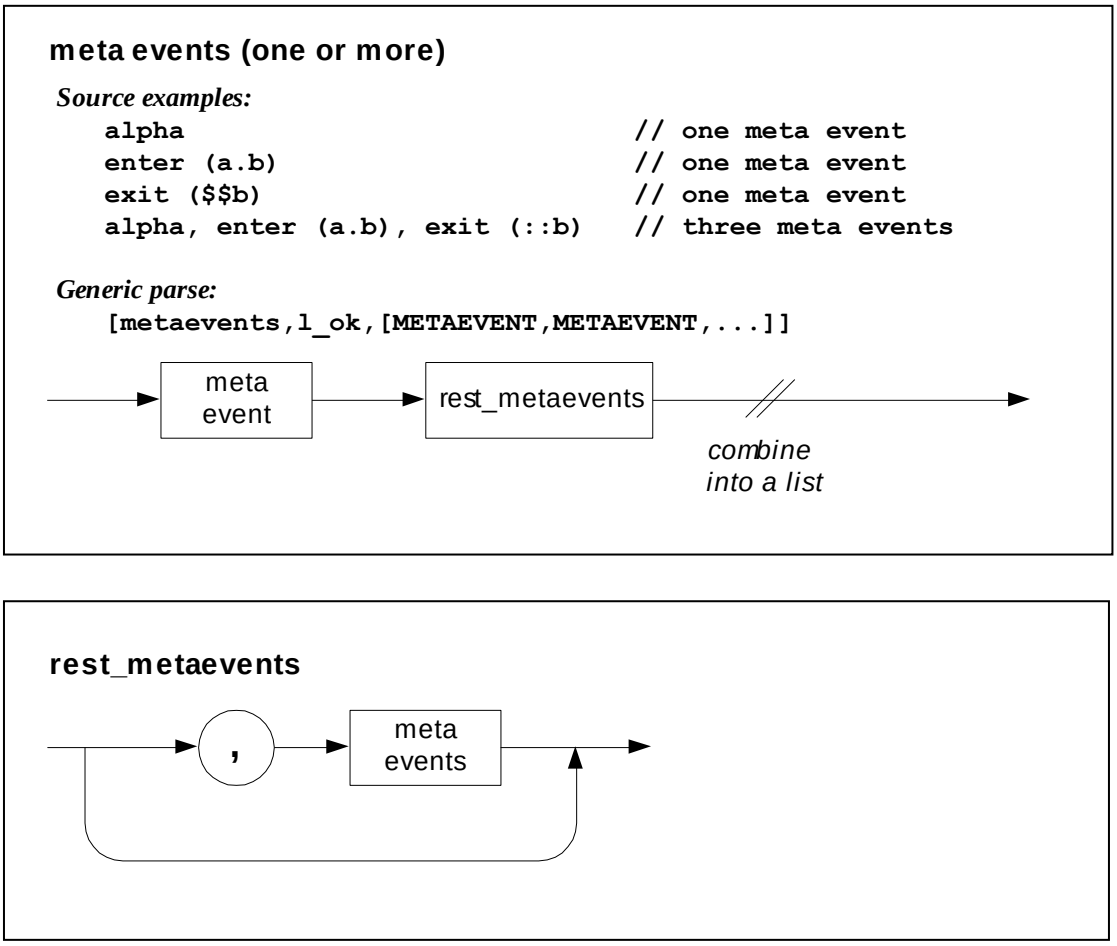


Figure 40. meta events

From test *[sc,sy,state,mevs,2]* $alpha(a.p1,$p2,p3), \ enter ($$a.b), \ exit (::c.d)	[metaevents,l_ok,[ME1,ME2,ME3]]], ME1=[metaevent,l_ok,[t_event,l_ok,TEV]], TEV=[EVTEXPR,TPARAMBLK], ME2=[metaevent,l_ok,[enter,XSTATE2]], ME3=[metaevent,l_ok,[exit,XSTATE3].

Table 20. Example of parsing "meta events"

meta event

Source examples

```
$alpha(p1,$a.p2)
enter(a.b)
exit(::)
```

Example parse:

```
[metaevent,l_ok,[EVT_EXPR,TPARAMBLK]]
[metaevent,l_ok,[enter,STA_EXPR]]
[metaevent,l_ok,[exit,STA_EXPR]]
```

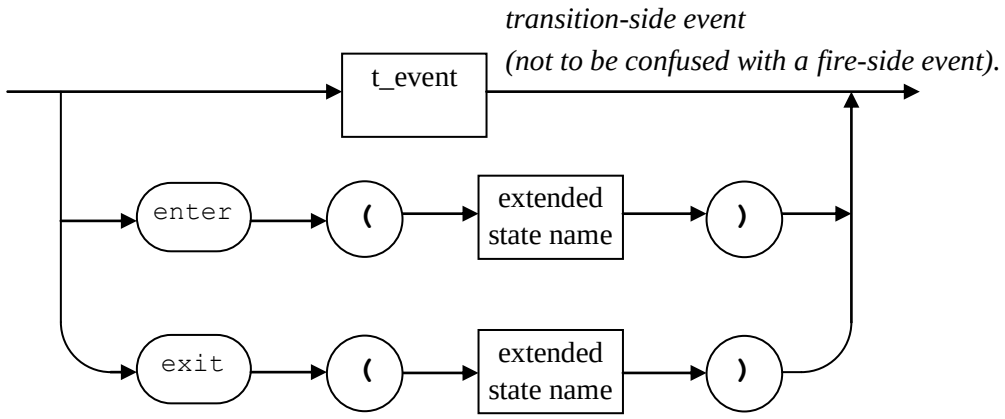


Figure 41. meta event

From test *[sc,sy,state,mev,1]* `$alpha(a.p1,$p2,p3)`	`[metaevent,l_ok,[t_event,l_ok,TEV]] ],` `TEV=[EVTEXPR,TPARAMBLK],` `EVTEXPR=[ex_evt_expr,[[ex_monadic,mback],` `[ex_id,alpha]]],` `TPARAMBLK=[tevt_paramblk,l_ok,` `[ [ex_var_expr,[[ex_dyadic,descend],` `[ex_id,a],[ex_id,p1]]],` `[ex_var_expr,[[ex_monadic,mback],[ex_id,p2]]],` `[ex_var_expr,[ex_id,p3]]].
From test *[sc,sy,state,mev,2]* `enter ( $$$aa.bb )`	`[metaevent,l_ok,[enter,XSTATE]] ],` `XSTATE=[xstate,l_ok,STAEXPR],` `STAEXPR= [ex_sta_expr,` `[ [ex_monadic,mback],` `[ [ex_monadic,mback],` `[ [ex_monadic,mback],` `[ [ex_dyadic,descend],` `[ex_id,aa],` `[ex_id,bb]]]]].

Table 21. Examples of parsing "metaevent"

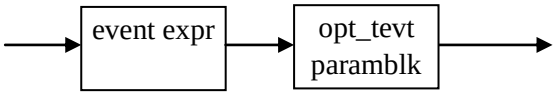
t_event

Source example

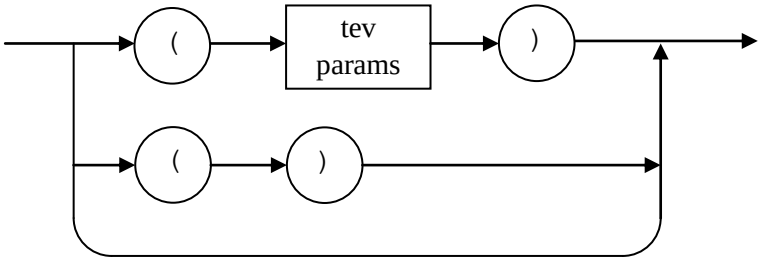
```
$alpha(a.p1,$p2,p3)
```

Parse:

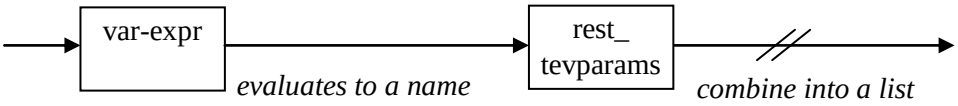
```
[metaevent,l_ok,[EVTEXPR,TPARAMBLK]]
```



opt_tevt_paramblk



tevparams



rest_tevparams

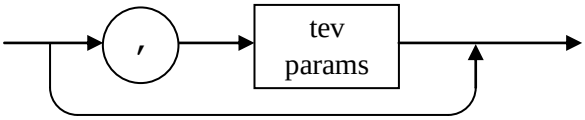


Figure 42. t_event

From test [sc,sy,state,tev,4] `$alpha(a.p1,$p2,p3)`	`[metaevent,l_ok,[EVTEXPR,TPARAMBLK]]` `  EVTEXPR=[ex_evt_expr,[[ex_monadic,mback],` `  [ex_id,alpha]]],` `  TPARAMBLK=[tevt_paramblk,l_ok,` `  [ [ex_var_expr,[[ex_dyadic,descend],` `  [ex_id,a],[ex_id,p1]]],` `  [ex_var_expr,[[ex_monadic,mback],[ex_id,p2]]],` `  [ex_var_expr,[ex_id,p3]]` `  ] ]`

Table 22. Example of parsing "t_event"

extended state

Source examples:

```
::a
$a%%b.p
```

Generic parse:

```
[xtate,l_ok,STATE_EXPRESSION]
```

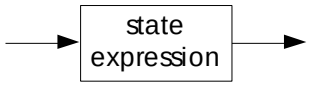


Figure 43. Extended state

a	[xstate,l_ok,[ex_sta_expr, [ex_id,a]]]
$a	[xstate,l_ok,[ex_sta_expr, [[ex_monadic,mback],[ex_id,a]]]]
::a	[xstate,l_ok,[ex_sta_expr, [[ex_monadic,mscope],[ex_id,a]]]]
a.b	[xstate,l_ok,[ex_sta_expr, [[ex_dyadic,descend],[ex_id,a],[ex_id,b]]]]
a%%b	[xstate,l_ok,[ex_sta_expr, [[ex_dyadic,dparent],[ex_id,a],[ex_id,b]]]]
$a%%b	[xstate,l_ok,[ex_sta_expr, [[ex_monadic,mback], [[ex_dyadic,dparent],[ex_id,a],[ex_id,b]]]]]

Table 23. Examples of parsing "extended state"

condition (optional)

Source example:

```
[$c==1 && !in($a)]
[#rubbish]
null
```

Generic parse:

```
[condition,l_ok,[ex_expr,EXPRESSION]]
[condition,l_er,['**Error: condition']]
[condition,l_ok,[]]
```

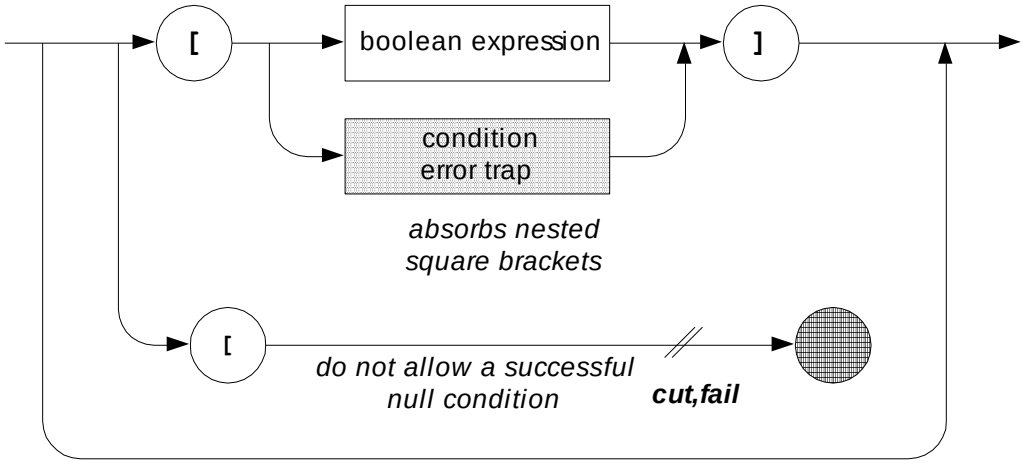


Figure 44. condition (optional)

`[$c==1 && !in($a)]`	`[condition,l_ok,` `  [ex_expr,` `  [[ex_dyadic,land],` `  [[ex_dyadic,eq],` `  [[ex_monadic,mback],` `  [ex_id,c]],` `  [ex_co,int,1]],` `  [[ex_monadic,lnot],` `  [[ex_dyadic,fcall],` `  [ex_id,in],` `  [[[ex_monadic,mback],` ` [ex_id,a]]]]]]]]
`[#rubbish]`	`[condition,l_er,[**Error: condition]]`
null	`[condition,l_ok,[]]`

Table 24. Examples of parsing "condition"

route

Source example

```
null                    // allows for actions only
-> c.d.(e/\f.g)         // destination only, /\ allowed
->a.b->c.d.(e/\f.g)     // orbit and destination
```

Generic parse:

```
[route,l_ok,[[],[]]]
[route,l_ok,[[],DEST]]
[route,l_ok,[ORBIT,DEST]]
```

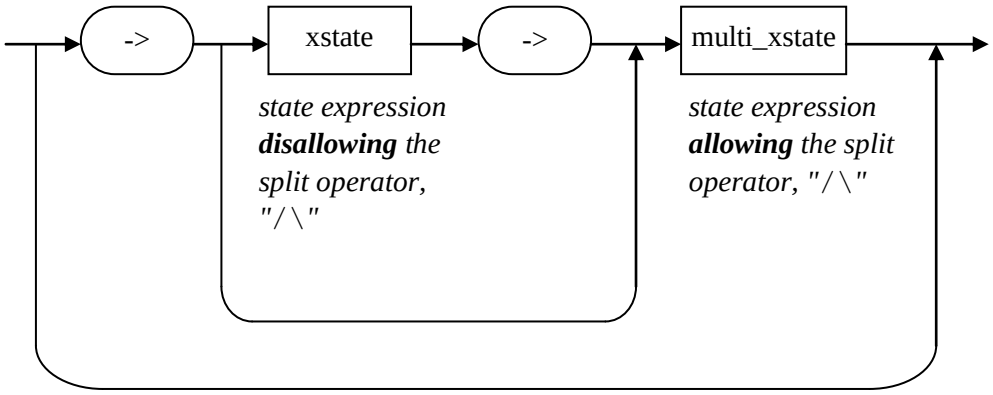


Figure 45. route

opt_destination

Source examples:

```
->a.b
null
```

Generic parse:

```
[optdest.l_ok,XSTATE]
[optdest,l_ok,[]]
```

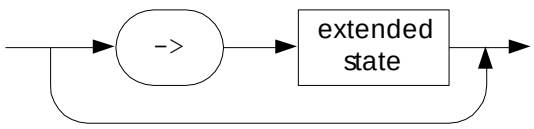


Figure 46. opt_destination

`->a.b`	`[route,l_ok,[[],[xstate,l_ok,[ex_sta_expr, [[ex_dyadic,descend],[ex_id,a],[ex_id,b]]]]]]`

`->a.b->c.d`	`[route,l_ok,` `  [ [xstate,l_ok,[ex_sta_expr,` `  [[ex_dyadic,descend],[ex_id,a],[ex_id,b]]]],` `  [xstate,l_ok,[ex_sta_expr,` `  [[ex_dyadic,descend],[ex_id,c],[ex_id,d]]]]]]`

Table 25. Examples of parsing "route"

opt_action_block

Source example:

```
{a+=b+c; fire alpha; if (a>b) {c=d;} else {e=f;}}}
{ #rubbish}  // trapped at action block level
null
```

Generic parse:

```
[optactblk,l_ok,[ACTION,ACTION,ACTION,...]]
[optactblk,l_er,['**Error: action block']]
[optactblk,l_ok,[]]
```

Note:

The parse does not add a parsing level; it transports the LSTATUS and PARSE of ***action block*** *to its own LSTATUS and PARSE.*

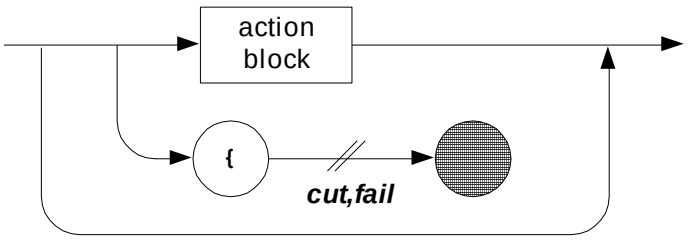


Figure 47. opt_action_block

From test *[sc,sy,state,oacb,1]* { x += 3 ; }	[optactblk,l_ok,[ACTION1]]], ACTION1=[action,l_ok,[action_as,l_ok,EXPR1]], EXPR1=[ex_expr,[[ex_dyadic,asplus],[ex_id,x], [ex_co,int,3]].
null	[optactblk,l_ok,[]]
{ #rubbish }	[optactblk,l_er,[**Error: action block]]

Table 26. Example of parsing "opt_action_block"

action block

Source example:

```
{a+=b+c; fire alpha; if (a>b) {c=d;} else {e=f;} }
{ #rubbish}
```

Generic parse:

```
[actionblk,l_ok,[ACTION,ACTION,ACTION,...]]
[actionblk,l_er,['**Error: action block']]
```

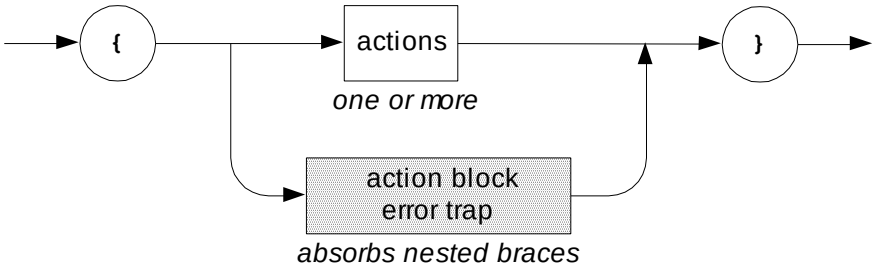


Figure 48. action block

Note: action block parse information is transported to an opt_action block.

