OpIntSetDriver.C

```
#include <iostream.h>
#include "OpIntSet.h"

int main()
{
    OpIntSet s0,s1;

    s0.add(1); s1.add(1);
    s0.add(2); s1.add(2);
    s0.add(3); s1.add(3);

    cout << "s0 == s1: " << (s0 == s1) << endl;

    s0.remove(2);

    cout << "s0 == s1: " << (s0 == s1) << endl;
}
```

OpIntSet.C

```
#include "OpIntSet.h"

int operator==(const OpIntSet& lhs,const OpIntSet& rhs)
{
    if (lhs.curSize != rhs.curSize)
        return 0;

    for (int i = 0; i < lhs.curSize; i++)
        if (rhs.findPos(lhs.s[i]) < 0)
            return 0;

    return 1;
}
```

OpIntSet.h

```
#include "IntSet.h"

class OpIntSet : public IntSet {
public:
    friend int operator==(const OpIntSet&,const OpIntSet&);
};
```

SetDriver.C

```
#include <iostream.h>
#include "Set.h"

int main()
{
    Set<float> s;
    s.add(1.4142);
    s.add(1.732);
    s.add(3.1416);

    cout << "isMember(1.732): " << s.isMember(1.732) << endl;

    s.remove(1.732);

    cout << "isMember(1.732): " << s.isMember(1.732) << endl;
}
```

Set.C

```
template<class Element> Set<Element>::Set()
{
    curSize = 0;
}

template<class Element> void Set<Element>::add(Element x)
{
    if (findPos(x) >= 0) {
        throw DuplicateExc();
    } else if (curSize == MAXSIZE) {
        throw FullExc();
    }
    s[curSize++] = x;
}
```

Set.h

```
const int MAXSIZE = 100;

// exception classes
class DuplicateExc {};
class FullExc {};
class NotFoundExc {};

template<class Element> class Set {
public:
    Set();
    void add(Element);
    void remove(Element);
    int isMember(Element) const;
    int size() const;
protected:
    int findPos(Element) const;
    Element s[MAXSIZE];
    int curSize;
};
```

IntSetDriver.C

```
#include <iostream.h>
#include "IntSet.h"

int main()
{
    IntSet s;

    s.add(1);
    s.add(2);
    s.add(3);
    cout << "isMember(2): " << s.isMember(2) << endl;
    s.remove(2);
    cout << "isMember(2): " << s.isMember(2) << endl;
    try {
        s.remove(2);
    }
    catch (NotFoundExc) {
        cout << "NotFoundExc caught" << endl;
    }
}
```

IntSet.C

```
#include "IntSet.h"

IntSet::IntSet()
{
    curSize = 0;
}

void IntSet::add(int x)
{
    if (findPos(x) >= 0) {
        throw DuplicateExc();
    } else if (curSize == MAXSIZE) {
        throw FullExc();
    }
    s[curSize++] = x;
}
```

IntSet.h

```
const int MAXSIZE = 100;

// exception classes
class DuplicateExc {};
class FullExc {};
class NotFoundExc {};

class IntSet {
public:
    IntSet();
    void add(int);
    void remove(int);
    int isMember(int) const;
    int size() const;
protected:
    int findPos(int) const;
    int s[MAXSIZE];
    int curSize;
};
```

Work Product Criteria—Test Impl'n

For the module under test, M:

- Audience
  - those considering running or modifying M's testing
- Prerequisites
  - a thorough understanding M's MIS and some understanding of M's MID
- Purpose
  - implement the TP simply and inexpensively

Work Product Criteria—Test Plan

For the module under test, M:

○ Audience

- those considering running or modifying M's testing.

○ Prerequisites

- a thorough understanding M's MIS and some understanding of M's MID

○ Purpose

- serve as a planning tool for the TI
- document the TI

BSHAM System Testing (con.)

test implementation strategy

target `runtest1` in `Makefile`

test cases for command-line errors

for each file f in `input/`

bsham f **>** **act/**f

diff act/f **exp/**f

statement coverage for $sham$ and $exec$ measured using the Unix utility *tcov*

100% coverage for statements not associated with ISHAM

BSHAM System Testing (con.)

test cases

- command-line errors
  - hardcoded in Makefile
- load-time exceptions
  - `ldexc`: one load-time exception
- run-time exceptions
  - `addrexc`: $mem[pc] = ADD.object$, $mem[pc+1] = 100$
  - `arithexc`: $acc = 500 + 500$
  - `noopexc`: $pc = 99$, $mem[pc] = LOADCON.object$
  - `objectexc`: $mem[pc] = 10$
- normal-case
  - `halt[1-3]`: $HALT$ instruction, check with $PRINT$ instruction
  - `print`: print special values
  - `two+two,sum`: programs from Appendix of RS

BSHAM System Testing (con.)

test case selection strategy

special values

command-line errors

each command-line error once

content of input file

one load-time exception

every run-time exception once

SHAM instructions

halt with $pc = \{0, \texttt{SY\_MEMSIZ}/2, \texttt{SY\_MEMSIZ} - 1\}$

print with interval rule on the contents of the accumulator

BSHAM System Testing

test environment

entire BSHAM system
input stored in files
output saved in files, checked with delta testing
directory structure:

`sham/`
- `input/` - test cases stored one per file
- `exp/` - expected results of test case (same file name)
- `act/` - actual results of test case (same file name)

`sham/cmdlin.exp` - expected results for command-line test cases
`sham/cmdlin.act` - actual results for command-line test cases

System Testing

- Test the system in its production environment
  - test whether system satisfies its RS
  - focus on parts of system not yet tested

    —$exec$ and $sham$

- Other forms of system testing
  - volume and stress testing
  - reliability testing
  - recovery testing
  - acceptance testing

$sham$ Testing

- Straightforward implementation
- Postpone $sham$ testing until system test

exec Testing

- External interaction
  - stubs for *absmach* provide poor controllability

    —use production code
  - loading memory in *absmach* from test script tedious

    —use production *load* and *token*
  - hard to use PGMGEN

    —use production *sham* Coordinator

- Production code for all of *sham*
  - postpone all *exec* testing until system test

load Testing

- Controllability problems
  - reads input from a file
  - cannot use PGMGEN
  - customized driver or *sham* Coordinator as driver?
    —*sham* Coordinator

- Observability problems
  - parses strings using *token*
  - loads memory using *absmach*
  - stubs or production code for *token* and *absmach*?
    —*token*: production code
    —*absmach*: stubs

absmach TP

- PGMGEN driver

- Special module state values
  - based on contents of mem, pc and acc
  - every run-time exception at least once
  - every instruction at least once

- Special parameter values
  - **am_s_acc**(i), **am_s_pc**(i), **am_s_mem**(a, i), **am_g_mem**(a)
    —interval rule for a and i

token TI

```
int cmp_valtyp(actvtp,expvtp)
tk_valtyp *actvtp,*expvtp;
{
     if (!strcmp(actvtp->val,expvtp->val))
          return(actvtp->typ == expvtp->typ);
     else
          return(0);
}

void prt_valtyp(actvtp,expvtp)
tk_valtyp *actvtp,*expvtp;
{
     printf("Expected value:<%s,%d>. Actual value:<%s,%d>\n",
             expvtp->val,expvtp->typ,actvtp->val,actvtp->typ);
}
```

token TP

- PGMGEN driver

- Special module state values
  - number of tokens in sequence: 0, 1, and 3 tokens
  - types of tokens: `TK_INT`, `TK_ID`, and `TK_BADTOK`

- Special parameter values—`tk_s_str`(s)
  - interval rule on length of s
  - vary number of blanks before/after tokens

Design for Testability

- To make testing affordable and effective
  - testability must be considered at design time
  - key considerations: controllability and observability

- Controllability
  - ease with which an arbitrary input may be applied
  - example: call to initialize a module

- Observability
  - ease with which the output may be observed
  - example: undisciplined use of print statements

Comparing Non-standard Types

```
float epsilon = 0.00001;

int cmp_fuzz(aval,eval)
float aval,eval;
{
     float diff;

     diff = eval-aval;
     if (diff > epsilon || diff < -epsilon) return(0);
     else return(1);
}

int prt_fuzz(aval,eval)
float aval,eval;
{
     printf("expval:%f. actval:%f. epsilon:%f\n",eval,aval,epsilon);
}
```

Comparing *actval* and *expval* (con.)

```
int cmp_bool(aval,eval)
int aval,eval;
{
     if ((eval == 0) && (aval == 0)) return(1);
     else if ((eval != 0) && (aval != 0)) return(1);
     else return(0);
}

int prt_bool(aval,eval)
int aval,eval;
{
     printf("\tExpected value:%d.  Actual value:%d\n",eval,aval);
}
```

Comparing *actval* and *expval*

- To compare *actval* and *expval* of type T
  - call boolean C function `cmp_`T(*actval*, *expval*)
  - if it returns false, call `prt_`T(*actval*, *expval*)

