Rascal Cheat Sheet

http://www.rascal-mpl.org
https://github.com/usethesource/rascal

Modules

```
module Example

import ParseTree;         // import
extend lang::std::Layout; // "inherit"
```

Declarations

```
// Algebraic data types (ADT)
data Exp
  = var(str x)         // unary constructor
  | add(Exp l, Exp r); // binary constructor

data Person        // keyword parameter
  = person(int id, bool married=false);

alias Age = int; // type alias

anno loc Exp@location; // annotation

private real PI = 3.14; // variables

// Functions:  signatures are lists of patterns
// May have keyword parameters.
void f(int x) { println(x); }      // block style
int inc(int x) = x+1;              // rewrite style
int inc0(int x) = x+1 when x == 0; // side condition
default int inc0(int x) = x;       // otherwise

// Test functions (invoke from console with :test)
test bool add() = 1+2 == 3;

// randomized test function
test bool comm(int x, int y) = x+y == y+x;

// Foreign function interface to Java
@javaClass{name.of.javaClass.with.Method}
java int method();
```

```
// Context-free grammars
start syntax Prog       // start symbol
  = prog:  Exp* exps    // production
  | stats:  {Stat ";"}* // separated list
  | stats:  {Stat ";"}+ // one-or-more sep.  list
  | "private"?  Func;   // optional

syntax Exp
  = var:  Id
  | left mul:  Exp l "*" Exp r     // or right, assoc
  | left div:  Exp!div "/" Exp!div // reject
  > left add:  Exp l "+" Exp r     // ">" = priority
  | bracket "(" Exp ")";

lexical Comment
  = ^"#" ![\n]* $; // begin/end markers

lexical Id
  = ([a-zA-Z] !<<         // look behind restriction
    [a-zA-Z][a-zA-Z0-9_]* // character classes
    !>> [a-zA-Z0-9_])     // lookahead restriction
    \ Reserved;           // subtract keywords

layout Layout // for whitespace/comments
  = [\ \t\n\r]*;

keyword Reserved   // keyword class
  = "if" | "else"; // finite langs
```

Statements

```
// Standard control-flow
if (E) S;
if (E) S; else S;
while (E) S;
do S; while(E);
continue; break;
return; return E;

// Loop over all bindings produce by patterns
for (i <- [0..10]) S; // Loop 10 times

fail;     // control backtracking
append E; // add to loop result list
```

```
// Pattern-based switch-case
switch (E) {
  case P: S;   // do something
  case P => E  // rewrite it
  default:  S; // otherwise
}

// Traversal with visit; like switch, but matches
// at arbitrary depth of value
visit (E) {
  case P: S;  // do something
  case P => E // rewrite something
  case P => E when E
}
insert E; // rewrite subject as statement

// Strategies:  bottom-up, innermost, outermost,
// top-down-break, bottom-up-break
top-down visit (E) {}

try S;      // pattern-based try-catch
catch P: S; // match to catch
finally S;

throw E; // throw values

// Fix-point equation solving;
// iterates until all params are stable
solve (out,ins) {
  out[b] = ( {} | it + ins[s] | s <- succ[b] );
  ins[b] = (out[b] - kill[b]) + gen[b];
};

x = 1;                // assignment
nums[0] = 1;          // subscript assignment
nums[1,3..10] = 2;    // sliced (see below)
p.age = 31;           // field assignment
ast@location = l;     // annotation update
<p, a> = <"ed", 30>; // destructuring

// A op=E  == A = A op E
A += E; A -= E; A *= E;
A /= E; A &= E;
```