From test *[sc,sy,state,acb,4]* `{ x += 3 ;  \` `deep_clear ( aa ); }`	`[actionblk,l_ok,[ACTION1,ACTION2]] ],` `ACTION1=[action,l_ok,[action_as,l_ok,EXPR1]],` `EXPR1=[ex_expr,[[ex_dyadic,asplus],` `[ex_id,x],[ex_co,int,3]]],` `ACTION2=[action,l_ok,[action_as,l_ok,` `[ex_expr,[[ex_dyadic,fcall],` `[ex_id,deep_clear],[[ex_id,aa]]]]]].

Table 27. Example of parsing "action block"

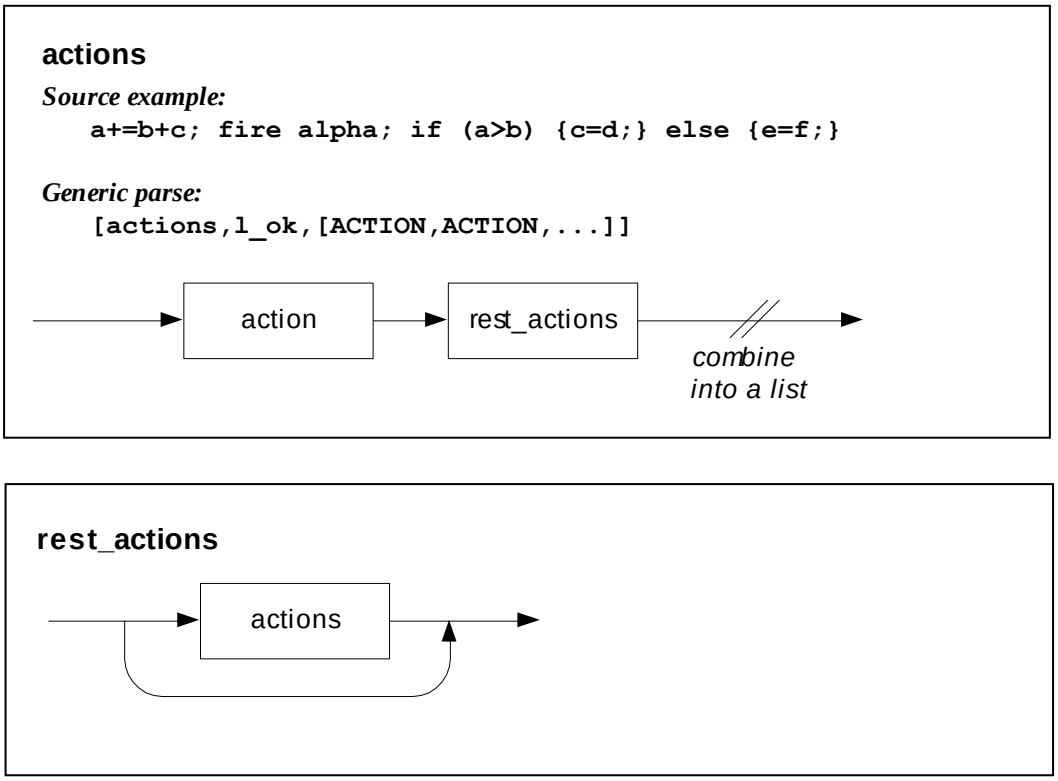


Figure 49. actions

Parse examples can be seen nested within the **opt_action_block** parse examples.

action

Source example:

```
a+=b+c;
fire alpha;
if (a>b) {c=d;} else {e=f;}
```

Generic parse:

```
[action,l_ok,SPECIFIC_ACTION]
```

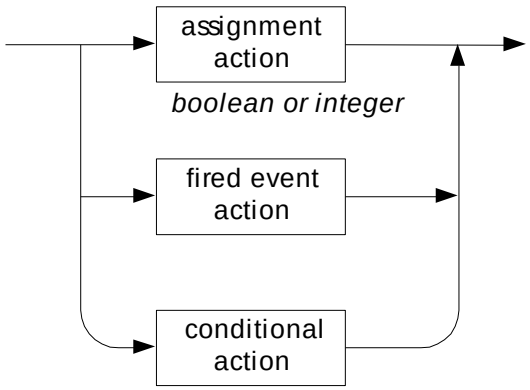


Figure 50. action

An earlier version of STATECRUNCHER included *clear* and *deep_clear* actions (to clear history). These are now implemented as standard functions.

`a+=b+c;`	`[action,l_ok,[action_as,l_ok,` `  [ex_expr,[[ex_dyadic,asplus],` `  [ex_id,a],` `  [[ex_dyadic,dplus],` `  [ex_id,b],[ex_id,c]]]]]]`
`From test` *`[sy,sc,state,ac,3]`* `fire alpha ;`	`[action,l_ok,E]],` `E=[action_ev,l_ok, [[param_ev,l_ok,` `  [[ex_evt_expr,[ex_id,alpha]],` `  [evt_paramblk,l_ok,[]]]]].`
`if (a>b)` `  {c=d;}` `else` `  {e=f;}`	`[action,l_ok,[action_co,l_ok,` `  [[ex_expr,` `  [[ex_dyadic,gt],` `  [ex_id,a],[ex_id,b]]],` `  [[action,l_ok,[action_as,l_ok,` `  [ex_expr,[[ex_dyadic,assign],` `  [ex_id,c],[ex_id,d]]]]]],` `  [[action,l_ok,[action_as,l_ok,` `  [ex_expr,[[ex_dyadic,assign],` `  [ex_id,e],[ex_id,f]]]]]]]]`

Table 28. Examples of parsing "action"

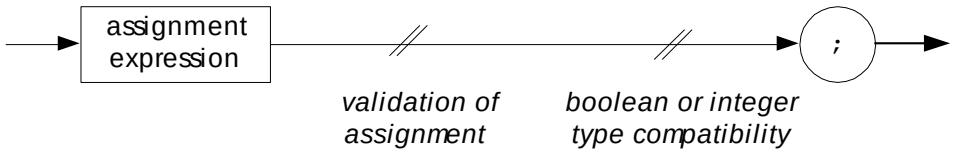
assignment action

Source example:

```
a+=b+c;
```

Generic parse:

```
[action_as,l_ok,ASSIGNMENT_EXPRESSION]
```



validation of assignment

boolean or integer type compatibility

Figure 51. assignment action

fired event action

Source example

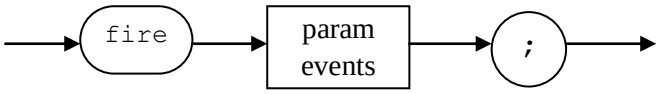
```
fire alpha,$beta(f1(fp1,$fp2),$$p2),gamma();
```

Example parse:

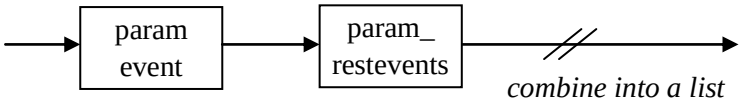
```
[g_ok,[action_ev,l_ok,[E1,E2,E3...]]]
```

where

```
E2=[param_ev,l_ok,[EVEXPR2,PBLK2]],
EVEXPR2=[ex_evt_expr,EVT_EXPR],
PBLK2=[evt_paramblk,l_ok,[P1,P2]],
P1=[ex_expr,EXPR1],
P2=[ex_expr,EXPR2]
```



param_events



param_restevents

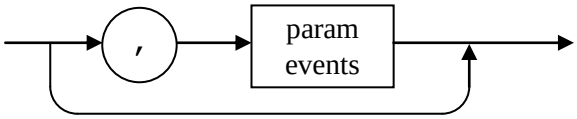


Figure 52. fired event action

param_event

Source example

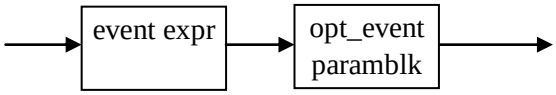
```
    $beta(f1(fp1,$fp2),$$p2)
```

Example parse:

```
     [param_ev,l_ok,[EVEXPR2,PBLK2]]
```

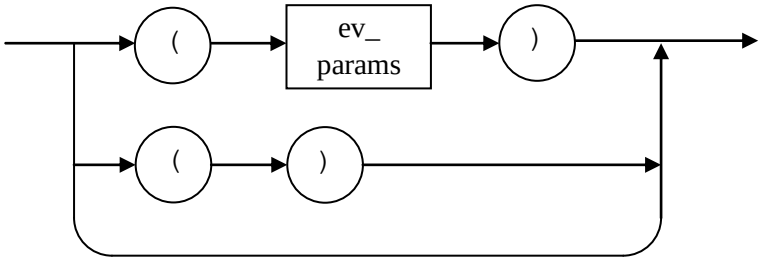
where

```
     EVEXPR2=[ex_evt_expr,EVT_EXPR],
     PBLK2=[evt_paramblk,l_ok,[P1,P2]],
     P1=[ex_expr,EXPR1],
     P2=[ex_expr,EXPR2]
```



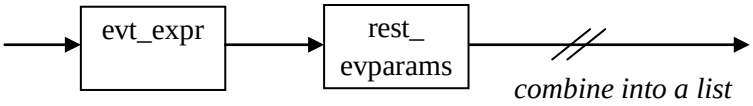
---

opt_evt_paramblk



---

ev_params



---

rest_evparams

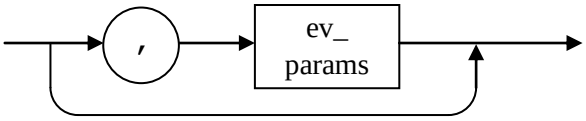


Figure 53. param_event

conditional action

Source example:

```
if (a>b) {c=d;} else {e=f;}
```

Generic parse:

```
[action_co,l_ok,[CONDITION,IFACTIONS,ELSEACTIONS]]
```

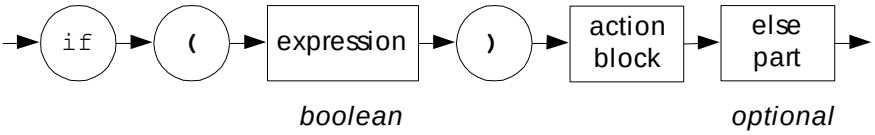


Figure 54. conditional action

else part

Source example:

```
else {e=f;}
null
```

Generic parse:

```
[elsepart,l_ok,[ACTION,ACTION,...]
[elsepart,l_ok,[]]
```

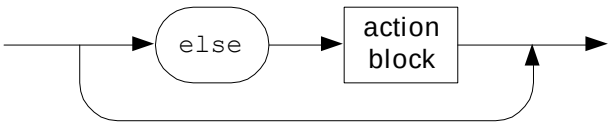


Figure 55. else part

label block (optional)

Source examples:

```
[key1=23,key2=xyz,key3=3.56]
[#rubbish ]
null
```

Generic parse:

```
[labelblk,l_ok,[LABEL,LABEL,...]]
[labelblk,l_er,['**Error: label block']]
[labelblk,l_ok,[]]
```

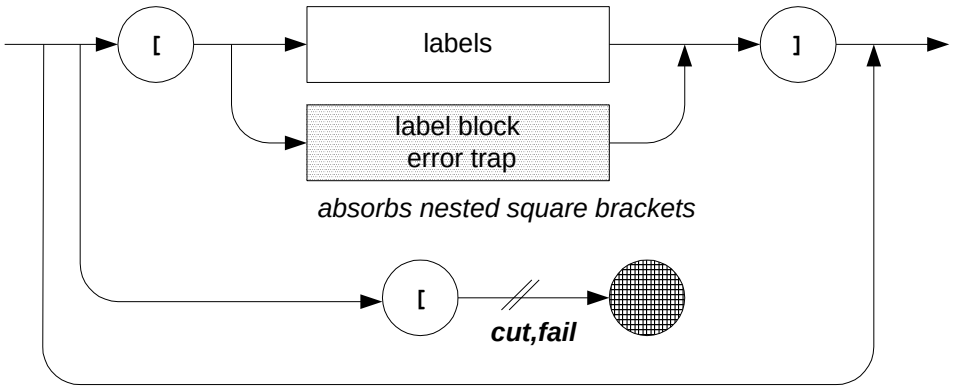


Figure 56. label block

`[lk_time=- 7, lk_cost=6]`	`[labelblk,l_ok,` `  [ [label,l_ok,[lk_time,` `  [ex_expr,` `  [[ex_monadic,mminus],[ex_co,int,7]]]]],` `  [label,l_ok,[lk_cost,` `  [ex_expr,[ex_co,int,6]]]]]`
`[lk_time=- 7, #rubbish]`	`[labelblk,l_er,[**Error: label block]]`

Table 29. Examples of parsing "label_block"