- Example:
  - `<s_init().s_push(10), noexc, g_top(), 10, int>`
  - compare function call: `cmp_int(g_top(),10)`
  - print function call: `prt_int(g_top(),10)`

Example—*symtbl* (con.)

```
{%
t_init();
t_next();
while (!t_end()) {
%}
     < , noexc, g_siz(), t_siz(), int>
     < , noexc, g_exsym(t_mksym(0,T_MKSYMMAX)), 0, bool>
{%
     for (i = 0; i < ST_MAXSYMS; i++) {
          if (i < t_siz()) {
%}
               < , noexc, g_exsym(t_sym(i)), 1, bool>
{%        } else {  %}
               /*check t_sym(i) not in table*/
               < , noexc, g_exsym(t_sym(i)), 0, bool>
{%        }
     }
     t_next();
} %}
```

Example—*symtbl* (con.)

test implementation strategy

C functions to support iterating over
the special module states, viewed as a sequence:

`void t_init`: set position to before first state

`void t_next`: load next state into symtbl

`int t_end`: true iff position is after last state

`int t_siz`: number of symbols in current state

`char* t_sym`(i): i-th symbol in current state

`char* t_mksym`(i, l): string consisting of i
converted to ASCII,
padded right with *'s to length l

Example—*symtbl*

...

test case selection strategy

special values

module state

number of symbols in table: $\{0, 1, \texttt{ST\_MAXSYMS}/2, \texttt{ST\_MAXSYMS}\}$

symbol length: short, `ST_MAXSYMLEN`

test cases

normal

for each special module state

check table length

for each i in $[0, \texttt{ST\_MAXSYMS} - 1]$

if i in $[0, \texttt{t\_siz} - 1]$

check $\texttt{t\_sym}(i)$ in table with correct location

else

check $\texttt{t\_sym}(i)$ not in table

Embedded C Code

- Arbitrary C Code delimited by `{%` and `%}` may appear
  - between test cases
  - as a call in a trace of a test case
  - as the *actval* or *expval* field of a test case
  - as a parameter of any call

- Example

```
{% for (i = 1; i <= 10; i++ ) { %}
        <s_init().s_push(10*i), noexc, g_top(), 10*i, int>
        <s_init().s_push(10*i), noexc, g_depth(), 1, int>
{% } %}
```

Example—*stack* Test Implementation

```
cases
...

/*partially full stack*/
<load(2).s_push(30), noexc, g_top(), 30, int>
<load(2).s_push(30), noexc, g_depth(), 3, int>
<load(2).s_pop(), noexc, g_top(), 10, int>
<load(2).s_pop(), noexc, g_depth(), 1, int>
<load(2), noexc, g_top(), 20, int>
<load(2), noexc, g_depth(), 2, int>

...
```

Example—*stack* Test Implementation

```
globcod
{%
#include "system.h"
#include "stack.h"

static void load(n)
int n;
{
     int i;

     ps_s_init();
     for (i = 0; i < n; i++)
          ps_s_push((i+1)*10);
}
%}
```

Example—`stack` Test Plan

assumptions

$\texttt{PS\_MAXSIZ} > 2$

test environment

PGMGEN driver

no stubs

...

test implementation strategy

$\texttt{load}(n)$

loads stack with $10, 20, \ldots, 10 \times n$

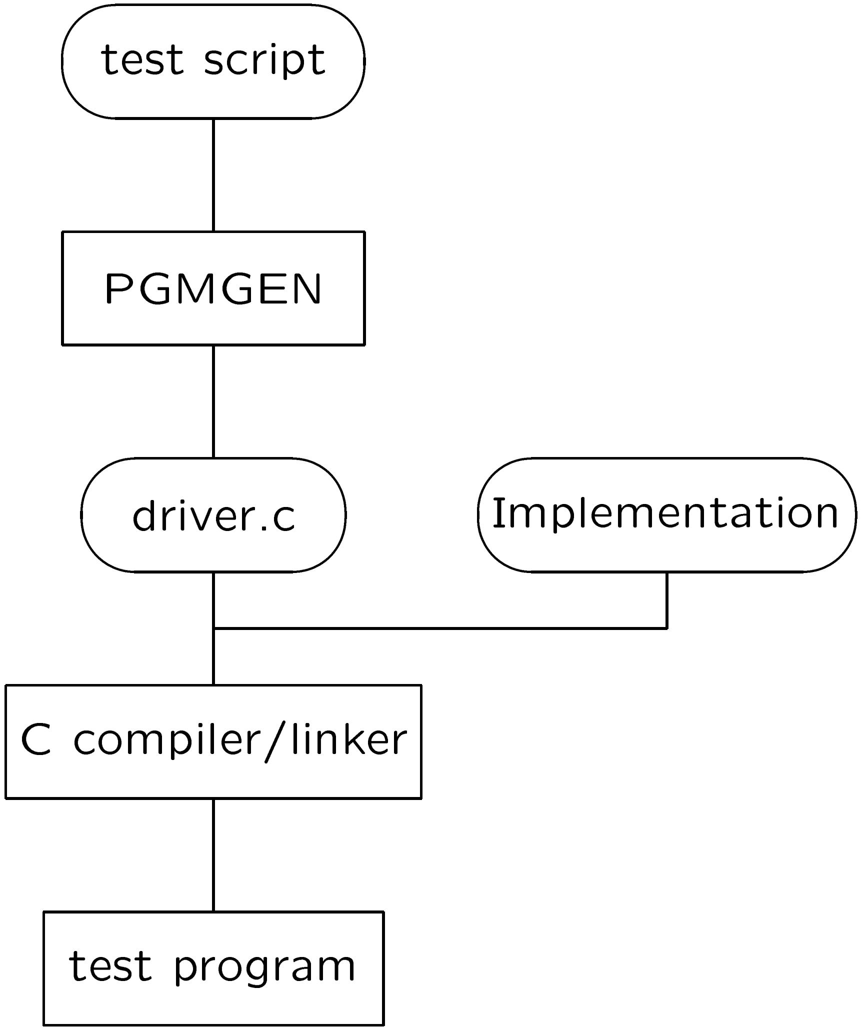
statement coverage measured using the `tcov` utility

PGMGEN Test Driver Generation

- For each case: $\langle$*trace*,*expexc*,*actval*,*expval*,*type*$\rangle$

    invoke $c_1, \cdots, c_n$, monitoring exception occurrences
    compare the actual occurrences to *expexc*
    if there are any differences
        print a message
    else
        if *actval* $\neq$ *expval*
            print a message
        if exceptions occurred since c_n was invoked
            print a message
    update summary statistics

The PGMGEN Test Driver Generator



PGMGEN Test Script—stack

```
module
     ps_

accprogs
     <s_init,s_push,s_pop,g_top,g_depth>

exceptions
     <empty,full>

globcod
{%
#include "system.h"
#include "stack.h"
%}

cases
<s_init().g_top(), empty, dc, dc, dc>
<s_init().s_push(10), noexc, g_top(), 10, int>
```

PGMGEN Testing Tool

- Test case: ⟨*trace*,*expexc*,*actval*,*expval*,*type*⟩
  - *trace*

    a *trace* used to exercise the module
  - *expexc*

    exception *trace* is expected to generate
  - *actval*

    an expression evaluated after *trace*
  - *expval*

    the value that *actval* is expected to have
  - *type*

    the data type of *actval* and *expval*

Test Case Selection—`stack`

test case selection strategy

special values

module state

interval rule on stack size: [0, `MAXSIZ`]

access routine parameters

none

test cases

For each of the special module state values

call `s_push`, `s_pop`, `g_top`, `g_depth`

check exception behavior

after set calls, check get call values

Our Approach

- Functional testing for test case selection
  - consider the MIS

    special values for parameters and module state
  - check MID and MI for other special values
- Structural analysis as a cross-check
  - measure statement coverage with `tcov`
  - 100% coverage necessary, but not sufficient

Structural Testing—Limitations

- Path testing
  - number of test paths often infinite
  - many paths are infeasible
- In general, structural coverage is not sufficient
- For the code below, the set {2} achieves path coverage

```
int double(x)
{
    return(x*x);
}
```

Structural Testing

- “Cover” as many parts of the program as possible

```
void tst(x)
int x;
{
     if (x > 0)
          pos = pos+1;
     if (x % 2 == 0)
          even = even+1;
     return;
}
```

- Statement coverage: $\{2\}$
- Branch coverage: $\{-1, 2\}$
- Path coverage: $\{-2, -1, 1, 2\}$

Functional Testing—Example

- Access routine $f(p_1, p_2)$
  - p_1 is an integer restricted to interval [0,100]
  - p_2 belongs to enumerated type $\{red, green, blue\}$

- Special values for p_1 : the interval rule
  - normal case values: $\{0, 50, 100\}$
  - exceptional values: $\{-100, 1, 101, 1000\}$

- Special values for p_2
  - normal case values: $\{red, green, blue\}$
  - no exceptional values