Expressions

```
// Standard operators
E + E; E - E; E * E; E / E; E % E;
E && E; E || E; E == E; E != E;
E > E; E >= E; E < E; E <= E; -E; !E;
E ? E : E;

// Projections
p.age;           // select field (tuple/constructor)
p[age=31];       // update field
ps<name,age>;    // select named column(s)
ps<1,0>;         // select/swap columns by position
graph["from"];   // image (list,str,map/rel/lrel)
alist[-1];       // subscript (last)
graph["from", "label"];
inc(2);          // function call
x[1..10];        // slicing (list, string)
x[0..]; x[..10]; // open slices
x[..-1];         // negative slicing (prefix)
x[0,2..10];      // slicing with next

[0..10];   // range (incl/excl)
[0,2..10]; // range with next

// Comprehensions
[ i*i | i <- [1..10] ];        // list
{ <i, i*i> | i <- [1..10] };   // set
( i:  i*i | i <- [1..10]);     // map
( 0 | it + i | i <- [1..10] ); // reducing

// Other operators
E mod E;  // modulo
E & E;    // intersection
E join E; // relation join
E o E;    // compostion

all(i <- [1..10], i > 0);      // big and
any(i <- [1..10], i % 2 == 0); // big or
E ==> E;                       // implication
E <==> E;                      // equivalence

E in E;    // membership
E notin E; // non-membership
E has N; // has label
E is N;  // is constructor

E+; // transitive closure
E*; // trans.  refl.  closure

E[N=E];  // update field
E[@N=E]; // update annotation

// Matching and generation
P := E;  // pattern match
P !:= E; // anti-match
P <- E;  // generator

// Closures
int(int x) { return x + 1; };
(str x) { println(x); }; // void
() { println("y"); };    // nillary void

// String templates
"x + y = <x + y>"; // interpolation

// Control-flow string interpolation (with for, if,
// while, do-while).  Tick (') indicates margin.
// Nested templates are auto-indented.
x = "<for (i<-[0..10]) {>
    '  <if (i % 2 == 0) {>
    '     i = <i>
    '  <}>
    '<}>";
```

Types and values

```
// Atomic types
bool x = true || false;
int x = 1;
real x = 2.3E-14;
rat x = 1r2;
num x = 1 + 3.0;
str x = "rascal";
datetime x = $1948-02-11$;
loc x = |file:///etc/passwd|;
loc x = |file://foo|(
   10,      // .offset
   5,       // .length
   <1, 2>   // .begin.line, .begin.column
   <1, 7>); // .end.line, .end.column

// Tuples
tuple[str, int] x = <"ed", 30>;
tuple[str name, int age] x = <"ed", 30>;

// Trees (all ADTs are subtype of node)
node x = "person"("ed", 30);     // generic node
Exp x = add(var("x"), var("y")); // ADT value
Exp e1 = (Exp)`x * y`;           // concrete
Exp e2 = (Exp)`a + (<Exp e1>)`;  // interpolation

// Collection values
list[int] x = [1,2,3];
set[bool] x = {true,false};
map[int, bool] x = (1:  true, 2:  false);
map[int n, bool b] x = (1:  true, 2:  false);
rel[int, bool] x = {<1, true>, <2, false>};
rel[int n, bool b] x = {<1, true>, <2, false>};
lrel[int n, bool b] x = [<1, true>, <1, true>];

// Functions
int(int,int)f = int(int x, int y) { return x+y; };

// Misc
value x = anything;   // top type
type[int] t = #int;   // reified types
int size(list[&T] l); // generics
```

Patterns

```
int x := 3;                          // typed
x := 3;                              // untyped
<int x, y> := <3, "x">;              // tuple
[1, 2, x] := [1, 2, 3];              // list
{x, 2, 3} := {2, 3, 1};              // set
[1, *xs, 4] := [1,2,3,4];            // splice-variable
add(l, r) := add(var("x"), var("y")); // constructor
/str x := add(var("x"), var("y"));   // deep
a:add(_,_) := x;                     // labeled
Exp a:add(_,_) := x;                 // typed/labeled
/[a-z]/ := "x";                      // regexp
/.<mid:[a-z]>./ := "abc";            // named groups


(Exp)`<Exp a> + <Exp b>` := e2;  // concrete matching
(Prog)`x, <{Exp ","}* es>` := p; // list matching
```