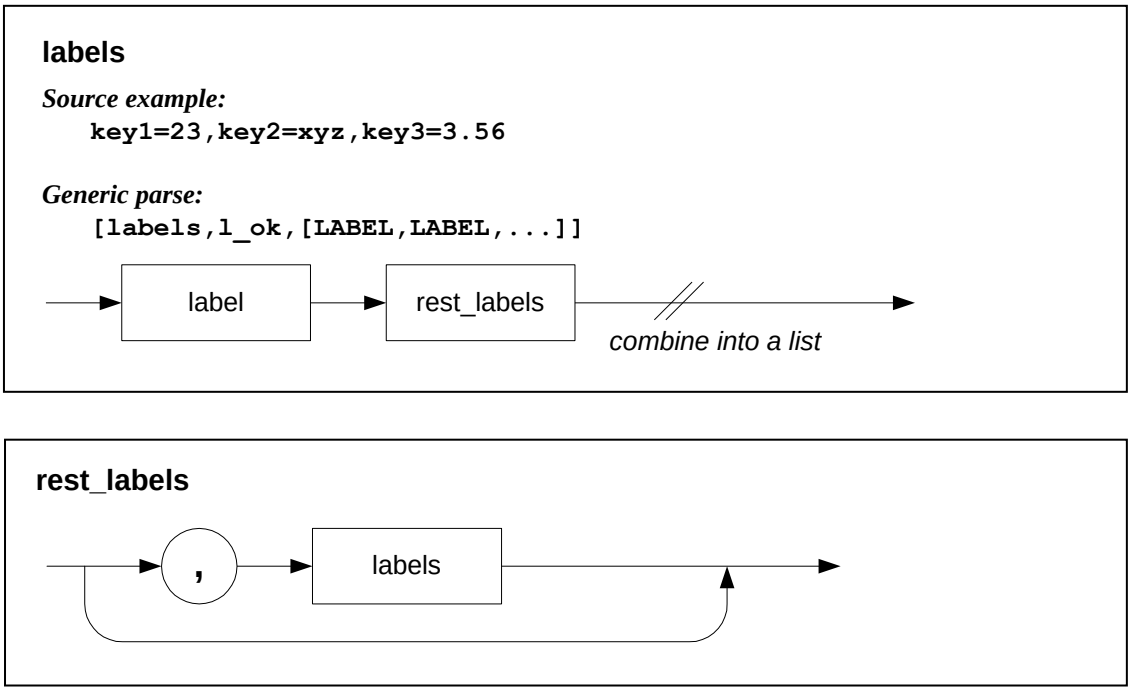


Figure 57. labels

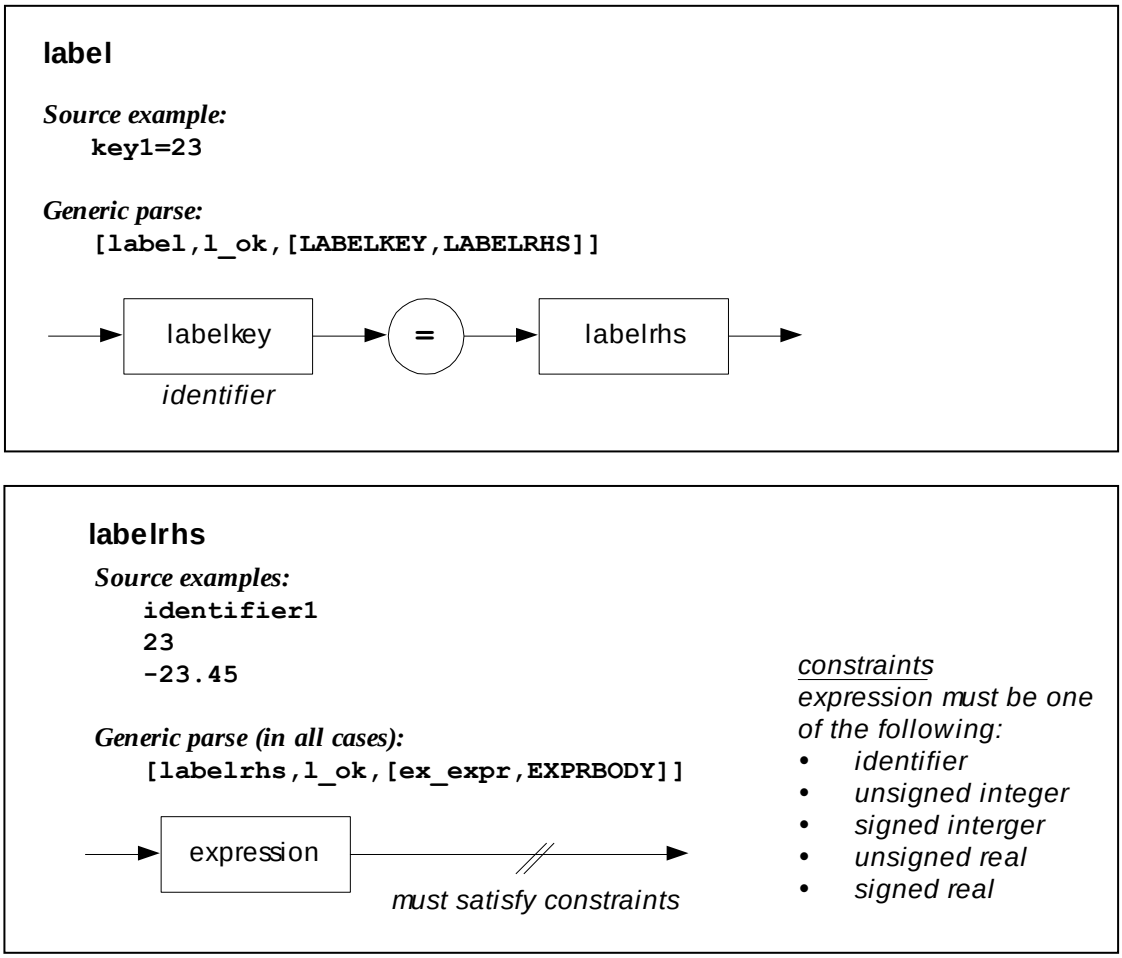


Figure 58. label

labelrhs type	example	parse
identifier	`highcost`	`[ex_expr,[ex_id,highcost]]`
unsigned integer	`23`	`[ex_expr,[ex_co,int,23]]`
negated integer	`-23`	`[ex_expr,` `  [[ex_monadic,mminus],` `  [ex_co,int,23]]]`
unsigned real	`23.34`	`[ex_expr,[ex_co,real,23.45]]`
negated real, (exponent notation)	`-2.34E-3`	`[ex_expr,` `  [[ex_monadic,mminus],` `  [ex_co,real,0.00234]]]`

Table 30. Constraints: Examples of permissible expression signatures for labels

4. Expressions

4.1 Categories of expression

STATECRUNCHER Syntax contains three categories of expression:

- Arithmetic expressions
- Scoping expressions
- Mixed expressions

Arithmetic expressions yield a numerical result; logical terms are represented numerically. Strings terms if implemented in a future version, can be regarded as a data type and so belong in the category of arithmetic expressions.

Scoping expressions yield the name of an item; the item may be a data variable or some other entity (tag, PCO, event, state).

In principle, arithmetic expressions and scoping expressions are separate concepts. Apart from mixed expression requirements, they *could* even use the same symbols for different operations. However, in STATECRUNCHER, arithmetic and scoping operators are distinct. A second fundamental difference is that the *evaluation* of a (sub-)expression depends on whether it is to be regarded as an arithmetic expression or a scoping expression. Arithmetic expressions yield a value; scoping expressions yield a name.

Mixed expressions arise because

- arithmetic expressions may include scoping operators, e.g. **`$x+y`**, where `$` alters the scope of x.
- arithmetic expressions may include special functions that take a scoping-expression argument, e.g. **`in(a.b)`**. The meaning of this is to test whether the state denoted by a relative path **`a.b`** is currently occupied. It is a boolean term, but the argument to the function is a scoping expression that requires evaluation to a state name, not a numerical value. The function **`in`** then examines whether the state is occupied. We see that the function `in()` takes a different kind of argument to arithmetic functions, and it is evaluated in a fundamentally different way.

GP4 supports parameterization of expressions in that it allows an operator set to be specified for use with the parse of any particular expression. Purely scoping expressions can be parsed using an operator set that only contains scoping operators. How should mixed expressions be parsed?

One solution would be to parse a mixed expression in a way that labels which parts are arithmetic and which parts are scoping expressions. However, this would require a special expression parser, which would know when to expect which kind of (sub-)expression within an expression. This could be done, but it would mean working with a variant on the ordinary GP4 expression parser. By default, there is in general no direct way to support mixed expression parsing of an expression such as

```
flag1 && !in(a.b) || flag2
```

An alternative approach, (the one taken), makes use of the fact that in STATECRUNCHER, arithmetic expressions always allow scoping operations as well, so one combined operator set includes arithmetic and scoping operators. Purely scoping expressions also occur, and these just use their own operator set. Mixed expressions are *parsed* as just one kind of expression, which we could call "extended-arithmetic". Scoping (sub-)expressions will be recognized at *evaluation* time.

The way the `in()` example is handled in STATECRUNCHER is that the expression is parsed as though `in()` were a regular function call, but its evaluation handler traps the call to it at the dyadic **`fcall`** operator level. This is in contrast to regular functions, which are handled by the generic **`fcall`** handler which evaluates the arguments, looks up the name of the Prolog implementation predicate, and calls it with those arguments.

The parse of the above example is :

```
[ex_expr,
   [[ex_dyadic,lior],
      [[ex_dyadic,land],
         [ex_id,flag1],
         [[ex_monadic,lnot],
             [[ex_dyadic,fcall],
                 [ex_id,in],
                 [[[ex_dyadic,descend],
                     [ex_id,a],[ex_id,b]]]]]],
      [ex_id,flag2]]]
```


If we examine what the **`fcall`** handler does when provided with function `in`, compared with the handler for a regular function, such as `maximum`, we have the following picture, revealing the differences:

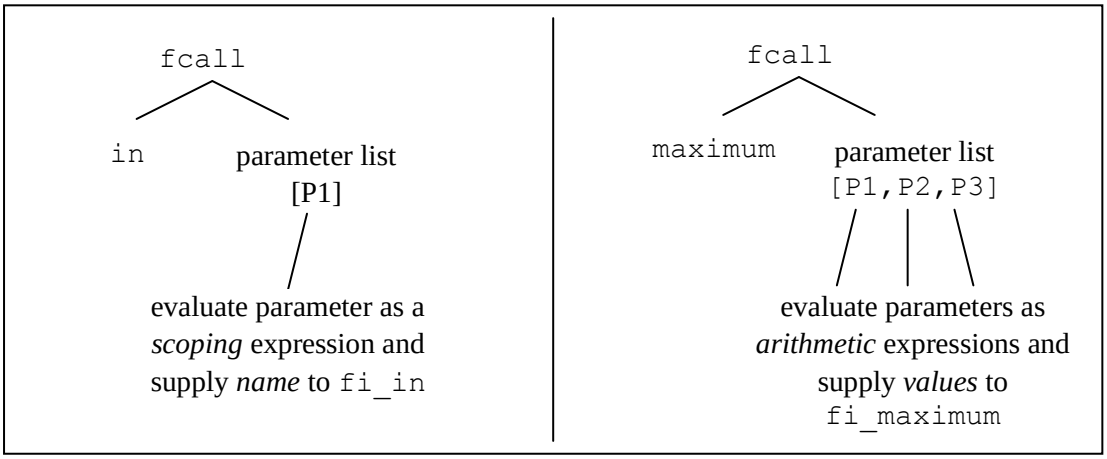


Figure 59. Handling fcall

4.2 Operator set customization

The GP4 expression parser parses expressions without regard to type compatibility, so no distinction is made between a ***boolean expression*** and an ***integer expression*** on the initial parse. It is left as a task for a ***validator*** to ensure that the result of an expression is of the required type. The validator may also ensure that the operators used in the expression are acceptable in the places where they occur, if there are restrictions on this.

However, the GP4 expression parser can be tailored to an expression in various ways:

- Specifying the operator set with which to work.
- Specifying the starting operator precedence level.

The following operator sets have been defined for STATECRUNCHER

operator set	**purpose**
sc	numeric, boolean *and scoping* operators
scb	*RESERVED for boolean operators only*
sct	operators for specifying tagnames
scv	operators for specifying variable names
scp	operators for specifying PCOs
sce	operators for specifying events
scs	operators for specifying states
additional sets reserved	*e.g. for transition description expressions (which are currently implemented without using operators)*

Table 31. Operator sets for STATECRUNCHER

STATECRUNCHER operator precedence levels have been standardised as follows:
level 1: The lowest level. This is used for the comma operator
level 2: The level of assignment operators
level 3+: Other operators

This gives the following useful variations on starting operator precedence level:

- level 1: all operators are active, including the comma
- level 2: the comma operator is not active
- level 3: the comma and assignment operators are not active

4.3 Arithmetic operators

The following operators are supported at a parsing level; those not implemented in version 1.0 at an execution level are marked as such.

Operation	Symbol	Definition parameter (as per GP4)	Executn. support?
Primary Suffixes			
Array indexing	`[]`	`[op,18,[f,argi],sqbr]` `[op,18,[continued],sqbr]`	
Function call	`()`	`[op,17,[f,argl],fcall]` `[op,17,[continued],fcall]`	
Various monadic			
plus	`+`	`[op,16,[f,y],mplus]`	
minus	`-`	`[op,16,[f,y],mminus]`	
logical not	`!`	`[op,16,[f,y],lnot]`	
post increment		`[op,16,[y,f],postinc]`	
post decrement		`[op,16,[y,f],postdec]`	
pre increment		`[op,16,[f,y],predinc]`	
pre decrement		`[op,16,[f,y],predec]`	
Multiplicative			
multiplication	`*`	`[op,15,[y,f,x],xmul]`	
division	`/`	`[op,15,[y,f,x],xdiv]`	
modulo	`%`	`[op,15,[y,f,x],mod]`	
Additive			
addition	`+`	`[op,14,[y,f,x],dplus]`	
subtraction	`-`	`[op,14,[y,f,x],dminus]`	
Shifting			

arithmetic shift right	>>	[op,13,[y,f,x],asr]	No
arithmetic shift left	<<	[op,13,[y,f,x],asl]	No
Relational			
less than or equal	<=	[op,12,[y,f,x],le]	
greater than or equal	>=	[op,12,[y,f,x],ge]	
less than	<	[op,12,[y,f,x],lt]	
greater than	>	[op,12,[y,f,x],gt]	
equal	==	[op,11,[y,f,x],eq]	
not equal	!=	[op,11,[y,f,x],ne]	
Bitwise			
bitwise and	&	[op, 10,[y,f,x],band]	No
bitwise xor	^	[op, 9,[y,f,x],bxor]	No
bitwise eqv	~^	[op, 9,[y,f,x],beqv]	No
bitwise incl or	|	[op, 8,[y,f,x],bior]	No
Logical			
short-circuit and	&&	[op, 7,[y,f,x],land]	
xor	^^	[op, 6,[y,f,x],lxor]	
equivalence	!^^	[op, 6,[y,f,x],leqv]	
short-circuit or	||	[op, 5,[y,f,x],lior]	
Arithmetic conditional			
arithmetic if	? :	[op,3,[x,f,x,g,y],aif] [op,3,[continued],aif]	No
Assignment			
assign	=	[op,2,[x,f,y],assign]	
exponentiate-assign	**=	[op,2,[x,f,y],aspwr]	No
multiply-assign	*=	[op,2,[x,f,y],asxmul]	
divide-assign	/=	[op,2,[x,f,y],asxdiv]	
modulo-assign	%=	[op,2,[x,f,y],asmod]	
add-assign	+=	[op,2,[x,f,y],asplus]	
subtract-assign	-=	[op,2,[x,f,y],asminus]	
bitwise-and-assign	&=	[op,2,[x,f,y],asband]	No
bitwise-xor-assign	^=	[op,2,[x,f,y],asbxor]	No
bitwise-equiv-assign	!^=	[op,2,[x,f,y],asbeqv]	No
bitwise-incl-or-assign	|=	[op,2,[x,f,y],asbior]	No

arith shift right assign	>>=	[op,2,[x,f,y],asasr]	No
arith shift left assign	<<=	[op,2,[x,f,y],asasl]	No
Sequence			
sequence	,	[op,1,[y,f,x],seq]	No

Table 32. Arithmetic operators

4.4 Scope and scoping operators

Lucas [CHSM, p.21] describes how CHSM state names are specified. The following figure illustrates non-local transitions

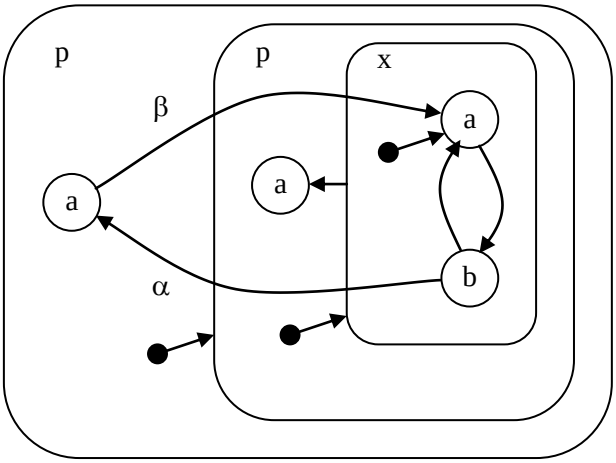


Figure 60. Non-local transitions

Lucas describes three ways to specify the destination state of an "out-scoping" transition *(my terminology)*, such as the transition on event α in the above figure.

- Absolute scoping `::p.a`
- Back-out scoping: `..a`
- Back to parent scoping `.p.a`

The exact semantics of the operators in the third example are not given.

Lucas does not explicitly describe how to specify the destination state of an "in-scoping" transition such as the transition on event β in the figure. However, from examples (e.g. [CHSM, p.24] cluster `Display`, a transition to `NotTime.Counter`), it appears that our example would be

```
beta->p.x.a
```

However, this shares its operator symbol with respect to out-scoping specification.

We reconsider the scoping issue, offering as much flexibility as is practicable, since it is anticipated that scoping will play an important part when state models of subsystems are combined into higher-level system models.

Design of scoping operators

The above discussion leads us to design some operators to unambiguously cover all the functionality described by Lucas. We desire individual operators for

- back-out one level
- back-out to a named parent
- back out to the outermost level
- enter one named level

Before designing these operators, we recall that scope is already present by default. PCO's, events, tagnames and variables can be declared at any level in the machine hierarchy, e.g.

```
statechart sc(p)
  PCO pco1;
  event alpha@pco1;
  enum colour {red=1,green=3,blue=4};
  colour mycolour=red;

  cluster p(q)
     PCO pco1;
     event alpha@pco1;
     enum colour {red=1,green=3,blue=4};
     colour mycolour=red;

     cluster q(a,b)
        PCO pco1;
        event alpha@pco1;
        enum colour {red=1,green=3,blue=4};
        colour mycolour=red;

        state a    ;
        state b    ;
```

Although various names have been used repeatedly in this example, this is permitted because each item has a scope associated with it. The variable `mycolour` has a scope associated with it in two ways:

- It is defined in a certain scope
- Its type contains the scope of the tagname used in the variable declaration.

When an item is accessed, the most local item found is taken as the one intended. An outward search takes place to locate the item at successively more global levels if the item is not found at a local level.

4.4.1 Scoping operators - general design

Various items in STATECRUNCHER can be declared or accessed outside their natural scope by means of scoping operators. These operators can be used to form scoping expressions.
The operators (with their implementation names) are:

- `$` (mback) back out one level from the current scope
- `%%` (dparent) back out to a named parent
- `::` (mscope) back out to the outermost level
- `.` (descend) enter one named level deeper

4.4.2 The mback operator "$"

This is a monadic operator. The term "`$a`" means: "`a`" as it would be accessed if addressed in the hierarchical state one level more global than the current one. This operator is right associative, so "`$$$a`" takes us back three levels.

4.4.3 The dparent operator "%%"

This is a dyadic operator. The term "`a%%b`" means: back out of the current level until a hierarchical machine named "`a`" is found. At least one level is always backed out. Then address "`b`" in that level. The operator is right associative, so that the expression "`a%%b%%c`" reads: back out to level "`a`", then back out from there to level "`b`", and evaluate "`c`" in that scope.

4.4.4 The mscope operator "::"

This is a monadic operator. The term "`::a`" means: address "`a`" at the statechart level.