- Test cases: cover all 21 combinations

Functional Testing

- Choose special values for access routine parameters
  - both exceptional and normal case values
- Choose special values for MID state
  - only normal case values
- Test all combinations of special values
  - combinatorially explosive
  - there may be dependencies

Test Case Selection

- Functional testing
  - tests are based on specification
  - largely ignore the implementation
- Structural testing
  - tests are based on the implementation
  - ignores missing code

Top-Down vs. Bottom-Up Testing

- Top-down testing
  - \+ early availability of executable program
  - \+ early detection of flaws in interface design
  - – poor controllability and observability
  - – cost of developing and maintaining stubs
- Bottom-up testing
  - \+ better controllability and observability
  - – cost of developing and maintaining drivers
  - – executable program available very late
- Use a mixture of top-down and bottom-up testing

Bottom-Up Testing

- Test standalone module using drivers
- Gradually replace drivers by production code

module	t_1	t_2	t_3
$sham$	not used	not used	MI
$load$	not used	MI	MI
$exec$	not used	MI	MI
$token$	MI	MI	MI
$absmach$	MI	MI	

Top-Down Testing

- Test top-level module using stubs

- Gradually replace stubs by production code

module	t_1	t_2	t_3	t_4
$sham$	MI	MI	MI	MI
$load$	stubs	MI	MI	MI
$exec$	stubs	stubs	stubs	MI
$token$	stubs	MI	MI	MI
$absmach$	stubs	stubs	MI	MI

Module and System Testing

- Hard to test a module in its production environment
  - poor controllability and observability
  - example: SHAM exception handlers never called

- Module testing
  - test module in isolation using stubs and drivers

- Integration testing
  - test combination of modules as subsystems

- System testing
  - test entire system

Test Drivers

- Interactive drivers
  - prompts user for access routine name, parameters
  - prints return values and exceptions
  - good for debugging
  - tedious when there are many test cases
  - tedious when cases must be repeated many times
- PGMGEN driver
  - simple test script language
  - automates execution and checking of results
- Customized batch drivers
  - modules that interact with the environment

Test Implementation

- TI implements the TP as simply and inexpensively as possible

- Includes
  - test scaffolding such as drivers and stubs
  - test data files
  - manual and automated procedures for executing tests

Test Plan

- TP serves as a specification for the TI Implementation

- Required sections

  assumptions

  assumptions on which testing depends

  test environment

  environment in which testing is performed

  test case selection strategy

  choice of test cases

  test implementation strategy

  key aspects of the TI

Testing Tasks

1. Build the test harness
2. Generate the test inputs
3. Determine the expected outputs
4. Execute the test cases, monitoring the behavior
5. Compare actual outputs to expected outputs
6. Evaluate the test results

- Focus automation on frequently repeated tasks: 4–6

Systematic Testing

- Ad-hoc approach to testing is
  - ineffective: too late to influence design decisions
  - expensive: testing not reused during maintenance
- Systematic testing
  - planned: permit design for testability
  - documented: tests can be understood and evaluated
  - maintained: tests executed after every change
- Work products
  - Test Plan: strategy for selecting and executing tests
  - Test Implementation: implements the Test Plan

scndr.c

```
void sd_s_mem()
{
        sg_fld f;
        int r,c;
        char s[SG_NUMCOL+1];

        for (r = 0; r < 10; r++) {
                for (c = 0; c < 10; c++) {
                        FLD(f,SG_MEM,r,c);
                        sprintf(s,"%d",am_g_mem(10*r+c));
                        rjust(s,sg_g_len(f));
                        ss_s_str(sg_g_row(f),sg_g_col(f),s);
                }
        }
        ss_s_ref();
}
```

scndr.c

```
void sd_s_con()
{
        int i;

        prtcon(SG_SCNTTL,0,0);
        prtcon(SG_MEMTTL1,0,0);
        prtcon(SG_MEMTTL2,0,0);
        for (i = 0; i < 10; i++) {
                prtcon(SG_MEMCOLHDR,0,i);
                prtcon(SG_MEMROWHDR,i,0);
        }
        prtcon(SG_PCTTL,0,0);
        prtcon(SG_ACCTTL,0,0);
        prtcon(SG_PRTTTL,0,0);
        prtcon(SG_PROMPTTTL,0,0);
        prtcon(SG_MSGTTL,0,0);
        ss_s_ref();
}
```

scndr.c

```
/*pad s on the left with blanks to length l
*        truncate s to l if longer than l
*/
static void rjust(s,l)
char *s;
int l;
{
        int shift;

        s[l] = '\0';
        shift = l-strlen(s);
        while (--l >= shift)
                s[l] = s[l-shift];
        while (l >= 0)
                s[l--] = ' ';
}
```

scndr.c

```
#define FLD(f,t,r,c) (f.nam = t, f.row = r, f.col = c)

/*print constant field with fieldname t, row r, and column c*/
static void prtcon(t,r,c)
sg_fldnam t;
int r,c;
{
        sg_fld f;

        FLD(f,t,r,c);
        ss_s_str(sg_g_row(f),sg_g_col(f),sg_g_val(f));
}
```

scngeom.c

```
int sg_g_row(fld)
sg_fld fld;
{
        if (!legfld(fld)) {
                sg_badfld();
                return(0);
        }
        return(fldtbl[(int)fld.nam].row+fld.row);
}
```

scngeom.c

```
/*out := (fld is a legal field identifier)*/
static int legfld(fld)
sg_fld fld;
{
        int maxrow,maxcol;

        if (fld.nam == SG_MEM || fld.nam == SG_MEMROWHDR)
                maxrow = 9;
        else
                maxrow = 0;
        if (fld.nam == SG_MEM || fld.nam == SG_MEMCOLHDR)
                maxcol = 9;
        else
                maxcol = 0;
        return(
                (int)fld.nam >= 0 && (int)fld.nam <= (int)SG_MSGTTL &&
                fld.row >= 0 && fld.row <= maxrow &&
                fld.col >= 0 && fld.col <= maxcol
        );
}
```

scngeom.c

```
static struct {
        int row,col,len;
        char *val;
} fldtbl[] = {
        /*SG_MEM*/        {5,15,3,""},
        /*SG_PC*/         {17,45,2,""},
        /*SG_ACC*/        {18,44,3,""},
        /*SG_PRT*/        {19,44,3,""},
        /*SG_MSG*/        {23,9,63,""},
        /*SG_SCNTTL*/     {0,33,4,"SHAM"},
        /*SG_MEMTTL1*/    {5,0,4,"Main"},
        /*SG_MEMTTL2*/    {6,0,7,"memory:"},
        /*SG_MEMCOLHDR*/  {3,15,3,""}, /*depends on row*/
        /*SG_MEMROWHDR*/  {5,9,3,""}, /*depends on column*/
        /*SG_PCTTL*/      {17,24,19,"   Program counter:"},
        /*SG_ACCTTL*/     {18,24,19,"       Accumulator:"},
        /*SG_PRTTTL*/     {19,24,19,"Last value printed:"},
        /*SG_PROMPTTTL*/  {22,0,46,
                "Enter command: \"s\" to single step; \"e\" to exit"},
        /*SG_MSGTTL*/     {23,0,8,"Message:"} };
```

Work Product Criteria (con.)

- Well-formed
  - obeys MID format
- Comprehensible
  - by the intended audience
- Correct
  - with respect to the MIS
- Feasible
  - can be implemented affordably

Work Product Criteria

- Audience
  - module designer and implementor
- Prerequisites
  - understanding of the MIS
  - understanding of the C constructs for declarations
- Purpose
  - specify access routine behavior
    in terms of concrete state

sham MID

- Coordinator module
  - no access routines
  - interface specified in the RS

- MID omitted

$\boxed{exec}$ MID

○ $exec$ MIS—external interaction

- scn and $stdout$ environment variables
- calls $absmach$ access routines

○ $exec$ MI—external interaction

- calls $scndr$ access routines and `printf`
- calls $absmach$ access routines

○ MID omitted

- no concrete state

load MID

- *load* MIS—external interaction
  - *stdout* environment variable
  - references *absmach* state variables

- *load* MI—external interaction
  - calls to `printf`
  - calls to `am_s_mem`

- MID omitted
  - no concrete state

$\boxed{absmach \text{ MID}}$

- Abstract state

  acc : [0..AM_MAXINT]

  pc : [0..AM_$MEMSIZ - 1$]

  mem : $sequence$ [AM_MEMSIZ] $of\ integer$

- Concrete state

```
int acc,pc;
int mem[AM_MEMSIZ];
```

- State invariant

  1. acc $\in$ [0..AM_MAXINT]
  2. pc $\in$ [0..AM_$MEMSIZ - 1$]
  3. $(\forall i \in [0..\texttt{AM\_MEMSIZ} - 1])(\texttt{mem}[i] \in [0..\texttt{AM\_MAXINT})])$

- Access routine semantics, abstraction function omitted

token MID—Semantics (con.)

