Lucid Language Cheat Sheet

Events

```
// declare a kind of event
event foo(type1 id1, type2 id2, ...);

// create an event value
event x = foo(arg1, arg2, ..., argN);

// pack arguments directly into packet
packet event foo(type1 id1, ...);

// send event to port n
generate_port(n, x);

// send event to switches in group g
generate_port(g, x);

// queue event for recirculation
generate(x);
```

Parser Declarations

```
// example parser implementation
parser main(bitstring pkt) {
  int<48> d = read(pkt);
  int<48> s = read(pkt);
  int<16> t = read(pkt);
  match t with
    | LUCID_ETHERTY -> {do_lucid_parsing(
    pkt);}
    | _ -> {
      skip(32, pkt);
      int<16> csum = hash<16>(checksum, d,
     s, t);
      generate(eth_with_csum(csum, d, s, t
    , Payload.parse(pkt))); }
}
```

Parser Actions

```
// read from bitstring
read(pkt);

// pop n bits off bitstring
skip(n, pkt);

// computes the n-bit checksum
hash<n>(checksum, a1, a2, ..., an);
```

Parser Steps

```
// generate event
generate(foo(a1, a2, ..., an));

// call to previously declared parser
my_parser(a1, a2, ..., an);

// drop the packet
drop;

// branch to parse blocks
match proto with
  | 0x800 -> {parse_ipv4(pkt);}
  | _ -> {...}
```

Handlers

```
// define handler for corresponding event
handle foo(type1 id1, type2 id2, ...) {
  ...
}
```

Arrays

```
// array with n entries of sz-bit ints
global Array.t<<sz>> arr = Array.create(n)

// return the value at index
Array.get(arr, idx);

// store the value at index
Array.set(arr, idx, v);

// apply memop setop to v using getarg
Array.getm(arr, idx, getop, getarg);

// apply memop setop to v using setarg
Array.setm(arr, idx, setop, setarg);

// return getop(v, getarg) and replace
    with setop(v, setarg)
Array.update(arr, idx, getop, getarg,
    setop, setarg);

// update for three-argument memop
Array.update_complex(arr, idx, memop, arg1
    , arg2, default)
```

Tables

```
// create a table
global Table.t<<key_ty, data_ty, arg_ty,
    ret_ty>> t = Table.create(sz, actions,
    default_action, default_data);

// table lookup
ret_ty result = Table.lookup(t, key, arg);

// install table entry; vendor dependent
Table.install(t, key, acn, data);
```

Memops

```
// two arguments: two forms

// single return
memop foo(int mem, int local) {
    return <e>;
}

// single if with single return per branch
memop foo(int mem, int local) {
    if (<e>) then { return <e>; } else {
    return <e>; }
}


// four arguments, structured like
memop foo(int mem1, int mem2, int local1,
    int local2) {
  bool b1 = <boolexp>; // May be omitted
  bool b2 = <boolexp>; // May be omitted

  // Omitted entirely, or just else branch
    if (<cond>) { cell1 = <ret_exp> } else
  { if (<cond>) { cell1 = <ret_exp> }

  // Omitted entirely, or just else branch
    if (<cond>) { cell2 = <ret_exp> } else
  { if (<cond>) { cell2 = <ret_exp> }

  // May be omitted. No else permitted
  if (<cond>) { return <local_exp> }
}

// three arguments; no mem2
memop foo(int mem, int loc1, int loc2)
```

Functions

```
// functions may contain arbitrary
    statements
fun rty foo(type1 idN, ..., typeN idN) {
    ...
}
```

Actions

```
// actions are not constructed directly
action_constr mk_my_acn(bool x) = {
  return action res_t _ (int a) {
    // single return statement
    return {val = a; is_hit = x};
  };
};
```

Match

```
// represent TCAM tables with static rules
match (e1, ..., eN) with
  | 42      -> { ... } // integers
  | 0b**10  -> { ... } // bitstrings
  | foo     -> { ... } // variables
  | _       -> { ... } // wildcard
```

Constants, Externs and Symbolics

```
// defined throughout the program
const <type> foo = ...;

// each switch may have different values
extern <type> foo;

// extern, except values in .symb file
symbolic <type> foo;
```

Groups

```
// set of ports, must be const ints
{0, 4, 7}

// generate group for all ports except x
flood x
```

Records

```
// declare records
{type1 label1; ...; typeN labelN}

// create records
{label1 = exp1; ...; labelN = expN}

// same as foo except given values
{foo with label3 = exp3'; label4 = exp4'}
```

Types

```
// user types, like C structs
type foo = {
  int<48> bar;
  int<16> baz;
}

// constructor, necessary for global types
constr <type> foo(<args>) = <expression>

global my_type x = constr_name(<<args>>)

// user defined size
size a = 16;
```

Vectors and Loops

```
// vector (immutable size)
int[4] v = [0; 3; 5; 7];

// get value at index
int i = v[2];

// loop (var is size type)
for (var < size) { ... }
```

Builtins

```
/* global */
self // id of the switch
recirculation_port

/* local */
this // event value that spawned handler
ingress_port
```

Miscellaneous

```
// c-like printf function
printf("Is %d > 10? %b", 3, 3 > 10);

// return size-bit int hash
hash<<size>>(seed, arg1, arg2, ...);

// standard ipv4 1s-complement checksum
hash<16>(checksum, arg1, arg2, ...);

/* system functions */
Sys.time(); // elapsed time in ns
Sys.random(); // random 32-bit int
```