4.4.5 The descend operator "."

This is a dyadic operator. The term "`a.b`" means: enter the immediately deeper hierarchical level "`a`" and address "`b`" in that scope. The operator is right associative, which means that the expression "`a.b.c`" reads: enter "`a`", then "`b`" and address "`c`" in that scope.

4.4.6 Combining scoping operators

The monadic and dyadic operators combine with dyadic operations binding tighter.

4.4.7 Effect of scoping operators on declarations

Scoping operators can be used when *accessing* (rather than *declaring*) any item, i.e. PCO's, events, tagnames and variables and states.

Scoping operators can be used in *declarations* of PCO's, events, tagnames and variables (but not states). They have the effect of declaring the item as if it were an ordinary declaration in another part of the hierarchy. For example, to declare various items as if they all belonged one level up in the hierarchy:

```
PCO $pco1;
event $alpha;
enum $colour {red=1,green=3,blue=4};
$colour $mycolour;
```

In the variable declaration

```
$colour $mycolour;
```

the variable has a scope determined by its own scoping expression, and a type affected by the scoping expression on its tagname.

These operators should be composable into a scoping expression. We need to select

- operator symbols

and facilitate expression building by our choice of:

- operator precedence
- operator associativity

Before selecting these operator properties, it is good to realise that there is a major difference in the way scoping operators work compared with, say, arithmetic operators. Arithmetic operators apply their own operation ***after*** evaluating their arguments (which latter they do by a recursive call to the evaluator). For example, a simplified predicate to evaluate the monadic minus operation on a parameter P1 might be:

```
ev_expr(MPATH,[[ex_monadic,mminus],P1],V):-
      ev_expr(MPATH,P1,VV), /* evaluate argument  */
      VV=[ex_co,TYPE,K],    /* analyse evaluation */
      KK is -K,             /* operator's own action */
      V=[ex_co,TYPE,KK],    /* set up return value   */
      !.
```

Note that P1 is evaluated by a recursive call before the negation takes place (`KK is -K`).

Similarly for dyadic operations (simplified):

```
ev_expr(MPATH,[[ex_dyadic,dminus],P1,P2],V):-
      ev_expr(MPATH,P1,VV1),    /* evaluate P1 */
      ev_expr(MPATH,P2,VV2),    /* evaluate P2 */
      VV1=[ex_co,T1,K1],    /* T1=type, K1=value     */
      VV2=[ex_co,T2,K2],    /* T2=type, K2=value     */
      KK is K1-K2,          /* operator's own action */
      ev_gtype(T1,T2,TT),   /* get most general type */
      V=[ex_co,TT,KK],      /* set up return value   */
      !.
```

In these predicates, MPATH is the machine path (i.e. scope) in which the evaluation takes place. Termination of the recursion takes place at a terminal item, such as an identifier (whose value is then obtained from a table or database).

Now when it comes to scoping operators, they must perform their own operation - i.e. changing the scope - ***before*** evaluating their arguments. It will be seen that this has implications for the choice of precedence and associativity. Here is what the back-out operator does

```
ev_expr([HMPATH|TMPATH],[[ex_monadic,mback],P1],V):-
    ev_expr(TMPATH,P1,V), /* remove head of machine path */
    !.
```

The predicate first modifies the supplied machine path. It effectively removes the head of a list describing the machine path, the head being the most local part of the path. Then it performs the recursive call to have its parameter, P1, evaluated in the new scope.

Similarly, for a dyadic scoping operator. The following operator takes its first argument (P1) as a required addition to the machine path, so as to make the scope more local. The second argument (P2) is then evaluated recursively in the new scope.

```
ev_expr(MPATH,[[ex_dyadic,descend],P1,P2],V):-
      P1=[ex_id,ID],          /* ID = desired extension  */
      MPATH2=[ID|MPATH],      /* extend the machine path */
      ev_expr(MPATH2,P2,V),   /* recursive evaluation    */
      !.
```

We now tabulate the operators implemented:

Operator syntax	Operator definition `[op,`*`prec`*`,`*`assoc`*`,`*`name`*`]`	Input argument 1	Input argument 2	Action
`$`*`arg1`*	`[op,19,[f,y],mback]`	misc.	n/a	evaluate arg1 in the new scope, which is backed-out one level.
`::`*`arg1`*	`[op,19,[f,y],mscope]`	misc.	n/a	evaluate arg1 in the new scope, which is at the outermost level.
`arg1.arg2`	`[op,20,[x,f,y], descend]`	machine path element	misc.	evaluate arg2 in the new scope, which enters level arg1 w.r.t the current scope
`arg1 %%arg2`	`[op,20,[x,f,y], dparent]`	machine path element	misc.	evaluate arg2 in the new scope, which backs out one level anyway, and then as far as the first occurrence of machine path element arg1.

Table 33. Scoping operators

Note that the "`.`" (descend) and "`%%`" (dparent) operators are right associative. This means that an expression such as

```
aa.bb.cc.dd
```

is equivalent to

```
aa.(bb.(cc.dd))
```

Bearing in mind the reasoning about scoping operators performing their operation ***before*** evaluating their arguments, the above expression will behave as follows:

- add `aa` to the machine path, making it one level deeper than the caller's level
- add `bb` to the machine path, making it one level deeper than as above
- add `cc` to the machine path, making it one level deeper still
- evaluate `dd` in this new scope

Similarly

```
aa%%bb%%cc%%dd
```

will evaluate `dd` in the scope that backs out to the first occurrence of `aa` (cutting blindly through `bb`'s and `cc`'s if they occur), then backs out further to the next occurrence of `bb` (cutting blindly through `cc`'s if they occur), then backs out further to the first occurrence of `cc`, and finally evaluates `dd` in this scope.

Similarly, the "`::`" (mscope) and monadic "`$`" (mback) operators are right associative. This means that expressions consisting of multiple monadic operators can be composed simply:

```
$$$aa
```

which is equivalent to

```
$($($aa))
```

backs out three levels then evaluates `aa`.

The expression

```
::$aa
```

backs out to the outermost shell, then backs out one more level (which in STATECRUNCHER v1.0 is admissible, as the "`::`" operator backs out to the `statechart` level, from which it is possible to back out once more to the absolute level.

The expression

```
$::aa
```

would normally be pointless, as it backs out one level before performing a global back-out operation.

These monadic and dyadic operators combine with dyadic operations binding tighter, so that

```
$$aa.bb.cc
```

which is equivalent to

```
$($(aa.(bb.cc)))
```

means back out two levels, then enter `aa` then enter bb then enter `cc`. The rule is emerging that **the expression is to be interpreted as a sequence of actions in left-to-right reading order**.

One consideration is that dyadic operators have a higher precedence than monadic ones, which is fine for expressions such as

```
$$aa.bb.cc
```

but it means that brackets are needed for adjacent dyadic-monadic accumulations, e.g.

```
cc%%($$dd.var2)
```

which is to be read as: back-out to parent `cc`, then back out twice more, then descend into `dd`, then evaluate `var2` in this scope.

Scoping operators have a higher precedence than non-scoping ones. An example of a combined expression, extending the above example, is:

```
var1 + cc%%($$dd.var2)
```

which is to be read as: evaluate `var1`, back-out to parent `cc`, then back out twice more, then descend into `dd`, then evaluate `var2` in this new scope, then finally add together with the evaluation of `var1`.

Adjacent expressions

A constraint on the choice of operator symbols is that if two expressions follow in succession, they should not be accidentally parsable as one expression. This situation could arise in variable declarations, where the tagname and variable name are both scoping expressions. This makes it impossible to select the dot "`.`" for the mback operator, because then the declaration

```
.colour .mycolour =red;
```

would cause a parse of the tagname as

```
.colour.mycolour
```

and no variable name would be found.

Another alternative for the mback operator that was considered is "`@`". This would be possible, but it would be confusing in event declarations such as

```
event alpha@@pco1;
```

The choice of "`$`" has advantages and disadvantages

- advantage: it is a single symbol, which reads well in expressions such as `$$$var1`.
- advantage: it is not in use nor is it a candidate for alternate use as an operator.
- disadvantage: some systems allow the `$` in identifier names, typically as an extension to C-style names for system variables. STATECRUNCHER could still accommodate this, but mback operations would require white space: `$$$ $var2`.

Other symbols considered were

- "`\`" (backslash). In Windows path names, this is more like the descend operator. In "C" is it used to start octal digits, although this would not appear to clash. In STATECRUNCHER, the backslash is at the end of a line is currently a line-continuation character.
- "`#`" (hash). In "C" macros it is used as a string producer. A double hash is used for token-pasting.

- "^" (caret). This has the advantage of reading well ("go up"). In "C" it is the dyadic bitwise-or operation, and it may be wise to keep this operator available to STATECRUNCHER.

4.5 Items parsed as expressions

The following items are expressions:

- Tagnames in **declaration** (`enum` statement) / **usage** (variable declaration)
- Variables in **declaration** / **usage** (e.g. initialization, condition, action, label)
- PCO's in **declaration** / **usage**
- Events in **declaration** / **usage**
- States in **usage** ("declaration" is determined by the machine hierarchy)

This means that there is opportunity to access, and even define, items either less locally or more locally than the current scope.

Tagnames, variables, PCOs and events of a more global scope are implicitly in scope, unless masked by a more local homonym.

States must always be specified precisely, except that the "back to parent" operator will search for a parent state.

It is recommended that non-local scoping should be used sparingly, especially non-local declarations. In any case exceptional scoping should not be used gratuitously, but only when composition of subsystem models requires it.

We now discuss the semantics of scoped expressions.

Let us first define a machine context for the expressions to be considered in the following subsections; we take machine path **`sc.p.q.r.s`**

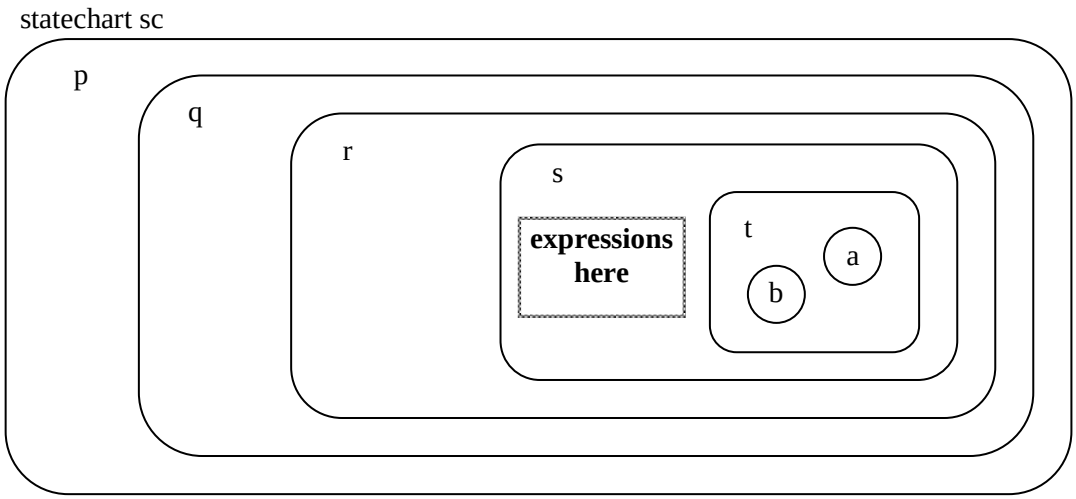


Figure 61. Scoping example

Note: the internal representation of machine path **sc.p.q.r.s** is **[s,r,p,q,sc]**. In the examples that follow, the **sc.p.q.r.s** style notation has been used, including in symbol table entries. This format may differ in representation from what is output by the STATECRUNCHER software.

In the following discussions, we will use the $ operator as a typical scoping operator; it defines a machine path that is one level shallower than the current machine path. The effect of a scoping expression is best illustrated by reference to the symbol table entry. All STATECRUNCHER symbol table entries are of the form

symbol, machine-path, type, value

The machine-path is the effective level at which the symbol is defined. There can be several identical symbols with different machine paths. The symbol that is actually *referred* to in any context is the "nearest" one working outwards from the current machine path.

4.5.1 Tagname declarations

Tag declarations are used to define ranges and enumerators:

Syntax

enum tagexpression {integer,..,integer};
enum tagexpression {enumerator-list};

Examples

Suppose in the above-mentioned machine context (`sc.p.q.r.s`) we have

```
enum busroutes {112,..,118};
enum colour {red,green=3,blue};
```

Without the use of any scoping operators, these tagnames and enumerators are entered in the symbol table as follows

name	machine path	type	value
busroutes	sc.p.q.r.s	[typedecl,range]	[112,118]
colour	sc.p.q.r.s	[typedecl,enumerated]	[0,3,4]
red	sc.p.q.r.s	[enumerator,colour]	0
green	sc.p.q.r.s	[enumerator,colour]	3
blue	sc.p.q.r.s	[enumerator,colour]	4

Table 34. Unscoped tagnames in symbol table

Now the effect of a scoping expression on the tagname is to define a new machine path for

- the tagname
- the enumerators

Scoping operations are not permitted on *enumerators* in a tagname declaration.

Examples

```
enum $busroutes {112,..,118};
enum $colour {red,green=3,blue};
```

name	machine path	type	value
busroutes	sc.p.q.r	[typedecl,range]	[112,118]
colour	sc.p.q.r	[typedecl,enumerated]	[0,3,4]
red	sc.p.q.r	[enumerator,colour]	0
green	sc.p.q.r	[enumerator,colour]	3
blue	sc.p.q.r	[enumerator,colour]	4

Table 35. Symbol table entries for scoped tagnames

Note the difference in machine path due to the $ operator.

4.5.2 Variable declarations

A variable declaration specifies the type:

Syntax

typeexpression variableexpression;
typexpressione variableexpression = expression;
bool variableexpression;
bool variableexpression = expression;

Examples

```
busroutes  myroute = 115;
colour     my_tie_col =red;
busroutes  yourroute = myroute +1;
bool       b1 = true;
```

We ignore the issue of type compatibility of "`myroute +1`" for the moment.

Whether or not the declarations of the tagnames are in the same machine context as these variable declarations, these variables are entered in the symbol table as follows:

name	machine path	type	value
b1	sc.p.q.r.s	[vardecl,[bool]	true
myroute	sc.p.q.r.s	[vardecl,[enumtype,[route,[sc.p.q.r.s]]]]	115
my_tie_col	sc.p.q.r.s	[vardecl,[enumtype,[colour,[sc.p.q.r.s]]]]	0
yourroute	sc.p.q.r.s	[vardecl,[enumtype,[route,[sc.p.q.r.s]]]]	116

Table 36. Variables in symbol table

The tagnames may be defined at a shallower scope:

statechart sc

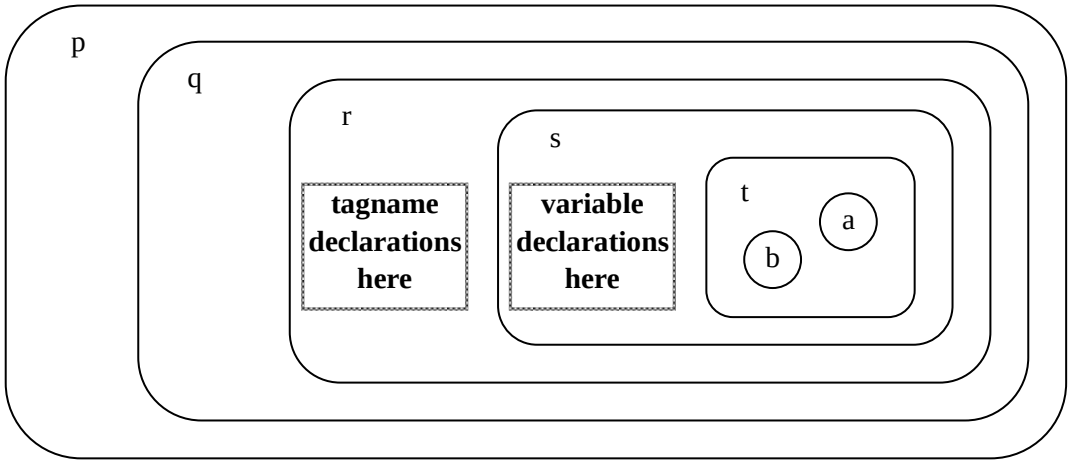


Figure 62. Tagname in shallower scope

This has no effect on the symbol table entries of the variables being declared. However, whenever a variable's *actual* type is accessed, the true path in which the tagname was declared will be used.