`tk_sg_next`:

transition/output:

Let $curtok$ be the token beginning at $\mathtt{buf}[\mathtt{cur}]$

$curtoktyp$ be the token type of $curtok$

$out := \langle curtok, curtoktyp \rangle$

$\mathtt{cur} :=$ (there is a token, beginning at position i,
in $\mathtt{buf}[\mathtt{cur} + |curtok| + 1..leftnull(\mathtt{buf}) - 1] \Rightarrow i$
$|\ true \Rightarrow leftnull(\mathtt{buf}))$

exceptions: $exc := (\mathtt{buf}[\mathtt{cur}] = null \Rightarrow \mathtt{tk\_end})$

`tk_g_end`:

output: $out := (\mathtt{buf}[\mathtt{cur}] = \text{null} \Rightarrow true \mid true \Rightarrow false)$

exceptions: none

token MID—Access Routine Semantics

access routine semantics

`tk_s_init`:

transition: $\texttt{buf}, \texttt{cur} := \texttt{""}, 0$

exceptions: none

$\texttt{tk\_s\_str}(s)$:

transition:

$$\begin{aligned}\texttt{buf} := (&\text{there is a token in } s[0..leftnull(s)-1] \Rightarrow \\ &\quad rmblanks(s) \parallel \texttt{" "} \\ &\mid true \Rightarrow \texttt{""})\end{aligned}$$

$\texttt{cur} := 0$

exceptions: $exc := (|s| > \texttt{TK\_MAXSTRLEN} \Rightarrow \texttt{tk\_maxlen})$

$\boxed{token \text{ MID—Local Functions}}$

local functions

$leftnull : string \rightarrow integer$

$leftnull(s)$:= (there is a null in $s \Rightarrow$
the index of the leftmost one)

$rmblanks : string \rightarrow string$

$rmblanks(s)$:= s, with leading blanks removed

$\boxed{token \text{ MID}}$

state variables

```
char buf[TK_MAXSTRLEN+2];
int cur;
```

state invariant

1. $\texttt{buf}[0..\texttt{TK\_MAXSTRLEN} + 1]$ contains a null
2. $\texttt{cur} \in [0..leftnull(\texttt{buf})]$
3. $\texttt{cur} < leftnull(\texttt{buf}) \rightarrow \texttt{buf}[\texttt{cur}] \neq \texttt{' '} \wedge$
$\texttt{buf}[leftnull(\texttt{buf}) - 1] = \texttt{' '}$

abstraction function

$toklist =$ the sequence of tokens in $\texttt{buf}[\texttt{cur}..leftnull(\texttt{buf}) - 1]$

token MID—Concrete State

- Reduce the number of special cases
  - always advance `cur` to start of the next token
  - use trailing blank as sentinel

- Concrete state

```
char buf[TK_MAXSTRLEN+2];
int cur;
```

token MID—Abstract & Concrete State

- Abstract state

  toklist : *sequence of string*

- Possible concrete state

  ```
  char s[TK_MAXSTRLEN+1]; /*string passed to s_str */
  int cur; /*index of next character*/
  ```

- Special cases
  - string with only blanks
  - blanks before first token: present or not
  - blanks after last token: present or not

symtbl MID—Access Routine Semantics

$\texttt{st\_s\_add}(sym, loc)$:

transition: $\texttt{tblcnt}, \texttt{tbl}[\texttt{tblcnt}] := \texttt{tblcnt} + 1, \langle sym, loc \rangle$

exceptions: $exc :=$

$(|sym| > \texttt{ST\_MAXSYMLEN} \Rightarrow \texttt{st\_maxlen}$

$|\ findsym(sym) \neq \texttt{NOTFOUND} \Rightarrow \texttt{st\_exsym}$

$|\ \texttt{tblcnt} = \texttt{ST\_MAXSYMS} \Rightarrow \texttt{st\_full})$

$\texttt{st\_g\_exsym}(sym)$:

output: $out := (findsym(sym) \neq \texttt{NOTFOUND})$

exceptions: none

symtbl MID—Constants and Functions

local constants

```
#define NOTFOUND -1
```

local functions

$findsym : string \rightarrow integer$

$$findsym(s) = ((\exists i \in [0..\mathtt{tblcnt} - 1])(s = \mathtt{tbl}[i].\mathtt{sym}) \Rightarrow i \\ \mid true \Rightarrow \mathtt{NOTFOUND})$$

symtbl MID

state variables

```
struct { char sym[ST_MAXSYMLEN+1];
         int loc;
} tbl[ST_MAXSYMS];
int tblcnt;
```

state invariant

- every symbol in $\texttt{tbl}[0..\texttt{tblcnt}-1]$ contains a null
- there are no duplicate symbols in $\texttt{tbl}[0..\texttt{tblcnt}-1]$
- $\texttt{tblcnt} \in [0..\texttt{ST\_MAXSYMS}]$

abstraction function

$$intspec.tbl = \{\langle sym, loc\rangle \mid (\exists i \in [0..\texttt{tblcnt}-1]) \\ (sym = \texttt{tbl}[i].\texttt{sym} \wedge loc = \texttt{tbl}[i].\texttt{loc})\}$$

symtbl MID—Abstract & Concrete State

- Abstract state

  $tbl : set\ of\ tuple\ of\ (sym : string, loc : integer)$

- Concrete state

```
struct {
      char sym[ST_MAXSYMLEN+1];
      int loc;
} tbl[ST_MAXSYMS];
int tblcnt;
```

$stack$ MIS and MID—Execution Tables

Call	Abstract state	Concrete state	Output	Exception
`s_init`	$\langle\rangle$	$\langle\langle ?, ?, ?\rangle, 0\rangle$	—	—
`s_push(1)`	$\langle 1\rangle$	$\langle\langle 1, ?, ?\rangle, 1\rangle$	—	—
`s_push(2)`	$\langle 1, 2\rangle$	$\langle\langle 1, 2, ?\rangle, 2\rangle$	—	—
`g_top`	$\langle 1, 2\rangle$	$\langle\langle 1, 2, ?\rangle, 2\rangle$	2	—

Call	Abstract state	Concrete state	Output	Exception
`s_init`	$\langle\rangle$	$\langle\langle ?, ?, ?\rangle, 0\rangle$	—	—
`s_push(1)`	$\langle 1\rangle$	$\langle\langle 1, ?, ?\rangle, 1\rangle$	—	—
`s_pop`	$\langle\rangle$	$\langle\langle 1, ?, ?\rangle, 0\rangle$	—	—
`g_top`	$\langle\rangle$	$\langle\langle 1, ?, ?\rangle, 0\rangle$	?	`empty`

State Invariant and Abstraction Function

- Documentation of the concrete state
  - state invariant: indicates legal states
  - abstraction function: the “meaning” of a state

- After every call to an access routine
  - state invariant holds
  - A(concrete state) = abstract state

stack MID (con.)

$\texttt{ps\_s\_pop}$:

transition: $\texttt{siz} := \texttt{siz} - 1$

exceptions: $exc := (\texttt{siz} = 0 \Rightarrow \texttt{ps\_empty})$

$\texttt{ps\_g\_top}$:

output: $out := \texttt{stack}[\texttt{siz} - 1]$

exceptions: $exc := (\texttt{siz} = 0 \Rightarrow \texttt{ps\_empty})$

$\texttt{ps\_g\_depth}$:

output: $out := \texttt{siz}$

exceptions: none

$stack$ MID (con.)

access routine semantics

`ps_s_init`:

transition: $\texttt{siz} := 0$

exceptions: none

`ps_s_push`(x):

transition: $\texttt{stack[siz]}, \texttt{siz} := x, \texttt{siz} + 1$

exceptions: $exc := (\texttt{siz} = \texttt{PS\_MAXSIZ} \Rightarrow \texttt{ps\_full})$

$\boxed{stack \text{ MID}}$

state variables

```
int stack[PS_MAXSIZ];
int siz;
```

state invariant

$\texttt{siz} \in [0..\texttt{PS\_MAXSIZ}]$

abstraction function

$|s| = \texttt{siz} \land (\forall i \in [0..\texttt{siz} - 1])(s[i] = \texttt{stack}[i])$

Work Product Definition

- Similar to MIS
  - interface syntax omitted—same as MIS
  - semantics based on MSM
  - added: state invariant and abstraction function
- State invariant
  - predicate on state space
  - restricts legal states
- Abstraction function
  - function from concrete states to abstract states

Abstract State versus Concrete State

- Abstract state chosen for clarity and simplicity

- Concrete state required because
  - abstract state unavailable in MI language
  - abstract state inefficient

- MID sometimes omitted
  - when abstract and concrete state are the same
  - when there is no concrete state

Role of MID

- Differences between MIS and MI
  - different state space
  - different expression language

- Separate the two steps
  - MID handles change in state space
  - MIS: abstract state
  - MID: concrete state

$scndr$ MIS—semantics

access routine semantics

Note: MEM, PC, ACC, and MSG, are screen fields from the ISHAM RS

$\ldots$

`sd_s_con`:

transition: display the fixed screen fields

exceptions: none

`sd_s_mem`:

transition:

$(\forall r, c \in [0..9]) MEM[r, c] := \texttt{am\_g\_mem}(10 \times r + c)$,
converted to ASCII, right justified and
padded left with blanks

exceptions:

$((\exists r, c \in [0..9]) \texttt{am\_g\_mem}(10 \times r + c) \notin [0..\texttt{SY\_MAXINT}] \Rightarrow \texttt{sd\_len})$

$\ldots$

scndr MIS—syntax

Routine names	Inputs	Outputs	Exceptions
sd_s_init			
sd_s_clrscn			
sd_s_con			
sd_s_mem			
sd_s_pc			
sd_s_acc			
sd_s_prt	int		
sd_s_msg	char*		
sd_s_hlt	int *boolean*		

$scnstr$ MIS—semantics (con.)

access routine semantics

`ss_s_init`:

$\ldots$

`ss_s_clrscn`:

$\ldots$

`ss_s_str`(row, col, s):

transition: $(|s| > 0 \Rightarrow scnbuf[row][col..col + |s| - 1] := s)$

exceptions: $exc :=$

$(row \notin [0..\texttt{SS\_NUMROW} - 1] \Rightarrow \texttt{ss\_row}$

$|\ col \notin [0..\texttt{SS\_NUMCOL} - 1] \Rightarrow \texttt{ss\_col}$

$|\ |s| \notin [0..\texttt{SS\_NUMCOL} - col] \Rightarrow \texttt{ss\_len})$

`ss_s_hlt`(row, col, l, f):

$\ldots$

`ss_s_cur`(row, col):

$\ldots$

`ss_s_ref`:

transition: $scn, hlt, cur := scnbuf, hltbuf, curbuf$

exceptions: none

`ss_s_end`:

$\ldots$

$\boxed{scnstr \text{ MIS—semantics}}$

environment variables

scn : $sequence$ [$\mathtt{SS\_NUMROW}$][$\mathtt{SS\_NUMCOL}$] $of\ char$
    $scn[r][c]$ is the character at screen row r and column c,
    with numbering zero-relative and beginning at the upper-left corner

hlt : $sequence$ [$\mathtt{SS\_NUMROW}$][$\mathtt{SS\_NUMCOL}$] $of\ boolean$
    $hlt[r][c]$ is true if the position at row r and column c is highlighted,
    with numbering zero-relative and beginning at the upper-left corner

cur : $tuple\ of\ (row : [0..\mathtt{SS\_NUMROW} - 1], col : [0..\mathtt{SS\_NUMCOL} - 1])$
    the terminal cursor is at screen row $cur.row$ and column $cur.col$
    with numbering zero-relative and beginning at the upper-left corner

state variables

$scnbuf$: $sequence$ [$\mathtt{SS\_NUMROW}$][$\mathtt{SS\_NUMCOL}$] $of\ char$

$hltbuf$: $sequence$ [$\mathtt{SS\_NUMROW}$][$\mathtt{SS\_NUMCOL}$] $of\ boolean$

$curbuf$: $tuple\ of\ (row : [0..\mathtt{SS\_NUMROW} - 1], col : [0..\mathtt{SS\_NUMCOL} - 1])$

scnstr MIS—syntax

```
#define SS_NUMROW 24
#define SS_NUMCOL 80
```

Routine names	Inputs	Outputs	Exceptions
ss_s_init			
ss_s_clrscn			
ss_s_str	int int char*		ss_row ss_col ss_len
ss_s_hlt	int int int *boolean*		ss_row ss_col ss_len
ss_s_cur	int int		ss_row ss_col
ss_s_ref			
ss_s_end			

$scngeom$ MIS—semantics (con.)

access routine semantics

`sg_s_init`:

  transition: none

  exceptions: none

`sg_g_legfld`(fld):

  output: out := (fld is a legal field identifier)

  exceptions: none

`sg_g_row`(fld):

  output: out := starting screen row for fld, zero-relative

  exceptions: exc := (fld is not a legal field identifier ⇒ `sg_badfld`)

`sg_g_col`(fld):

  ...

`sg_g_len`(`fld`):

  ...

`sg_g_val`(fld):

  ...

scngeom MIS—semantics

Identifier	Legal row values	Legal column values	Associated field in ISHAM RS
Variable fields			
`SG_MEM`	[0..9]	[0..9]	*MEM*
`SG_PC`	0	0	*PC*
`SG_ACC`	0	0	*ACC*
`SG_PRT`	0	0	*PRT*
`SG_MSG`	0	0	*MSG*
Fixed fields			
`SG_SCNTTL`	0	0	Screen title
`SG_MEMTTL1`	0	0	*MEM* title line 1
`SG_MEMTTL2`	0	0	*MEM* title line 2
`SG_MEMCOLHDR`	0	[0..9]	*MEM* column header
`SG_MEMROWHDR`	[0..9]	0	*MEM* row header
`SG_PCTTL`	0	0	*PC* title
`SG_ACCTTL`	0	0	*ACC* title
`SG_PRTTTL`	0	0	*PRT* title
`SG_PROMPTTTL`	0	0	Prompt title
`SG_MSGTTL`	0	0	Error message title

scngeom MIS—syntax

```
#define SG_NUMROW 24
#define SG_NUMCOL 80

typedef enum { SG_MEM,SG_PC,SG_ACC,SG_PRT,SG_MSG,
      ...
} sg_fldnam;

typedef struct {
      sg_fldnam nam;
      int row,col;
} sg_fld;
```

Routine names	Inputs	Outputs	Exceptions
sg_s_init			
sg_g_legfld	sg_fld	*boolean*	
sg_g_row	sg_fld	int	sg_badfld
sg_g_col	sg_fld	int	sg_badfld
sg_g_len	sg_fld	int	sg_badfld
sg_g_val	sg_fld	char*	sg_badfld

keybdin MIS—semantics (con.)

considerations

- Keystrokes are returned by `ki_sg_next` in first-in-first-out order.
- Characters are returned immediately, without waiting for a newline.
- If, on entry, there is no new keystroke available, `ki_sg_next` will not return until another keystroke occurs.

keybdin MIS—semantics

environment variables

assumptions

The *curses* module has been initialized.

Calls to *keybdin* obey the following pattern:
(`ki_s_init`.`ki_sg_next` $*$.`ki_s_end`)$*$,
where $X*$ indicates zero or more occurrences of X

access routine semantics

`ki_s_init`:
- **transition**: turn off keystroke echoing
- **exceptions**: none

`ki_sg_next`:
- **transition-output**: $out :=$ the next available character
- **exceptions**: none

`ki_s_end`:
- **transition**: turn on keystroke echoing
- **exceptions**: none

keybdin MIS—syntax

Routine names	Inputs	Outputs	Exceptions
`ki_s_init`			
`ki_sg_next`		`char`	
`ki_s_end`			

ISHAM Interface Specification

- *keybdin*
  - provides character-at-a-time keyboard access
- *scnstr*
  - writes a string at any screen location
- *scngeom*
  - provides positions and lengths for screen fields
- *scndr*
  - keeps the screen image up-to-date
  - relies heavily on *scngeom* and *scnstr*

Sufficiency—token Module

```
char s[TK_MAXSTRLEN+1];
tk_valtyp valtyp;

tk_s_init();
...
tk_s_str(s);
while (!tk_g_end()) {
      tk_sg_next(&valtyp);
      printf("val=%s typ=%d\n", valtyp.val,valtyp.typ);
}
```

Work Product Criteria (con.)

- Sufficient
  - provides required service
- Feasible
  - can be implemented, verified affordably
- High quality
  - no unnecessary violations of quality heuristics

Work Product Criteria (con.)

- Well-formed
  - obeys the MIS format
- Comprehensible
  - by the intended audience
- Complete
  - every call in every state covered by an assumption, exception, or normal case

Work Product Criteria

- Audience
  - designer, implementor, verifier, user
- Prerequisites
  - a reading knowledge of the MIS notations
- Purpose
  - specify the module interface

$sham$ Module—Services and Secret

- **Service**. Uses the other modules to provide the load-and-go assembler specified in the SHAM Requirements Specification.

- **Secret**. The way in which the other modules are used and the handling of command-line parameters.

exec MIS—Semantics (con.)

access routine semantics

`ex_s_init`:

transition:

if flag `ISHAM` is set then

initialize the screen

exceptions: none

`ex_s_exec`:

transition:

if flag `BSHAM` is set then

perform the exec. phase as in the BSHAM RS

else if flag `ISHAM` is set then

perform the exec. phase as in the ISHAM RS

In either case:

- Use mem, acc, and pc values stored in $absmach$
- Invoke `am_sg_next` to execute the next instruction
- Use `am_sg_next` return value to determine whether a normal case or exception output is needed

exceptions: none

exec MIS—Semantics

environment variables

scn

the terminal screen

stdout

Unix standard output

assumptions

Before `ex_s_exec` is called, `ex_s_init` is called and the *absmach* module has been initialized.