We now consider what happens when scoping operators are applied

- to the tagname
- to the variable name
- to terms on the right hand side of the initialization expression

Consider the following state machine

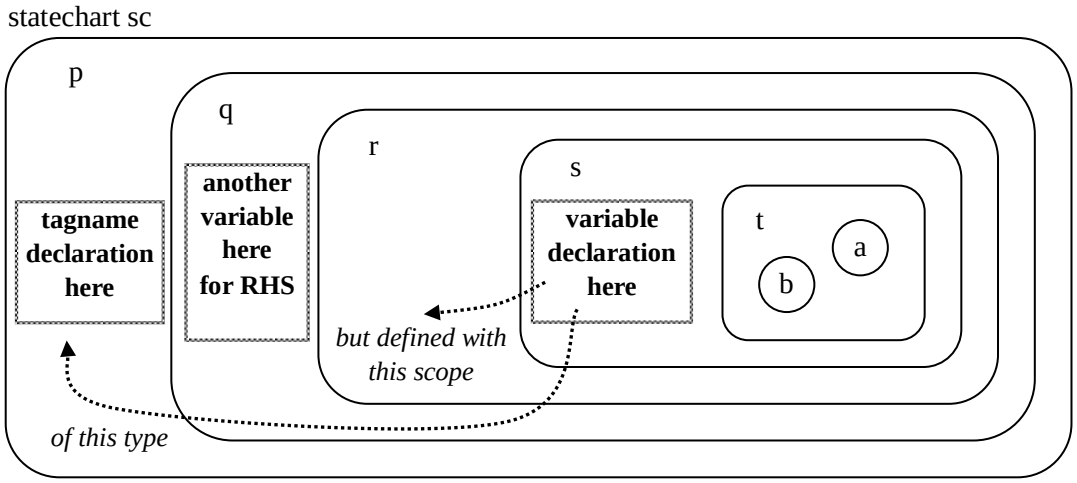


Figure 63. Scoping example for variables

In STATECRUNCHER this is represented by

```
// "Tie" example

statechart sc(p)
  PCO pco1;
  event alpha@pco1;
  cluster p(q)
     enum colour {red=1,green=3,blue=4};
     cluster q(r)
        colour colour_of_the_day=green;
        cluster r(s)
           cluster s(t)
                $$$colour $my_tie_col = $$colour_of_the_day;
                cluster t(a,b)
                  state a    ;
                  state b    ;
```

The statement under consideration (in machine path context **`sc.p.q.r.s`**) is

```
$$$colour  $my_tie_col = $$colour_of_the_day;
```

We examine the constituent parts.

`$my_tie_col:` This statement defines the variable `my_tie_col` as if it had been defined directly in machine path **`sc.p.q.r`**

`$$$colour:` The **scoping expression** on the *tagname* affects the **type** of the *variable*. In this case, the variable `my_tie_col` is defined to be of type


```
[vardecl,[enumtype,[colour,[sc.p]]]].
```

Note the shortened machine path within the type. This is the way the symbol table entry is made, irrespective of whether `colour` is to be found at this level or a more outer level.

`$$colour_of_the_day:` Variables (and enumerators) on the right hand side have their machine path modified with respect to the statement machine path (not the tagname or left-hand-side-variable machine path). So the scope in which `colour_of_the_day` is first sought will be **`sc.p.q`**. As usual when it comes to looking up a symbol within some scope, an outward search takes place to find the symbol as necessary. In this case the symbol is found in this first-attempt scope, **`sc.p.q.`** If that were not the case, it would also be sought in scope **`sc.p`** and then in scope **`sc`**.

Symbol table entry made as the result of the example statement:

name	machine path	type	value
`my_tie_col`	`sc.p.q.r`	`[vardecl,[enumtype,[colour,[sc.p]]]]`	*e.g.* `3`

Table 37. Symbol table entries for scoped variables

4.5.3 PCO declaration statements

Syntax of a PCO declaration

PCO *pcoexprssion, pcoexpression,...;*

Example, in the machine path context of **`sc.p.q.r.s`**

```
PCO pco1,$pco2;
```

This will declare a symbol `pco1` in context **`sc.p.q.r.s`** and a symbol `pco2` in context **`sc.p.q.r`**

name	machine path	type	value
`pco1`	`sc.p.q.r.s`	`pcodecls`	-
`pco2`	`sc.p.q.r`	`pcodecls`	-

Table 38. Symbol table entries for PCO declarations

4.5.4 Event declaration statements and PCOs in use

Syntax

event eventexpression,eventexpression,...[@pcoexpression];

Example, in the machine path context of **sc.p.q.r.s**

```
event omega
event alpha,$beta@$$pco1
```

This will declare an event `alpha` in machine path context `sc.p.q.r.s`, and an event `beta` in machine path context `sc.p.q.r`. The PCO expression `$$pco1` is evaluated with respect to the statement machine path. Both of these events will be related to the PCO expression by the event **value** field in the symbol table entry.

name	machine path	type	value
omega	sc.p.q.r.s	pcodecls	[]
alpha	sc.p.q.r.s	pcodecls	[pco1,[sc.p.q]
beta	sc.p.q.r	pcodecls	[pco1,[sc.p.q]

Table 39. Symbol table entries for event declarations

4.5.5 Events in use

Events are used in transitions, which are part of a state statement (cluster set or leaf-state). The transition description does not add any symbols to the symbol table. So determining the reference to a specified event in use is simply a matter of evaluating the scope of an *eventexpression* and searching for it in the symbol table from the evaluated scope outwards.

Example

```
alpha,$beta->d
```

The events specified are:

`alpha` machine path context `sc.p.q.r.s`

`beta` in machine path context `sc.p.q.r`

If these events have only been defined at a higher level, then that level will be taken.

4.5.6 State declaration statements

The scope of a state is determined by its hierarchical position in the complete ECHSM machine description. The current version of STATECRUNCHER does not provide any means to override this.

Symbols are entered in the symbol table with machine paths representing states above but not including the current state.

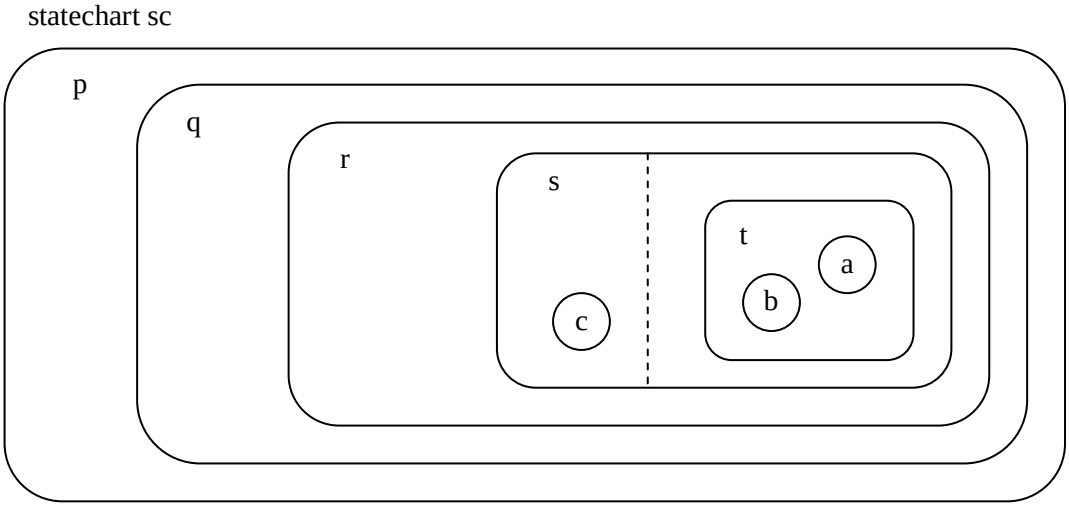


Figure 64. States declarations

name	machine path	type	value
sc	[]	statechart	-
p	[sc]	cluster	-
q	[sc.p]	cluster	-
r	[sc.p.q]	cluster	-
s	[sc.p.q.r]	set	-
t	[sc.p.q.r.s]	cluster	-
a	[sc.p.q.r.s.t]	leafstate	-
b	[sc.p.q.r.s.t]	leafstate	-
c	[sc.p.q.r.s]	leafstate	-

Table 40. Symbol table entries for state declarations

4.5.7 States in use

States are specified as scoped expressions, but with an exceptional aspect. The machine path used to evaluate a *stateexpression* is first *shortened* by one level. This rule gives a natural representation of transition target states. The shortening of the machine path can be interpreted as "moving out one level from the nested machine hierarchy". This is the state from which it is natural to view target states.

The split operator

This operator is used to define multiple target states of transitions. STATECRUNCHER allows transitions to specify targets in more than one member of a set. This can take place at various hierarchical levels, so requiring a *target state tree*. This is illustrated in the figure below.

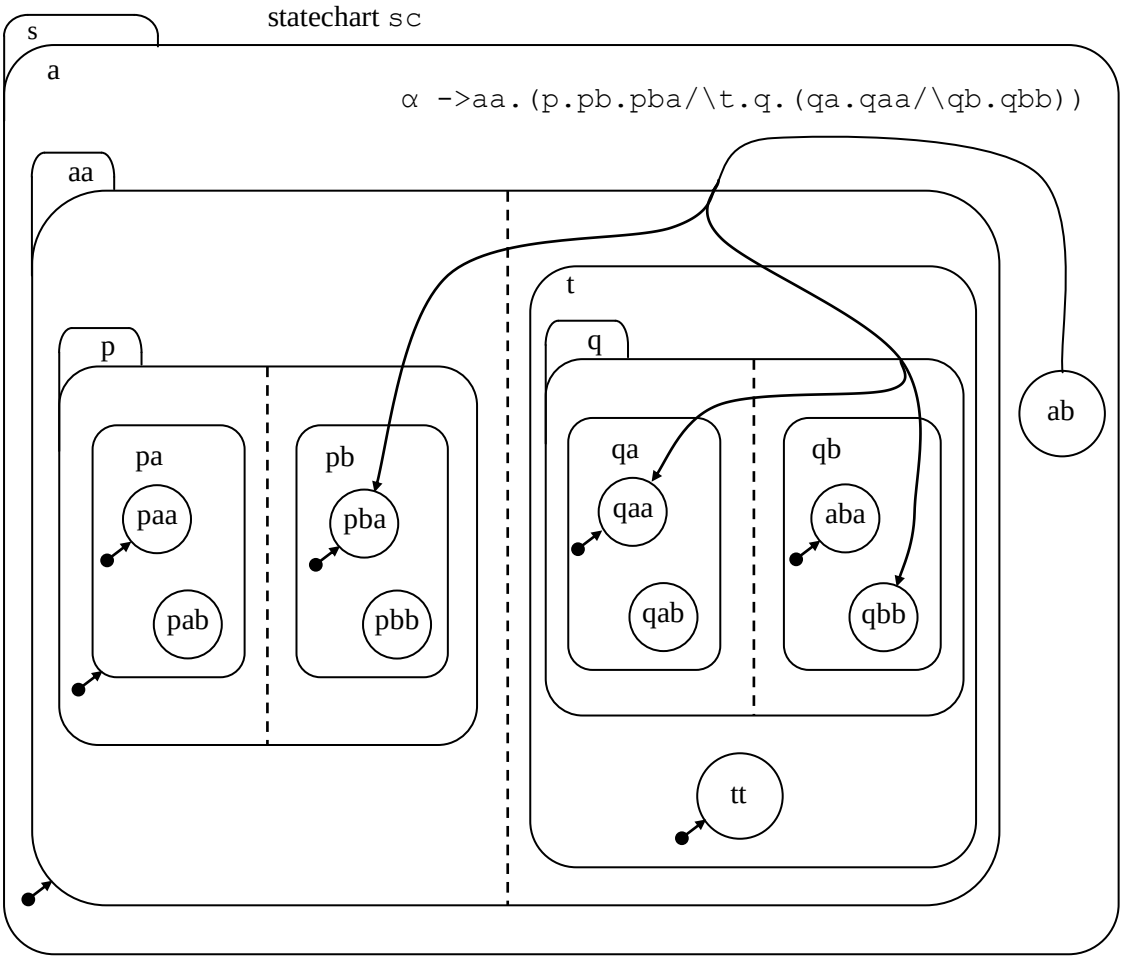


Figure 65. Multiple target states

Note that the target state tree need not specify all targets in a set – defaults (or historical states) will be taken where no specific target is specified.

The target state tree is specified using an operator to denote "and co-member", represented above by the symbol `/\`. The operator is available to target state expressions but is not available in other state expressions.

The operator is specified (in the same notation as used for scoping operators) as follows:

Operator syntax	Operator definition `[op,`*`precedence`*`,`*`shape`*`,`*`name`*`]`	Input argument 1	Input argument 2	Action
`arg1``/\`*`arg2`*	`[op,14,[y,f,x],split]`	state-expr	state-expr	the two operands, after evaluation, define two state-expressions, to be returned in a list.

Table 41. Split operator

This gives a lower binding precedence than the scoping operators (`::` `%%` `$` `.`). Either associativity could have been chosen. The term `[y,f,x]` denotes a LEFT associative operator, (such as the + operator), so that

```
a /\ b /\ c /\ d = ((a/\b)/\c)/\d.
```

Restriction

Note: The left hand side of the `"."` and "`%%`" operators should not be a term which has already been split, (although such a thing does make sense), since such a construction is unusual and the evaluator does not currently support it. So, in the figure below, it would not be permissible to write

```
α->a.((aa/\ab).x)
```

Instead, the following should be used:

```
α->a.(aa.x/\ab.x)
```

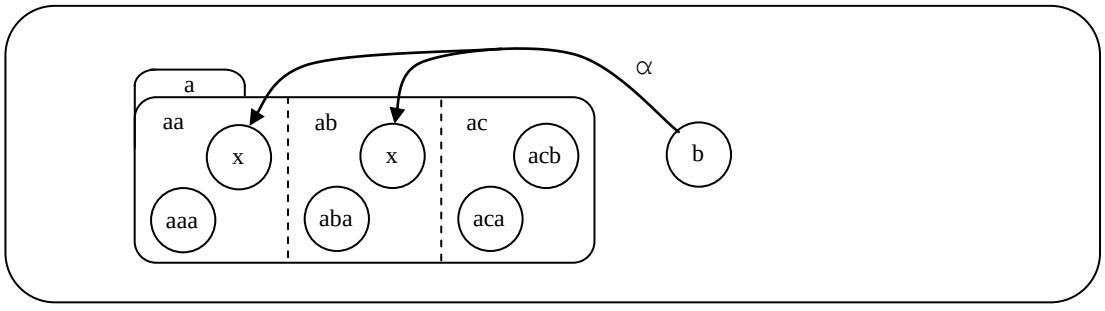


Figure 66. Restriction in use of the *split* operator

Evaluation of the split operator

The *evaluator* for terms combined with this operator produces a list of `[STATE|SPATH]` lists representing the target tree. Expressions are evaluated in an *evaluation scope* representing a state in hierarchy. Scopes are represented as a list, with the most local part of the hierarchy at the head of the list (i.e. on the left). Typical evaluations are as follows:

	Evaluation Scope	**Expression**	**Evaluation**
1	`[bb,aa]`	`dd/\ee`	`[[dd,bb,aa],` `[ee,bb,aa]]`
2	`[bb,aa]`	`pp.dd/\ee`	`[[dd,pp,bb,aa],` `[ee,bb,aa]]`
3	`[bb,aa]`	`(pp.dd)/\ee`	`[[dd,pp,bb,aa],` `[ee,bb,aa]]`
4	`[bb,aa]`	`pp.(dd/\ee)`	`[[dd,pp,bb,aa],` `[ee,pp,bb,aa]]`
5	`[bb,aa]`	`pp.(dd/\(ee/\ff.gg))`	`[[dd,pp,bb,aa],` `[ee,pp,bb,aa],` `[gg,ff,pp,bb,aa]]`
6	`[bb,aa]`	`pp.((dd/\ee)/\ff.gg)`	`[[dd,pp,bb,aa],` `[ee,pp,bb,aa],` `[gg,ff,pp,bb,aa]]`
7	`[bb,aa]`	`pp.((dd/\$ee)/\ff.gg)`	`[[dd,pp,bb,aa],` `[ee,bb,aa],` `[gg,ff,pp,bb,aa]]`
8	`[bb,aa]`	`pp.((dd/\ee)/\(ff.f2/\gg.hh))`	`[[dd,pp,bb,aa],` `[ee,pp,bb,aa],` `[f2,ff,pp,bb,aa],` `[hh,gg,pp,bb,aa]]`
9	`[bb,aa]`	`pp.(dd/\ee)/\(ff.f2/\gg.hh)`	`[[dd,pp,bb,aa],` `[ee,pp,bb,aa],` `[f2,ff,bb,aa],` `[hh,gg,bb,aa]]`
10	`[cc,bb,aa]`	`$$pp.(dd.ee.ff/\$gg.hh.ii)`	`[[ff,ee,dd,pp,aa],` `[ii,hh,gg,aa]]`
11	`[cc,bb,aa,sc]`	`::pp.(dd.ee.ff/\$gg.hh.ii)`	`[[ff,ee,dd,pp,sc],` `[ii,hh,gg,sc]]`
12	`[cc,bb,aa,sc]`	`::$pp/\$$dd`	`[[pp],` `[dd,aa,sc]]`
13	`[cc,bb,x1,x2,` `aa,sc]`	`aa%%pp/\$$dd`	`[[pp,aa,sc],` `[ddx1,x2,aa,sc]]`
14	`[cc,bb,aa]`	`(pp/\qq).rr` `// violates the restriction mentioned above.`	`unknown`

Table 42. Evaluations of the *split* operator

The target of transition α in Figure 65 is represented by

```
aa.(p.pb.pba/\t.q.(qa.qaa/\qb.qbb))
```

in evaluation scope

```
[a,s,sc]
```

evaluating to


```
[[pba,pb,p,aa,a,s,sc],
 [qaa,qa,q,t,aa,a,s,sc],
 [qbb,qb,q,t,aa,a,s,sc]]
```

4.6 Type compatibility in expressions

A rigorously typed language would require exact type matching of terms in expressions, and in left and right hand sides of assignments. It is felt that in STATECRUNCHER more freedom should be allowed: certainly a range-type variable should be compatible with integers.