At compile-time, exactly one of these preprocessor flags is defined: `BSHAM`, `ISHAM`

exec MIS—Syntax

Routine names	Inputs	Outputs	Exceptions
`ex_s_init`			
`ex_s_exec`			

$exec$ Module—Service and Secret

- **Service**. Performs the execution phase, executing the program stored in $absmach$ and managing the run-time user interface, batch or interactive.

- **Secret**. The way in which the other modules are used, and the format and content of the exception messages.

$\boxed{load \text{ MIS—Semantics (con.)}}$

`ld_sg_load`(f): defined in terms of the SHAM RS.

transition/output:

(file f has no load errors $\Rightarrow$

$absmach.mem :=$ the object code for program f

$out :=$ `LD_NORMAL`

$|\ true \Rightarrow$

write the appropriate messages to $stdout$

$out :=$ `LD_ERROR`)

exceptions: $exc :=$ (error reading file $f \Rightarrow$ `ld_fil`)

load MIS Semantics

environment variables

stdout

Unix standard output

assumptions

`ld_s_init` is called before any other access routine.

The *absmach* and *token* modules have been initialized.

The argument to `ld_sg_load` points to an open FCB.

access routine semantics

`ld_s_init`:

...

load MIS—Syntax

```
typedef enum {LD_NORMAL,LD_ERROR} ld_stat;
```

Routine names	Inputs	Outputs	Exceptions
ld_s_init			
ld_sg_load	FILE*	ld_stat	ld_fil

$load$ Module—Service and Secret

- **Service**. Performs the load phase. Issues exception messages for incorrect input and, for correct input, stores the resulting object code in the $absmach$ module.

- **Secret**. The details of the load-phase user interface, including the source language concrete syntax and the exception messages.

$absmach$ MIS—Semantics (con.)

`am_sg_exec`:

transition-output:

(an error is specified in the Exec. Phase Exc. Table $\Rightarrow$

out := the error identifier

$|\ mem[pc] = \texttt{SY\_HALT} \Rightarrow out := \texttt{AM\_HALT}$

$|\ mem[pc] = \texttt{SY\_PRINT}) \Rightarrow$

$out, pc := \texttt{AM\_PRINT}, pc + 1$

$|\ true \Rightarrow\ out := \texttt{AM\_NORMAL}$

acc, pc, mem := values specified in the RS)

absmach MIS—Semantics (con.)

$\texttt{am\_s\_acc}(i)$:

transition: $acc := i$

exceptions: $exc := (i \notin [0..\texttt{AM\_MAXINT}] \Rightarrow \texttt{am\_int})$

$\texttt{am\_g\_acc}$:

output: $out := acc$

exceptions: none

$\texttt{am\_s\_pc}(a)$: $\cdots$

$\texttt{am\_g\_pc}$: $\cdots$

$\texttt{am\_s\_mem}(a, i)$: $\cdots$

$\texttt{am\_g\_mem}(a)$: $\cdots$

absmach MIS—Semantics

state variables

pc : [0..AM_MEMSIZ]

acc : [0..AM_MAXINT]

mem : *sequence* [AM_MEMSIZ] *of integer*

assumptions

am_s_init is called before any other access routine.

access routine semantics

am_s_init:

transition: $acc, pc, mem := 0, 0, 0$

exceptions: none

absmach MIS—Syntax (con.)

Routine names	Inputs	Outputs	Exceptions
am_s_init			
am_s_acc	int		am_int
am_g_acc		int	
am_s_pc	int		am_int
am_g_pc		int	
am_s_mem	int int		am_addr am_int
am_g_mem	int	int	am_addr
am_sg_exec		am_stat	

absmach MIS—Syntax

```
#define AM_MEMSIZ 100
#define AM_MAXINT 999

typedef enum {AM_NORMAL,AM_HALT,AM_PRINT,
      AM_ARITHEXC,AM_ADDREXC,
      AM_OBJECTEXC,AM_NOOPEXC
} am_stat;
```

absmach Module—Service and Secret

- **Service**. Implements the mem, acc, and pc state variables, as well as the Language Semantics Table from the SHAM Requirements Specification. Following each instruction execution, the user is given a status indicator and access to the state variables.

- **Secret**. The SHAM language semantics, including the execution-phase exceptions.

token MIS—Semantics (con.)

`tk_s_str`(s):

transition: $toklist := tokens(s)$

exceptions: $exc := (|s| > \texttt{TK\_MAXSTRLEN} \Rightarrow \texttt{tk\_maxlen})$

`tk_sg_next`:

transition/output: $toklist, out :=$

$toklist[1..|toklist - 1|]$,

$\langle toklist[0], toktyp(toklist[0])\rangle$

exceptions: $exc := (toklist = \langle\rangle \Rightarrow \texttt{tk\_end})$

`tk_g_end`:

output: $out := (toklist = \langle\rangle)$

exceptions: none

token MIS—Semantics (con.)

state variables

$toklist$: $sequence\ of\ string$

assumptions

`tk_s_init` is called before any other access routine.

All string parameters are legal C strings.

access routine semantics

`tk_s_init`:

transition: $toklist := \langle\rangle$

exceptions: none

$\boxed{token \text{ MIS—Semantics (con.)}}$

local functions

$tokens : string \rightarrow sequence\ of\ string$

$tokens(s)$ returns the sequence of tokens in s where

1. a token is a non-empty subsequence $s[i..j]$ of s
2. $s[i..j]$ contains no blanks
3. $(i = 0 \vee s[i-1] = \text{` '}) \wedge (j = |s| - 1 \vee s[j+1] = \text{` '})$

$toktyp : string \rightarrow \texttt{tk\_toktyp}$

$toktyp(s) :=$

$(s \in idtoksetT \Rightarrow \texttt{TK\_ID}$

$|\ s \in inttoksetT \Rightarrow \texttt{TK\_INT}$

$|\ true \Rightarrow \texttt{TK\_BADTOK})$

$\boxed{token \text{ MIS—Semantics}}$

local types

$idtoksetT =$

$\{s \mid s$ is a string of alphabetic or numeric characters $\wedge$
s[0] is alphabetic $\wedge$ $|s| \in [1..\texttt{TK\_MAXIDLEN}]\}$

$inttoksetT =$

$\{s \mid s$ is a string of numeric characters $\wedge$
$|s| \in [1..\texttt{TK\_MAXINTLEN}]\}$

token MIS—Syntax (con.)

Routine names	Inputs	Outputs	Exceptions
`tk_s_init`			
`tk_s_str`	`char*`		`tk_maxlen`
`tk_sg_next`		`tk_valtyp`	`tk_end`
`tk_g_end`		*boolean*	

token MIS—Syntax

```
#define TK_MAXSTRLEN 100
#define TK_MAXIDLEN 10
#define TK_MAXINTLEN 5

typedef enum {TK_ID,TK_INT,TK_BADTOK} tk_toktyp;

typedef struct {
      char val[TK_MAXSTRLEN+1];
      tk_toktyp typ;
} tk_valtyp;
```

token Module—Service and Secret

- **Service**. Extracts tokens from a string supplied by the user. Tokens are retrieved sequentially, in the order they occur in the user's string. The user is given access to the token value (a string) and the token type (integer, identifier, or unknown).

- **Secret**. The rules governing token types and token separators.

○ given: `"cat $* 123"`

○ needed ⟨⟨ `"cat"`, `ID` ⟩, ⟨ `"$*"` , `BADTOK` ⟩, ⟨ `"123"`,`INT` ⟩⟩

BSHAM Interface Specifications

- *token*
  - extracts tokens from a string
- *absmach*
  - stores mem, acc, pc; executes the current instruction
- *load*
  - performs the load phase
  - uses $token$; leaves object code in $absmach$
- *exec*
  - performs the execution phase
  - relies heavily on $absmach$
- *sham*
  - coordinator; invokes $load$, $exec$

Interaction with Other Modules

- Unspecified interaction
  - interaction used by implementor to provide service
  - no need for user to know
  - interaction should *not* be mentioned in MIS
  - example: $stack$

- Specified interaction: with module M
  - specified in terms of M's state or access routines
  - qualify module M's state vars with "M."
  - example: $load$

Interaction with the Environment

- Environment variables
  - new MIS section
  - name, type, interpretation for each variable

- Example: scn environment variable in $scnstr$
  - scn: $sequence$ [24][80] of $char$

    $scn[r][c]$ is the character at screen row r and column c, with numbering zero-relative and beginning at the upper-left corner

External Interaction

- Standalone modules
  - specified interaction is with caller only
  - must not interact with the environment
  - may call other modules; need not
  - examples: $stack$, $symtbl$, $token$, $absmach$

- Interaction with the environment
  - interface defined in terms of environment

- Interaction with other modules
  - interface defined in terms of other modules

Symtbl MIS—Semantics (con.)

$\texttt{st\_s\_loc}(sym, loc)$:

transition: $tbl := (tbl - \{\langle sym, loc_1\rangle\}) \cup \{\langle sym, loc\rangle\}$

where $\langle sym, loc_1\rangle \in tbl$

exceptions:

$exc := (\neg(\exists loc_1)(\langle sym, loc_1\rangle \in tbl) \Rightarrow \texttt{st\_notexsym})$

$\texttt{st\_g\_loc}(sym)$:

output: $out := loc$, where $\langle sym, loc\rangle \in tbl$

exceptions:

$exc := (\neg(\exists loc)(\langle sym, loc\rangle \in tbl) \Rightarrow \texttt{st\_notexsym})$

$\texttt{st\_g\_siz}$:

output: $out := |tbl|$

exceptions: none

Symtbl MIS—Semantics (con.)

`st_s_init`:

transition: $tbl := \{\}$

exceptions: none

`st_s_add`(sym, loc):

transition: $tbl := tbl \cup \{\langle sym, loc\rangle\}$

exceptions: $exc :=$

$(|sym| > \texttt{ST\_MAXSYMLEN} \Rightarrow \texttt{st\_maxlen}$

$|\ (\exists loc_1)(\langle sym, loc_1\rangle \in tbl) \Rightarrow \texttt{st\_exsym}$

$|\ |tbl| = \texttt{ST\_MAXSYMS} \Rightarrow \texttt{st\_full})$

`st_g_exsym`(sym):

output: $out := (\exists loc)(\langle sym, loc\rangle \in tbl)$

exceptions: none

Symtbl MIS—Semantics

state variables

$tbl : set\ of\ tuple\ of(sym : string, loc : integer)$

assumptions

`st_s_init` is called before any other access routine.

All string parameters are legal C strings.

Symtbl MIS—Syntax

```
#define ST_MAXSYMS 50
#define ST_MAXSYMLEN 20
```

Routine name	Input	Output	Exception
st_s_init			
st_s_add	char* int		st_maxlen st_exsym st_full
st_g_exsym	char*	*boolean*	
st_s_loc	char* int		st_notexsym
st_g_loc	char*	int	st_notexsym
st_g_siz		int	

Symtbl MIS—Overview

- Designed to support symbolic addresses in SHAM
- Module state
  - stores a set of symbol/location pairs
  - symbols must be unique; locations need not
- A Software-Decision Hiding module
  - hides underlying data structure and algorithm
- Remaining design steps
  - interface syntax, spec. trichotomy, normal case

A Module Interface Design Procedure

- Choose the state
  - often simple; takes time to get right
  - worthwhile specifying it precisely

- Access routine design
  - determine the interface syntax
  - establish the specification trichotomy
  - specify the normal case behavior

Interface Quality Criteria (con.)

- General
  - the ability to use a function for many purposes
  - user will do things never imagined by the designer
  - in the absence of knowledge, preserve freedom
- Minimal
  - keep independent features separate
  - many set-get calls violate minimality
- Opaque
  - the module secret is hidden
  - if likely change occurs, interface need not

Module Interface Quality Criteria

- Consistent
  - with partial knowledge ... the rest can be predicted
  - underlies all principles of quality

- Essential
  - do not provide unnecessary features
  - do not offer the same service in two ways

Module Interface Design Goals

- THE engineering goal: best product at lowest cost
- Minimize cost: development and maintenance
- Maximize run-time performance
- Maximize quality
  - interface quality largely ignored
  - hard to pin down, but worth trying

Access Routine Idioms—Tuple

Routine name	Input	Output	Exception
One set, get routine per field			
s_f_1	T_1		
g_f_1		T_1	
s_f_2	T_2		
g_f_2		T_2	
$\ldots$	$\ldots$	$\ldots$	$\ldots$
s_f_N	T_N		
g_f_N		T_N	
One set, get routine for entire tuple			
`s_val`	T		
`g_val`		T	

Access Routine Idioms—Sequence

Routine name	Inputs	Outputs	Exceptions
By absolute position			
s_add	int T		position full
s_del	int		position
s_val	int T		position
g_val	int	T	position
g_siz		int	
Sequential			
s_start			
sg_next		T	end
g_end		$boolean$	

Access Routine Idioms—Set

Routine name	**Input**	**Output**	**Exception**
`s_add`	T		`mem` `full`
`s_del`	T		`notmem`
`g_mem`	T	$boolean$	
`g_siz`		`int`	

Interface Design

- Design as a series of decisions
  - you must first see the available alternatives
  - you must then record your decisions

- Service and secret—the primary driving force
  - found in the Module Guide

- Access routine idioms
  - design from scratch is expensive, error-prone

- Quality heuristics
  - to suggest designs and guide reviews

Exception Signaling—*stack*

```
void ps_s_pop()
{
    if (siz == 0) {
        ps_empty();
        return;
    }
    --siz;
}
```

Exception Signaling

- MIS says when to signal an exception
  - need a way to inform module user
- Three principal approaches
  - idiomatic use of data
  - idiomatic use of control flow
  - builtin language constructs
- The SHAM approach
  - for each exception E, there is a C function called E
  - invoked by module implementation
  - supplied by module user

Stack MIS—Semantics (con.)

`ps_s_push`(x):

transition: $s := s \,\|\, \langle x \rangle$

exceptions: $exc := (|s| = \texttt{PS\_MAXSIZ} \Rightarrow \texttt{ps\_full})$

`ps_s_pop`:

transition: $s := s[0..|s|-2]$

exceptions: $exc := (|s| = 0 \Rightarrow \texttt{ps\_empty})$

`ps_g_top`:

output: $out := s[|s|-1]$

exceptions: $exc := (|s| = 0 \Rightarrow \texttt{ps\_empty})$

`ps_g_depth`:

output: $out := |s|$

exceptions: none

Stack MIS—Semantics

state variables

s : *sequence of integer*

assumptions

`ps_s_init` is called before any other access routine.

access routine semantics

`ps_s_init`:

transition: $s := \langle\rangle$

exceptions: none

Stack MIS—Syntax

```
#define PS_MAXSIZ 100
```

Routine name	Input	Output	Exception
ps_s_init			
ps_s_push	int		ps_full
ps_s_pop			ps_empty
ps_g_top		int	ps_empty
ps_g_depth		int	

Work Product Definition

- A *module interface* is the set of assumptions that
  - the user may make about module behavior
  - the developer may make about user behavior
- Interface syntax
  - access routine parameter, return value types
  - exported constants, types
- Interface semantics
  - based on MSMs
  - added: local functions, types, constants