Note that there is a type incompatibility if two types have the same name but due to scoping considerations they refer to type definitions at different scoping levels.

Example

```
$$$colour $$mycolour = $yourcolour;
```

There are two references to type a type definition named `colour`.
1: the one found by an outward search starting from `$$$`<current machine path>
2: the one found by an outward search starting from `$`<current machine path>

If these yield the same definition, the expression is type compatible, otherwise it is not.

Note:
In the current version of STATECRUNCHER, integers are compatible with *all* enum types.

4.6.1 Type compatibility in functions

STATECRUNCHER supports functions according to the GP4 implementation paradigm. This is an addition to the baseline document [ECHSM]. For simplicity in an initial version of STATECRUNCHER, functions are typeless. All functions accept any type in their parameters and the return parameter will match any type. This means that an identity function could act as a cast.

If rigorous typing is ever needed, function signatures could be provided:

- as an expedient, in Prolog directly (whereby the user follows a prescibed paradigm)
- as function declaration statements added to STATECRUNCHER syntax.

5. The validator module

The validator performs static validation of a STATECRUNCHER model, and produces

- error messages (on screen and in the validation listing file).
- a symbol table
- a cross reference table

These tables are produced on screen and in the validation listing file, and also in Prolog-readable form in a validation data file. The symbol table will be used by the run-time State Machine Engine (which is still future work at the time of writing). It is possible, but not likely, that the cross reference table will be used in the State Machine Engine.

The validator also initializes all variables and produces

- initial variable data in a Prolog "database" in the validation data file.

5.1 Validation checks

Validation checks can result in an error message or a warning message.

The following actions are performed and checks made:

Phase/Action	**Check for error condition:**	**Severity**
INITIAL		
	No compiled files loaded	Error
	Multiple compiled files loaded	Warning
	Version of compiler & validator incompatible	Error
	There were compile-time errors	Error
SYMBOL TABLE CONSTRUCTION		
	Double definition of an entry	Error
VARIABLE INITIALIZATION		
	Range error on initializing	Error
CROSS REFERENCE TABLE CONSTRUCTION		
	Polyvalent symbol in non-overlapping scopes	Warning
	Polyvalent symbol in overlapping scopes	Error
	Type mismatch in assignment	Error
	Type mismatch in expression	Error
	Unreferenced symbol	Warning

Table 43. Validation checks

Many more checks would be needed for completeness; in the current version additional checks, including transition validity checks, have been consigned to run-time. Certain checks could be very algorithm-specific; only in a very stable product would they usefully be performed statically.

5.2 Symbol table construction

While the symbol table is being constructed, it is a list containing entries in the following format

```
[SYMBOL, MACHINEPATH, SYMBOLTYPE, VALUE]
```

The machine path is the path that is obtained after applying scoping operations on the symbol. The following tables show symbol types and values.

Symbol type	Applies to
fixed_constant	fixed constant (e.g. `true`, `false`)
[typedecl,range]	TAGNAME (range)
[typedecl,enumerated]	TAGNAME (enumerated)
[enumerator,TAG]	ENUMERATOR
[vardecl,[enumtype,[TAG,TAGPATH]]]	VARIABLE (range or enumerated)
[vardecl,[bool]]	VARIABLE (boolean)
eventdecl	event
pcodecls	pco
statechart	statechart
set	set
cluster	cluster
leafstate	leaf state

Table 44. Symbol types

The VALUE field if a symbol table entry is dependent on the symbol type:

Symbol type	Value field
fixed_constant	numerical value, GP4-wrapped, e.g. [ex_co, int,0]
[typedecl,range]	list of low,high, e.g. [20,32]
[typedecl,enumerated]	legal values in a list, e.g. [0,1,3]
[enumerator,TAG]	numerical value, GP4-wrapped, e.g. [ex_co, int,3]
[vardecl,[enumtype,[TAG,TAGPATH]]]	numerical value, GP4-wrapped, e.g. [ex_co, int,3] or unknown
[vardecl,[bool]]	numerical value, GP4-wrapped, e.g. [ex_co, int,1] or unknown
eventdecl	name of corresponding PCO (or empty list if none)
pcodecls	no value (entered as st_novalue)
statechart, set, cluster, leafstate	no value (entered as st_novalue)

Table 45. Symbol values

When the list of `[SYMBOL, MACHINEPATH, SYMBOLTYPE, VALUE]` entries is complete, it is sorted with the following sorting priority:

1. The type (so that like entries appear together). The order is:
   - fixed constants
   - type declarations
   - enumerators
   - variable declarations
   - PCO declarations
   - event declarations
   - statecharts
   - sets
   - clusters
   - leafstates

2. The ALPHABETICAL ORDER of the symbol within a type

3. If two symbols are identical, the LONGER (most local) path comes first

4. If two path lengths are equal, the PATH ELEMENTS are used to sort alphabetically

The symbol table is then Prolog-asserted for use by subsequent code, and also written to the validator data file. The sorted list of all entries is not kept in scope. Entries are asserted individually as predicates as follows:

```
st_entry(SYMBOL, MACHINEPATH, SYMBOLTYPE, VALUE).
```

The symbol table can be accessed via the predicate `va_get_st_entry`. This predicate performs the outward search on the machine path to find the symbol that is in scope. Its parameters are

P1: INPUT	EPATH	effective machine path of symbol as used (after applying scoping operators)
P2: INPUT	SYMBOL	as in the symbol table
P3: OUTPUT	DPATH	machine path as declared as in the symbol table
P4: either	TYPE	symbol type as in the symbol table
P5: OUTPUT	VALUE	as in the symbol table

5.3 Cross reference table construction

Cross reference entries are asserted and written to the validator data file as follows:

```
xr_entry(SYMDECL,USEDIN)
```

where SYMDECL defines a symbol uniquely, and USEDIN describes a statement in which the symbol is used.

```
SYMDECL= [DSYMBOL, DPATH, DTYPE]
```

DSYMBOL = identifier for symbol as declared
DPATH = the machine path of the symbol as declared
DTYPE = the type of the symbol

For fixed constants, e.g. true

```
SYMDECL= [SYMBOL,[],DTYPE]
```

```
USEDIN= [USYMBOL,UPATH,UTYPE]
```

USYMBOL= statement name where SYMBOL is used
UPATH= machine path of statement with SYMBOL in use
UTYPE= type of statement with SYMBOL in use

There will be a symbol table entry for USYMBOL/UPATH.

Notes

1. Each entry is tested for existence in the symbol table or as a fixed constant. If it is not found, an error message is given (undefined symbol).

2. If a symbol occurs more than once in the same statement, it is given just one cross-reference entry. This also applies if the symbol appears in more than one guise (e.g. x and $x, but referring to the same x).

5.4 Type checking

We make a distinction between major types and variable types.

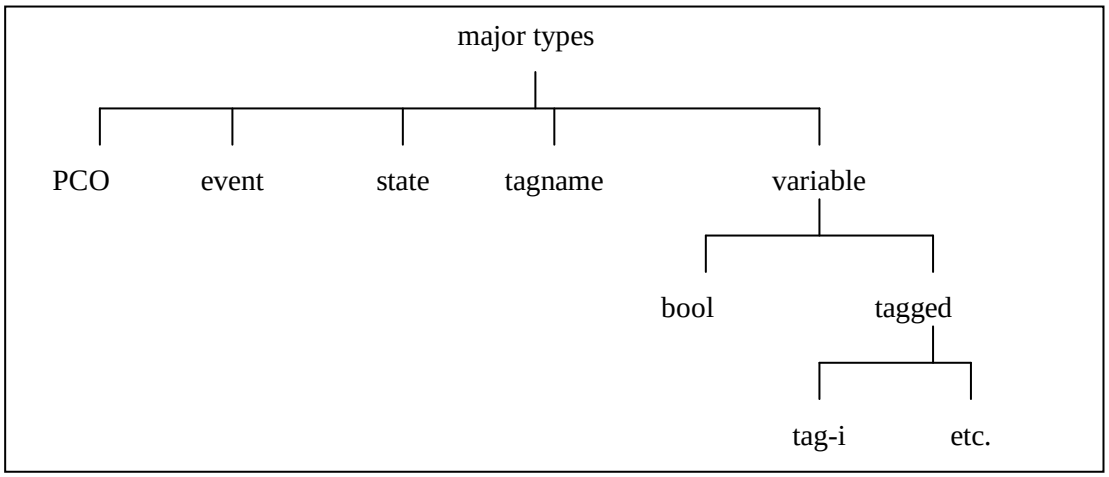


Figure 67. Types

Constants and enumerators are of the "variable" major type.

There are various kinds of type checking:

Undefined typed symbol

Whenever a symbol is encountered which must be of a certain major type, a check is made that there is a symbol of that major type in the symbol table. If that is not the case, an error message is given (irrespective of whether the symbol has not been defined at all, or has been defined as another major type).

Expression type checking

The issue here is compatibility of *variable types*. Expressions can be viewed as a parse tree with a top level operand; the types of operands are evaluated recursively under the following rules

- terminals return their own type as declared in the symbol table, (so after the outward search if they are not immediately in scope).
- operators examine the types of their operands to determine the return type. If the types of the operands are incompatible or unexpected, an error message is given. Examples
  - Scoping operators return the same type as their operand.
  - Arithmetic monadic operands return the same type as their operand.
  - Arithmetic dyadic operators expect their operands to be of the same type and they return that type.
- The function "`in`" returns the boolean type.

- Other functions accept any type and return a Prolog non-ground type, which will match any type. Functions currently have no prototypes and accept any type in their arguments.

The `va_expr_type` predicate that does the type checking returns type `[typerr,`*error-detail*`]` if the expression contains a typing error. This information is reproduced in the corresponding error message.

Assignment type checking
Here, the left hand side and right hand side of an assignment are checked for type compatibility.

5.5 Data store

Variables are stored in the data store as Prolog predicates of the signature

```
db_variable(WORLD,[SYMBOL,PATH],VALUE).
```

where the value is GP4-wrapped. Example

```
db_variable(WORLD,[b2,[p,s,my_st]],[ex_co,int,11]).
```

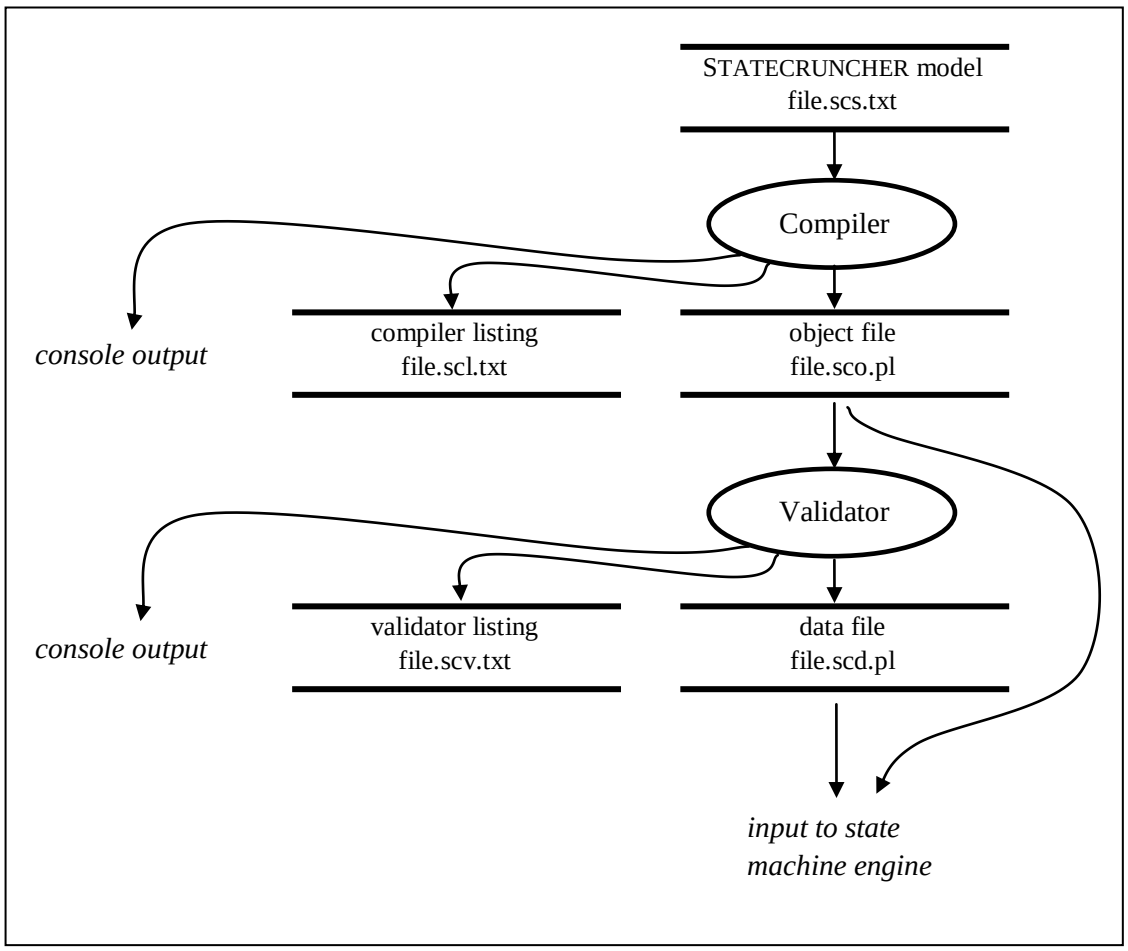
This format is not quite compatible with standard GP4, as that does not allow for a world or scoping component to the data. A consequence is that the STATECRUNCHER expression evaluator has some systematic variations with respect to the GP4 one.

As data changes, the relevant `db_variable` predicates are retracted, modified and re-asserted.

6. Compilation example

6.1 The STATECRUNCHER compilation & validation process

An overview is shown schematically in the figure below (as Figure 2):



6.2 Example model compiled

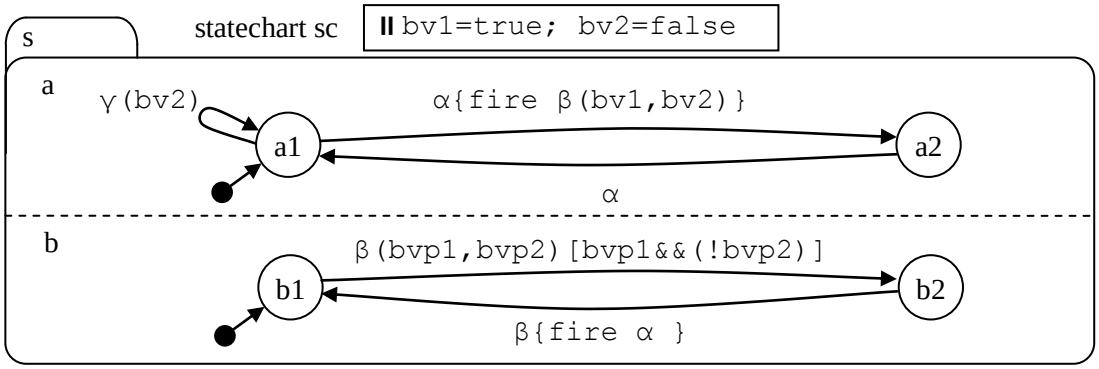


Figure 68. Fired event (deterministic) [model t5150]

Source code

```
statechart sc(s)
event alpha,beta,gamma;
bool bv1=true,bv2=false;
   set s(a,b)
      cluster a(a1,a2)
         state a1 {alpha->a2{fire beta(bv1,bv2);}; gamma(::bv2);  }
         state a2 {alpha->a1;}
      cluster b(b1,b2)
      bool b1.bvp1,b1.bvp2;
         state b1 {beta(bvp1,bvp2)[bvp1&&(!bvp2)]->b2;}
         state b2 {beta->b1 {fire alpha;};}
```

Object code of state a1

```
oc_state(
1,l_ok,[mpath,l_ok,[a1,a,s,sc]],[leafstate,a1,[opt2trblk,l_ok,[[enterblk,l_ok
,[]],[exitblk,l_ok,[]],[transitions,l_ok,[[transition,l_ok,[[metaevents,l_ok,
[[metaevent,l_ok,[t_event,l_ok,[[ex_evt_expr,[ex_id,alpha]],[tevt_paramblk,
l_ok,[]]]]]]],[condition,l_ok,[]],[route,l_ok,[[],[xstate,l_ok,[ex_sta_expr,[
ex_id,a2]]]]],[optactblk,l_ok,[[action,l_ok,[action_ev,l_ok,[[param_ev,l_ok,[
[ex_evt_expr,[ex_id,beta]],[evt_paramblk,l_ok,[[ex_expr,[ex_id,bv1]],[ex_expr
,[ex_id,bv2]]]]]]]]]]],[labelblk,l_ok,[]]]],[transition,l_ok,[[metaevents,
l_ok,[[metaevent,l_ok,[t_event,l_ok,[[ex_evt_expr,[ex_id,gamma]],[
tevt_paramblk,l_ok,[[ex_var_expr,[[ex_monadic,mscope],[ex_id,bv2]]]]]]]]]],[
condition,l_ok,[]],[route,l_ok,[[],[]]],[optactblk,l_ok,[]],[labelblk,l_ok,[]
]]]]]]]]).
```

Validator listing

```
+-------------------------------------------------------+
|  STATECRUNCHER VALIDATOR (Version 1.04)               |
|  Copyright (C) Philips Electronics N.V, 2000-2003     |
+-------------------------------------------------------+

+-------------------------------------------------------+
| SYMBOL AND CROSS REFERENCE TABLE                      |
+-------------------------------------------------------+
SYMB false      []          fixed_constant           [ex_co,int,0]

SYMB true       []          fixed_constant           [ex_co,int,1]

SYMB bv1        [sc]        [vardecl,[bool]]         [ex_co,int,1]
   XREF leafstate    a1:[a, s, sc]

SYMB bv2        [sc]        [vardecl,[bool]]         [ex_co,int,0]
   XREF leafstate    a1:[a, s, sc]

SYMB bvp1       [b1,b,s,sc] [vardecl,[bool]]         unknown
   XREF leafstate    b1:[b, s, sc]

SYMB bvp2       [b1,b,s,sc] [vardecl,[bool]]         unknown
   XREF leafstate    b1:[b, s, sc]

SYMB alpha      [sc]        eventdecl                []
   XREF leafstate    a1:[a, s, sc]
   XREF leafstate    a2:[a, s, sc]
   XREF leafstate    b2:[b, s, sc]

SYMB beta       [sc]        eventdecl                []
   XREF leafstate    a1:[a, s, sc]
   XREF leafstate    b1:[b, s, sc]
   XREF leafstate    b2:[b, s, sc]

SYMB gamma      [sc]        eventdecl                []
   XREF leafstate    a1:[a, s, sc]

SYMB sc         []          statechart

SYMB s          [sc]        set

SYMB a          [s,sc]      cluster

SYMB b          [s,sc]      cluster

SYMB a1         [a,s,sc]    leafstate
   XREF leafstate    a2:[a, s, sc]

SYMB a2         [a,s,sc]    leafstate
   XREF leafstate    a1:[a, s, sc]

SYMB b1         [b,s,sc]    leafstate
   XREF leafstate    b2:[b, s, sc]

SYMB b2         [b,s,sc]    leafstate
   XREF leafstate    b1:[b, s, sc]

+-------------------------------------------------------+
| END OF SYMBOL AND CROSS REFERENCE TABLE               |
+-------------------------------------------------------+

+-------------------------------------------------------+
|  STATECRUNCHER:  VALIDATION COMPLETE -    0 ERROR(S)  |
|                                           0 WARNING(S)|
+-------------------------------------------------------+
```


Validator-produced data (some Prolog directives excluded)

```
/*-------------------------------------------------------------*/
/*  STATECRUNCHER VALIDATOR-GENERATED DATA (Version 1.04)  */
/*  Copyright (C) Philips Electronics N.V, 2000-2003       */
/*-------------------------------------------------------------*/
vd_version('1.04').

/*---------enumerators---------*/

/*---------symbol table---------*/
st_entry(1,false,[],fixed_constant,[ex_co,int,0]).
st_entry(1,true,[],fixed_constant,[ex_co,int,1]).
st_entry(1,bv1,[sc],[vardecl,[bool]],st_see_database).
st_entry(1,bv2,[sc],[vardecl,[bool]],st_see_database).
st_entry(1,bvp1,[b1,b,s,sc],[vardecl,[bool]],st_see_database).
st_entry(1,bvp2,[b1,b,s,sc],[vardecl,[bool]],st_see_database).
st_entry(1,alpha,[sc],eventdecl,[]).
st_entry(1,beta,[sc],eventdecl,[]).
st_entry(1,gamma,[sc],eventdecl,[]).
st_entry(1,sc,[],statechart,st_novalue).
st_entry(1,s,[sc],set,st_novalue).
st_entry(1,a,[s,sc],cluster,st_novalue).
st_entry(1,b,[s,sc],cluster,st_novalue).
st_entry(1,a1,[a,s,sc],leafstate,st_novalue).
st_entry(1,a2,[a,s,sc],leafstate,st_novalue).
st_entry(1,b1,[b,s,sc],leafstate,st_novalue).
st_entry(1,b2,[b,s,sc],leafstate,st_novalue).

/*--------variables--------*/
db_variable(1,[bv1,[sc]],[ex_co,int,1]).
db_variable(1,[bv2,[sc]],[ex_co,int,0]).
db_variable(1,[bvp1,[b1,b,s,sc]],unknown).
db_variable(1,[bvp2,[b1,b,s,sc]],unknown).

/*---------cross reference table---------*/
xr_entry([true,[],int],[bv1,[sc],[vardecl,[bool]]]).
xr_entry([false,[],int],[bv2,[sc],[vardecl,[bool]]]).
xr_entry([bv1,[sc],[vardecl,[bool]]],[a1,[a,s,sc],leafstate]).
xr_entry([bv2,[sc],[vardecl,[bool]]],[a1,[a,s,sc],leafstate]).
xr_entry([bvp1,[b1,b,s,sc],[vardecl,[bool]]],[b1,[b,s,sc],leafstate]).
xr_entry([bvp2,[b1,b,s,sc],[vardecl,[bool]]],[b1,[b,s,sc],leafstate]).
xr_entry([alpha,[sc],eventdecl],[a1,[a,s,sc],leafstate]).
xr_entry([beta,[sc],eventdecl],[a1,[a,s,sc],leafstate]).
xr_entry([gamma,[sc],eventdecl],[a1,[a,s,sc],leafstate]).
xr_entry([alpha,[sc],eventdecl],[a2,[a,s,sc],leafstate]).
xr_entry([beta,[sc],eventdecl],[b1,[b,s,sc],leafstate]).
xr_entry([beta,[sc],eventdecl],[b2,[b,s,sc],leafstate]).
xr_entry([alpha,[sc],eventdecl],[b2,[b,s,sc],leafstate]).
xr_entry([a2,[a,s,sc],leafstate],[a1,[a,s,sc],leafstate]).
xr_entry([a1,[a,s,sc],leafstate],[a2,[a,s,sc],leafstate]).
xr_entry([b2,[b,s,sc],leafstate],[b1,[b,s,sc],leafstate]).
xr_entry([b1,[b,s,sc],leafstate],[b2,[b,s,sc],leafstate]).

vd_errorcount(0).
vd_warningcount(0).

/*---[End of file
F:\KWinPro\StCr\StCr2sand\..\StCr3ModelsTest\t5000me\t5150_determ_fire
\determ_fire.scd.pl ]---*/
```

7. Prolog grammar code example: STATECRUNCHER declarations

```
/*-------------------------------------------------------------------------*/
/* Module:    sy_sc_2.pl                                                   */
/* Author:    Graham Thomason, Philips Research Laboratories, Redhill      */
/* Date:      15 December, 1999                                            */
/* Purpose:   Statecruncher Syntax (2): Declarations                       */
/*                                                                         */
/* Project:   Testing Reusable Software Components (756)                   */
/*                                                                         */
/* Copyright (C) 1999 Philips Electronics N.V.                             */
/*-------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------*/
/* PRL PROTOTYPE Software                                                  */
/*                                                                         */
/*-------------------------------------------------------------------------*/


/*-------------------------------------------------------------------------*/
/*345678901234567890123456789012345678901234567890123456789012345678901234567*/
/*       1         2         3         4         5         6         7       */
/*-------------------------------------------------------------------------*/


/*-------------------------------------------------------------------------*/
/* DYNAMIC PREDICATES                                                      */
/*-------------------------------------------------------------------------*/


/*-------------------------------------------------------------------------*/
/* EXTERNALS                                                               */
/*-------------------------------------------------------------------------*/


/*-------------------------------------------------------------------------*/
/* ASSERTIONS AND RETRACTIONS                                              */
/*-------------------------------------------------------------------------*/


/*-------------------------------------------------------------------------*/
/* MULTIFILE DECLARATIONS                                                  */
/*-------------------------------------------------------------------------*/
:-multifile sy_present/1.


/*-------------------------------------------------------------------------*/
/* sy_present                                                              */
/* ==========                                                              */
/*   Predicate for the presence of this module,                            */
/*   Can be used to check completeness, but non-multipleness, of system load */
/*-------------------------------------------------------------------------*/
sy_present(sc_2).


/*-------------------------------------------------------------------------*/
```



```
/* Description:                                                               */
/*   Syntax of declarations                                                   */
/*--------------------------------------------------------------------------*/


/*##########################################################################*/
/*##########################################################################*/
/*##                                                                      ##*/
/*##  STATECHART - TOP LEVEL STATEMENT                                    ##*/
/*##                                                                      ##*/
/*##########################################################################*/
/*##########################################################################*/

/*--------------------------------------------------------------------------*/
/* sy_statechart                                                            */
/* =============                                                            */
/*                                                                          */
/*                                                                          */
/* SOURCE EXAMPLE                                                           */
/*   statechart newtv(xyz)                                                  */
/*   statechart #rubbish                                                    */
/*                                                                          */
/*                                                                          */
/* PARSE                                                                    */
/*   [statechart,l_ok,[STATECHART_NAME,STATE_NAME]]                         */
/*   [statechart,l_er,['**Error: statechart']]                              */
/*                                                                          */
/*--------------------------------------------------------------------------*/
sy_statechart(_,_)-->
   {sy_debug_log(sy_statechart), fail}.

sy_statechart(GSTATUS,[statechart,l_ok,[NAME,SNAMSBLK]])-->
   sy_keyword(g_ok,[_,_,statechart]),
   sy_identifier(g_ok,[_,_,NAME]),
   sy_statenames_block(GSTATUS,SNAMSBLK),
   {!}.

sy_statechart(g_er,[statechart,l_er,['**Error: statechart']])-->
   sy_keyword(g_ok,[_,_,statechart]),
   ex_any_text_long(TEXT),  /* illegal bit here */
   {!}.


/*##########################################################################*/
/*##########################################################################*/
/*##                                                                      ##*/
/*##  TYPE DECLARATION                                                    ##*/
/*##  ================                                                    ##*/
/*##                                                                      ##*/
/*##########################################################################*/
/*##########################################################################*/

/*--------------------------------------------------------------------------*/
/* sy_type_declaration                                                      */
/* ===================                                                      */
/*                                                                          */
/* SOURCE                                                                   */
/*                     enum TYPENAME { INTEGER .. INTEGER } ;               */
/*   or                enum colour   { red=9, green, blue=8, yellow }       */
/*   or <Error-parse>  enum ANYTEXT ;                                       */
/*                                                                          */
/* SCOPING EXTENSION: TYPENAME can be a TAG-EXPRESSION                      */
/*                                                                          */
/* PARSE                                                                    */
/*   [typedecl,l_ok,[ETYPENAME,ENUMBODY]]                                   */
/*                                                                          */
/*--------------------------------------------------------------------------*/
sy_type_declaration(_,_)-->
   {sy_debug_log(sy_type_declaration), fail}.
```

```
sy_type_declaration(GSTATUS,[typedecl,l_ok,[TAGEXPR,ENUMBODY]])-->
   sy_keyword(g_ok,[_,_,enum]),
   sy_expr(levels_all,sc_tag,EXPR),
   { EXPR=[ex_expr,E], TAGEXPR=[ex_tag_expr,E] },
   sy_enum_body(GSTATUS,ENUMBODY),
   sy_literal(';'),
   {!}.

/*-----------------------------------------------------------------*/
/* sy_type_declaration                                             */
/* ===================                                             */
/*                                                                 */
/*   enum **ERROR-HERE**                                           */
/*                                                                 */
/*-----------------------------------------------------------------*/
sy_type_declaration(g_er,[typedecl,l_er,PARSE])-->
   sy_keyword(g_ok,[_,_,enum]),
   ex_any_text_long(_),     /* illegal bit here */
   { PARSE=['**Error: type declaration: in enum'] },
   {!}.


/*#################################################################################*/
/*#                                                                               #*/
/*# ENUM BODY                                                                     #*/
/*# =========                                                                     #*/
/*#                                                                               #*/
/*#################################################################################*/

/*-------------------------------------------------------------------------------*/
/* sy_enum_body                                                                  */
/* ============                                                                  */
/*                                                                               */
/* An enum body is:                                                              */
/*   a  RANGE                  { integer .. integer }                            */
/*   or VALUE NAMES            { red=9, green, blue=8, yellow }                  */
/*   or an ERRONEOUS PARSE     { ANYTEXT }                                       */
/*-------------------------------------------------------------------------------*/
sy_enum_body(_,_)-->
   {sy_debug_log(sy_enum_body), fail}.

/*---------------------------------------*/
/* range type                            */
/*---------------------------------------*/
sy_enum_body(GSTATUS,[enumbody,l_ok,RANGE])-->
   sy_range(GSTATUS,RANGE),
   {!}.

/*---------------------------------------*/
/* value names type                      */
/*---------------------------------------*/
sy_enum_body(GSTATUS,[enumbody,l_ok,VALNAMS])-->
   sy_valuenames(GSTATUS,VALNAMS),
   {!}.

/*---------------------------------------*/
/* error type                            */
/*---------------------------------------*/
sy_enum_body(g_er,[enumbody,l_er,['**Error: enum body: between braces']])-->
   sy_bracepairs('{}',TEXT),   /* braces and illegal text picked up here */
   {!}.


/*#################################################################################*/
/*#                                                                               #*/
/*# RANGE                                                                         #*/
/*# =====                                                                         #*/
/*#                                                                               #*/
/*#################################################################################*/
```



```
/*-------------------------------------------------------------------------*/
/* sy_range                                                                */
/* ========                                                                */
/*                                                                         */
/* SOURCE                                                                  */
/*  { integer,..,integer }          second integer >= first                */
/*                                                                         */
/* PARSE                                                                   */
/*   [range,[2,12]]                                                        */
/*   [range,[sy_err|ERROR_DETAIL]]                                         */
/*                                                                         */
/* NOTE:                                                                   */
/*   We do not have an erroneous range handler at this level because       */
/*   there are other {...} production rules in a sister relationship to this */
/*                                                                         */
/*   We do, however, want to process the braces { } as they are a nice     */
/*   handle on this item, in particular for repeating items, such as       */
/*   occur in a sister predicate, for value names                          */
/*-------------------------------------------------------------------------*/
sy_range(_,_)-->
   {sy_debug_log(sy_range), fail}.

sy_range(STATUS,[range,LSTATUS,PARSE])-->
   sy_literal('{'),
   sy_int(VAL1),
   sy_literal(','),
   sy_literal('..'),
   sy_literal(','),
   sy_int(VAL2),
   sy_literal('}'),
   {
      ( (
           VAL2 >= VAL1,
           STATUS=g_ok,
           LSTATUS=l_ok,
           PARSE=[VAL1,VAL2]
      );(
           VAL2 <  VAL1,
           STATUS=g_er,
           LSTATUS=l_er,
           PARSE=[VAL1,VAL2,'**Error: range: lower .. higher']
      ) )
   },
   {!}.


/*#########################################################################*/
/*#                                                                       #*/
/*# VALUE NAMES                                                           #*/
/*# ===========                                                           #*/
/*#                                                                       #*/
/*#########################################################################*/

/*-------------------------------------------------------------------------*/
/* sy_valuenames                                                           */
/* =============                                                           */
/*                                                                         */
/* General approach                                                        */
/* ================                                                        */
/* SOURCE e.g.                                                             */
/*     { red=9, green, blue=8, yellow }                                    */
/*                                                                         */
/*      no requirement to be non-overlapping                               */
/*      no req to be ascending order                                       */
/*      FCHSM's do not allow negative numbers (or constant expressions)    */
/*                                                                         */
/* PARSE                                                                   */
/*   [valnams,[ [NAME,VALUE],[NAME,VALUE]...] ]]                           */
/*   [valnams,[sy_err|ERROR_DETAIL]]                                       */
/*                                                                         */
```

```
/* NOTE:                                                                    */
/*   We do not have an erroneous handler at this level -                    */
/*      see comments on "RANGE"                                             */
/*                                                                          */
/* syntax as seen at first level of parse productions                       */
/* =================================================                        */
/* SOURCE                                                                   */
/*   { ENTRY RESTVALUENAMES }                                               */
/*                                                                          */
/*--------------------------------------------------------------------------*/
sy_valuenames(_,_)-->
   {sy_debug_log(sy_valuenames), fail}.

sy_valuenames(GSTATUS,[valnams,l_ok,[HEAD|TAIL]])-->
   sy_literal('{'),
   sy_valname_entry(GSTATUS1,[_,_,HEAD]),
   sy_restvaluenames(GSTATUS2,[_,_,TAIL]),
   sy_literal('}'),
   { sy_combine_statuses([GSTATUS1,GSTATUS2],GSTATUS) },
   {!}.

/*--------------------------------------------------------------------------*/
/* sy_restvaluenames                                                        */
/* =================                                                        */
/*                                                                          */
/* SOURCE                                                                   */
/*   ,ENTRY,ENTRY...  (or null)                                             */
/*                                                                          */
/* PARSE                                                                    */
/*   [restvalnams,LSTATUS,[ENTRY,ENTRY,ENTRY,...]                           */
/*--------------------------------------------------------------------------*/
sy_restvaluenames(GSTATUS,[restvalnams,l_ok,[HEAD|TAIL]])-->
   sy_literal(','),
   sy_valname_entry(GSTATUS1,[_,_,HEAD]),
   sy_restvaluenames(GSTATUS2,[_,_,TAIL]),
   { sy_combine_statuses([GSTATUS1,GSTATUS2],GSTATUS) },
   {!}.

sy_restvaluenames(g_ok,[restvalnams,l_ok,[]])-->
      ex_opt_delim(_),
   {!}.


/*--------------------------------------------------------------------------*/
/* sy_valnam_entry                                                          */
/* ===============                                                          */
/*                                                                          */
/* SOURCE                                                                   */
/*   xyz                                                                    */
/*   xyz=6                                                                  */
/*                                                                          */
/* PARSE                                                                    */
/*   [valnament,LSTATUS,[xyz,6]]                                            */
/*--------------------------------------------------------------------------*/
sy_valuename_entry(_,_)-->
   {sy_debug_log(sy_valuename_entry), fail}.

sy_valname_entry(g_ok,[valnament,l_ok,[ENTRY,VAL]])-->
   sy_identifier(g_ok,[_,_,ENTRY]),
   sy_literal('='),
   sy_int(VAL),
   {!}.

sy_valname_entry(g_ok,[valnament,l_ok,[ENTRY]])-->
   sy_identifier(g_ok,[_,_,ENTRY]),
   {!}.



/*##########################################################################*/
```



```
/*############################################################################*/
/*##                                                                        ##*/
/*##  VARIABLE DECLARATIONS                                                 ##*/
/*##  =====================                                                 ##*/
/*##                                                                        ##*/
/*############################################################################*/
/*############################################################################*/


/*----------------------------------------------------------------------------*/
/* sy_var_declaration                                                         */
/* ==================                                                         */
/*                                                                            */
/* SOURCE                                                                     */
/*   xyztype xyz1=6+x,xyz2,xyz3=7+y;                                          */
/*   bool flag1;                                                              */
/*                                                                            */
/* SCOPING EXTENSION                                                          */
/*   applies to type and variable                                             */
/*                                                                            */
/* PARSE                                                                      */
/*   [vardecl,l_ok,[VARDECLTYPE,VARDECLENTRY,VARDECLENTRY,...]]               */
/*----------------------------------------------------------------------------*/
sy_var_declaration(_,_)-->
   {sy_debug_log(sy_var_declaration), fail}.

sy_var_declaration(GSTATUS,[vardecl,l_ok,[VDT,VDE|VDR]])-->
   sy_var_decl_type(GSTATUS1,VDT),
   sy_var_decl_entry(GSTATUS2,VDE),
   sy_rest_var_decl(GSTATUS3,[_,_,VDR]),
   sy_literal(';'),
   { sy_combine_statuses([GSTATUS1,GSTATUS2,GSTATUS3],GSTATUS) },
   {!}.

sy_var_declaration(g_er,[vardecl,l_er,PARSE])-->
   sy_var_decl_type(GSTATUS1,VDT),
   ex_any_text_long(_),  /* illegal bit here */
   { PARSE=['**Error: var declaration'] },
   {!}.



/*----------------------------------------------------------------------------*/
/* sy_var_decl_type                                                           */
/* ================                                                           */
/*                                                                            */
/* SOURCE                                                                     */
/*   bool                                                                     */
/*   ENUMTYPE                                                                 */
/*                                                                            */
/* EXTENSION FOR SCOPING EXPRESSIONS                                          */
/*                                                                            */
/* PARSE                                                                      */
/*   [vardecltype,LSTATUS,[bool]]                                             */
/*   [vardecltype,LSTATUS,[enumtype,TYPE]]                                    */
/*----------------------------------------------------------------------------*/
sy_var_decl_type(_,_)-->
   {sy_debug_log(sy_var_decl_type), fail}.

sy_var_decl_type(g_ok,[vardecltype,l_ok,[bool]])-->
   sy_keyword(g_ok,[_,_,bool]),
   {!}.

sy_var_decl_type(_,_)-->            /* exclude anything starting with a keyword */
   sy_keyword(g_ok,[_,_,KEYWORD]),
   {!,fail}.

sy_var_decl_type(g_ok,[vardecltype,l_ok,[enumtype,TAGEXPR]])-->
   /**sy_identifier(g_ok,[_,_,TYPE]),**/
   sy_expr(levels_all,sc_tag,EXPR),
```

```
   { EXPR=[ex_expr,E], TAGEXPR=[ex_tag_expr,E] },
   {!}.


/*-------------------------------------------------------------------------*/
/* sy_var_decl_entry                                                       */
/* =================                                                       */
/*                                                                         */
/* SOURCE                                                                  */
/*   xyz                                                                   */
/*   xyz=6+x                                                               */
/*                                                                         */
/* EXTENSIONS TO ALLOW FOR SCOPING OPERATORS                               */
/*                                                                         */
/* PARSE                                                                   */
/*   [vardeclent,LSTATUS,[xyz,EXPR]]                                       */
/*-------------------------------------------------------------------------*/
sy_var_decl_entry(_,_)-->
   {sy_debug_log(sy_var_decl_entry), fail}.

sy_var_decl_entry(g_ok,[vardeclent,l_ok,[VAREXPR,RHSEXPR]])-->
   sy_expr(levels_all,sc_var,EXPR),
   { EXPR=[ex_expr,E], VAREXPR=[ex_var_expr,E] },
   sy_literal('='),
   sy_expr(levels_noasg,all,RHSEXPR),
   {!}.

sy_var_decl_entry(g_ok,[vardeclent,l_ok,[VAREXPR]])-->
   /* Note: deep nesting of VAREXPR for compatibility with above */
   sy_expr(levels_all,sc_var,EXPR),
   { EXPR=[ex_expr,E], VAREXPR=[ex_var_expr,E] },
   {!}.


/*-------------------------------------------------------------------------*/
/* sy_rest_var_decl                                                        */
/* ================                                                        */
/*                                                                         */
/* SOURCE                                                                  */
/*   ,pqr=7                                                                */
/*   <null>                                                                */
/*                                                                         */
/* PARSE                                                                   */
/*   [restvardecl,LSTATUS,[pqr,EXPR]]                                      */
/*   [restvardecl,LSTATUS,[]]                                              */
/*-------------------------------------------------------------------------*/
sy_rest_var_decl(GSTATUS,[restvardecl,l_ok,[HEAD|TAIL]])-->
   sy_literal(','),
   sy_var_decl_entry(GSTATUS1,HEAD),
   sy_rest_var_decl(GSTATUS2,[_,_,TAIL]),
   { sy_combine_statuses([GSTATUS1,GSTATUS2],GSTATUS) },
   {!}.

sy_rest_var_decl(g_ok,[restvardecl,l_ok,[]])-->
   ex_opt_delim(_),
   {!}.



/*#########################################################################*/
/*#########################################################################*/
/*##                                                                     ##*/
/*##  PCO DECLARATIONS                                                   ##*/
/*##  ================                                                   ##*/
/*##                                                                     ##*/
/*#########################################################################*/
/*#########################################################################*/

/*-------------------------------------------------------------------------*/
/* sy_pco_declarations                                                     */
```



```
/* ==================                                                    */
/*                                                                       */
/* SCOPING OPERTORS NOW INCLUDED                                         */
/*                                                                       */
/* SOURCE                                                                */
/*   PCO pco1 , pco2 ;            (can also be scoping expressions)      */
/*   PCO <erroneous text> ;                                              */
/*                                                                       */
/* PARSE                                                                 */
/*   [pcodecls,LSTATUS,[PCODECL,PCODECL,...]]                            */
/*                                                                       */
/*-----------------------------------------------------------------------*/
sy_pco_declarations(_,_)-->
   {sy_debug_log(sy_pco_declarations), fail}.

sy_pco_declarations(GSTATUS,[pcodecls,l_ok,[PCOEXPR|PCOREST]])-->
   sy_keyword(g_ok,[_,_,'PCO']),
   sy_expr(levels_all,sc_pco,EXPR),
   { EXPR=[ex_expr,E], PCOEXPR=[ex_pco_expr,E]},
   sy_pco_restnames(GSTATUS,[_,_,PCOREST]),
   sy_literal(';'),
   {!}.

sy_pco_declarations(g_er,[pcodecls,l_er,PARSE])-->
   sy_keyword(g_ok,[_,_,'PCO']),
   ex_any_text_short(TEXT),  /* illegal bit here */
   sy_literal(';'),
   { PARSE=['**Error: pco declarations'] },
   {!}.


/*-----------------------------------------------------------------------*/
/* sy_pco_restnames                                                      */
/* ================                                                      */
/*                                                                       */
/* SOURCE EXAMPLES                                                       */
/*   ,pco1,pco2                                                          */
/*   <null>                                                              */
/*                                                                       */
/* PARSE                                                                 */
/*   [pcorestnames,LSTATUS,[PCODECL,PCODECL,...]]                        */
/*   [pcorestnames,LSTATUS,[]]                                           */
/*-----------------------------------------------------------------------*/
sy_pco_restnames(GSTATUS,[pcorestnames,l_ok,[PCOEXPR|TAIL]])-->
   sy_literal(','),

   sy_expr(levels_all,sc_pco,EXPR),
   { EXPR=[ex_expr,E], PCOEXPR=[ex_pco_expr,E]},

   sy_pco_restnames(GSTATUS,[_,_,TAIL]),
   {!}.

sy_pco_restnames(g_ok,[pcorestnames,l_ok,[]])-->
   ex_opt_delim(_),
   {!}.



/*#######################################################################*/
/*#######################################################################*/
/*##                                                                   ##*/
/*##  EVENT DECLARATIONS                                               ##*/
/*##  ==================                                               ##*/
/*##                                                                   ##*/
/*#######################################################################*/
/*#######################################################################*/


/*-----------------------------------------------------------------------*/
/* sy_event_decl                                                         */
```

```
/* =============                                                              */
/*                                                                            */
/* SOURCE                                                                     */
/*   event event1 , event2 @ pco1 ;                                           */
/*                                                                            */
/*         the event can be a scoping expression                              */
/*                                                                            */
/* PARSE                                                                      */
/*   [eventdecl,LSTATUS,[EVENTNAMES,PCO]]                                     */
/*----------------------------------------------------------------------------*/
sy_event_decl(_,_)-->
   {sy_debug_log(sy_event_decl), fail}.

sy_event_decl(GSTATUS,[eventdecl,l_ok,[EVENTNAMES,PCO]])-->
   sy_keyword(g_ok,[_,_,event]),
   sy_event_names(GSTATUS1,[_,_,EVENTNAMES]),
   sy_opt_pco(GSTATUS2,[_,_,PCO]),
   sy_literal(';'),
   { sy_combine_statuses([GSTATUS1,GSTATUS2],GSTATUS) },
   {!}.


/*------------*/
/* error trap */
/*------------*/
sy_event_decl(g_er,[eventdecl,l_er,PARSE])-->
   sy_keyword(g_ok,[_,_,event]),
   ex_any_text_long(_),          /* illegal bit here */
   sy_literal(';'),
   { PARSE=['**Error: event declaration'] },
   {!}.


/*----------------------------------------------------------------------------*/
/* sy_event_names                                                             */
/* ==============                                                             */
/*                                                                            */
/* SOURCE                                                                     */
/*   event1 , event2                                                          */
/*                                                                            */
/* PARSE                                                                      */
/*   [eventnames,LSTATUS,[EVENTNAME,EVENTNAME,...]]                           */
/*----------------------------------------------------------------------------*/
sy_event_names(_,_)-->
   {sy_debug_log(sy_event_names), fail}.

sy_event_names(GSTATUS,[eventnames,l_ok,[EVENTEXPR|EVENTREST]])-->
   sy_expr(levels_all,sc_event,EXPR),
   { EXPR=[ex_expr,E],EVENTEXPR=[ex_evt_expr,E]},
   sy_event_restnames(GSTATUS,[_,_,EVENTREST]),
   {!}.


/*----------------------------------------------------------------------------*/
/* sy_event_restnames                                                         */
/* ================                                                           */
/*                                                                            */
/* SOURCE EXAMPLES                                                            */
/*   ,event1,event2                                                           */
/*   <null>                                                                   */
/*                                                                            */
/* PARSE                                                                      */
/*   [eventrestnames,LSTATUS,[EVENTNAME,EVENTNAME,...]]                       */
/*   [eventrestnames,LSTATUS,[]]                                              */
/*----------------------------------------------------------------------------*/
sy_event_restnames(GSTATUS,[eventrestnames,l_ok,EVENTNAMES])-->
   sy_literal(','),
   sy_event_names(GSTATUS,[_,_,EVENTNAMES]),
   {!}.
```



```
sy_event_restnames(g_ok,[eventrestnames,l_ok,[]])-->
   ex_opt_delim(_),
   {!}.


/*-------------------------------------------------------------------------*/
/* sy_opt_pco                                                              */
/* ==========                                                              */
/*                                                                         */
/* EXTENSION TO ORIGINAL SYNTAX SINCE INTRODUCTION OF SCOPING EXPRESSIONS  */
/*                                                                         */
/* SOURCE                                                                  */
/*   @ pco1                                                                */
/*   @::m1.pco2    @...pco3                                                */
/*   <null>                                                                */
/*                                                                         */
/* PARSE                                                                   */
/*   [opt_pco,LSTATUS,[PCOEXPR]]                                           */
/*   [opt_pco,LSTATUS,[]]                                                  */
/*                                                                         */
/*-------------------------------------------------------------------------*/
/* ORIGINAL SYNTAX WITHOUT SCOPING EXPRESSIONS                             */
/*                                                                         */
/* SOURCE                                                                  */
/*   @ pco1                                                                */
/*   <null>                                                                */
/*                                                                         */
/* PARSE                                                                   */
/*   [opt_pco,LSTATUS,[PCONAME]]           (name only)                     */
/*   [opt_pco,LSTATUS,[]]                                                  */
/*-------------------------------------------------------------------------*/
sy_opt_pcol(_,_)-->
   {sy_debug_log(sy_opt_pco), fail}.

sy_opt_pco(g_ok,[opt_pco,l_ok,PCOEXPR])-->
   sy_literal('@'),
   sy_expr(levels_all,sc_pco,EXPR),
   { EXPR=[ex_expr,E], PCOEXPR=[ex_pco_expr,E]},
   {!}.

sy_opt_pco(g_ok,[opt_pco,l_ok,[]])-->
   ex_opt_delim(_),
   {!}.

/*-------------------------------------------------------------------------*/
/* END OF MODULE sy_sc_2.pl                                                */
/*-------------------------------------------------------------------------*/
```

8. References

STATECRUNCHER documentation and papers by the present author

Main Thesis	[StCrMain]	The Design and Construction of a State Machine System that Handles Nondeterminism

Appendices

Appendix 1	[StCrContext]	Software Testing in Context
Appendix 2	[StCrSemComp]	A Semantic Comparison of STATECRUNCHER and Process Algebras
Appendix 3	[StCrOutput]	A Quick Reference of STATECRUNCHER's Output Format
Appendix 4	[StCrDistArb]	Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison
Appendix 5	[StCrNim]	The Game of Nim in Z and STATECRUNCHER
Appendix 6	[StCrBiblRef]	Bibliography and References

Related reports

Related report 1	[StCrPrimer]	STATECRUNCHER-to-Primer Protocol
Related report 2	[StCrManual]	STATECRUNCHER User Manual
Related report 3	[StCrGP4]	GP4 - The Generic Prolog Parsing and Prototyping Package *(underlies the STATECRUNCHER compiler)*
Related report 4	[StCrParsing]	STATECRUNCHER Parsing
Related report 5	[StCrTest]	STATECRUNCHER Test Models
Related report 6	[StCrFunMod]	State-based Modelling of Functions and Pump Engines

References

[CHSM] Paul J. Lucas
An Object-Oriented System for Implementing Concurrent, Hierarchical, Finite State Machines.
MSc. Thesis, University of Illinois at Urbana-Champaign, 1993

[Clocksin] W.F. Clocksin & C.S.Mellish
Programming in Prolog, 2nd Edition
Springer Verlag, 1984

[ECHSM] M.J. Hollenberg
Extended Hierarchical Concurrent State Machines, Syntax and Semantics
Philips Draft Nat. Lab. Report

[SwiPro] SWI-Prolog
http://www.swi-prolog.org

[WinPro] WinProlog, Logic Programming Associates Ltd
http://www.lpa.co.uk

[Yule 97] D.C. Yule
Automatic State-Based Testing
Philips PRL Technical Note TN 3611, 1997 / DVD Document V19 C4 S415.