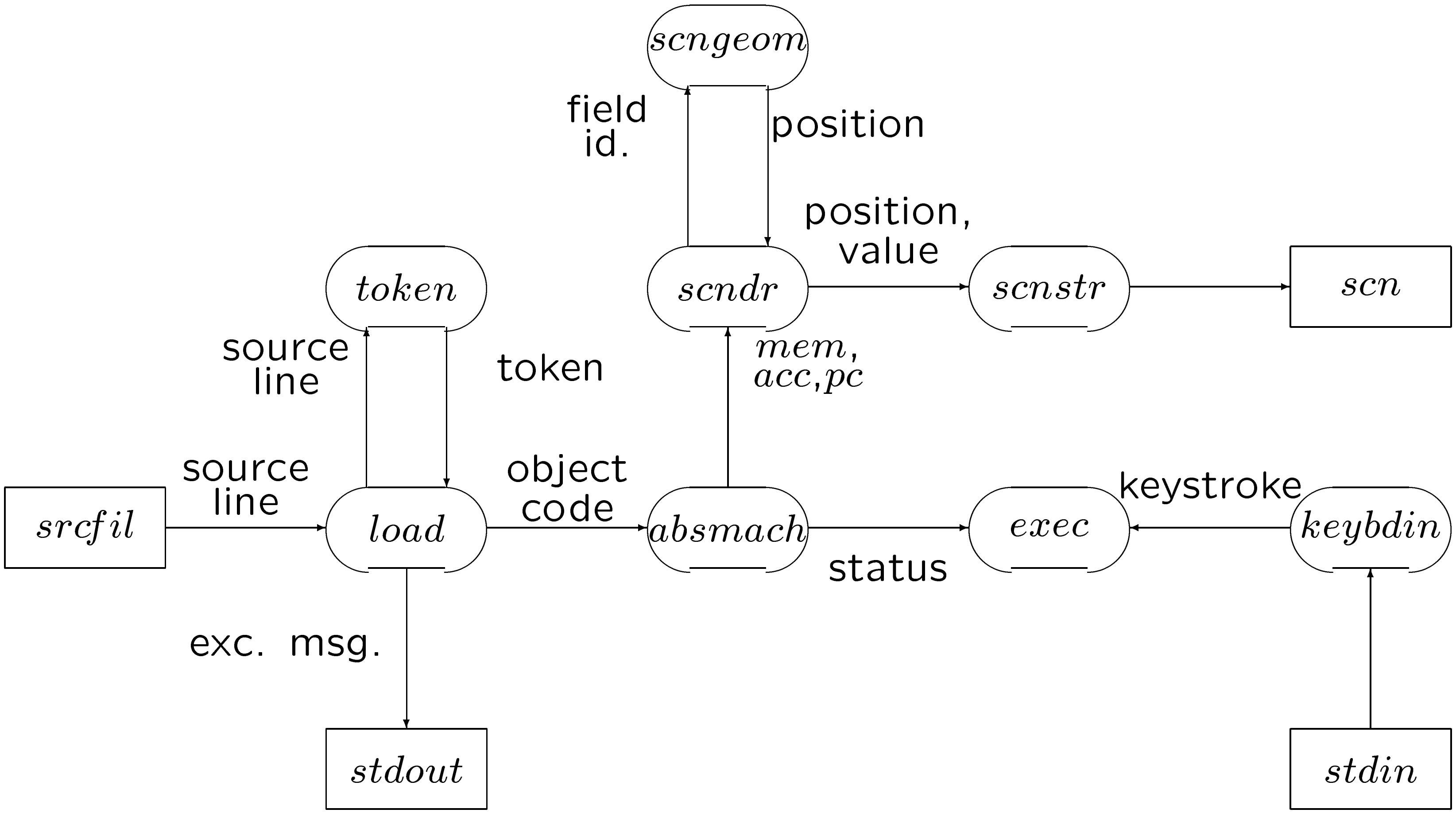
The Need for Precise MISs

- To support the four roles
  - designer, developer, verifier, user
- Why not use the source code as an MIS?
  - parallel implementation difficult
  - no record of implementor's assumptions
  - tester has no basis for correctness
  - complex implementation hard to use
  - source unavailable in proprietary systems
- Reliance on code as MIS causes massive problems
  - explicit MISs reduce these problems

Four MIS Roles

- Designer
  - determines M's behavior; records it in MIS
- Developer
  - creates implementation MI to meet MIS
- Verifier
  - determines if MI meets MIS
- User
  - reads MIS; writes programs using MI

ISHAM Dataflow Diagram



scndr Module—Service and Secret

- **Service**. Updates the terminal screen, using the values stored by *absmach* and according to the screen format described in the ISHAM Requirements Specification.

- **Secret**. The means used to accomplish screen updates.

scngeom Module—Service and Secret

- **Service**. Provides the length, row, and column position for each screen field, as per the screen format in the SHAM Requirements Specification.

- **Secret**. Hides, until execution time, the length, row, and column values.

scnstr Module—Service and Secret

- **Service**. Provides write access to the terminal screen. A string may be written to any position on the screen, the cursor may be moved to any position on the screen, and any screen position may be highlighted. To allow for efficient screen control, *scnstr* calls are buffered. An "apply changes to screen" access routine is provided; *scnstr* calls have no visible effect on the screen until the apply routine is invoked.

- **Secret**. The UNIX system services used to accomplish this task.

Modules for Output Variables

- scn
  - Machine hiding: $scnstr$
  - Output format: $scngeom$
  - Output driver: $scndr$
- $scnstr$ changes if the relevant curses functions do
- $scngeom$ changes if the location of the ACC field does
- $scndr$ changes if scrolling of MEM is introduced

keybdin Module—Service and Secret

- **Service**. Provides keyboard input, one character at a time, without echoing or waiting for carriage return.

- **Secret**. The UNIX system services used to accomplish this task.

Modules for Input Variables

○ $stdin$

- Machine hiding: $keybdin$
- Input format: none; syntax trivial

○ $keybdin$ changes if

- mechanism for turning off echoing changes

Summary

- Difficult design task
  - designer faces many alternatives
  - we have only guidelines to offer

- Information hiding guides the planning
  - suggests candidate decompositions
  - provides criteria for review of the decomposition
  - the MG supports the maintainer

- Successful IH depends on being able to predict changes
  - we cannot predict all changes
  - we plan for those we can

Verification Procedures

- Well-formed
  - faults easily revealed with line-by-line study
- Feasible
  - difficult to inspect effectively
  - design sketches useful
- Flexible
  - difficult to inspect effectively
  - imagine that the likely change has occurred
    show that the interface need not change

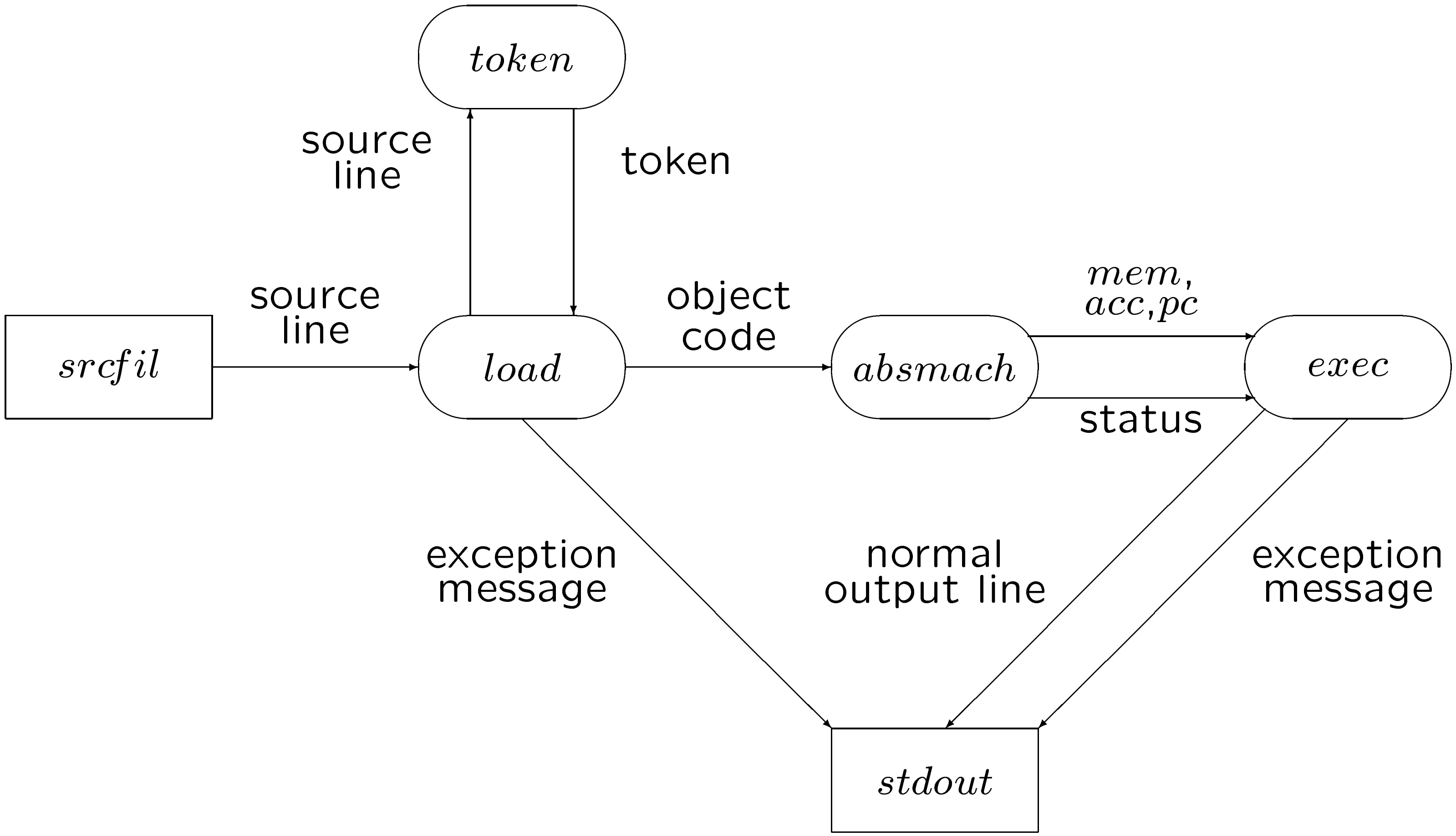
Work Product Criteria (con.)

- Well-formed
  - format & content obey the work product definition
- Feasible
  - a correct system can be implemented, with acceptable cost and performance
- Flexible
  - if a likely change is requested, the cost of the resulting modification will be reasonable

Work Product Criteria

- Audience
  - software designers, maintainers
- Prerequisites
  - a good understanding of the RS and of IH
- Purpose
  - describe and motivate the module decomposition

BSHAM Dataflow Diagram



sham Module—Service and Secret

- **Service**. Uses the other modules to provide the load-and-go assembler specified in the SHAM Requirements Specification.

- **Secret**. The way in which the other modules are used and the handling of command-line parameters.

$exec$ Module—Service and Secret

- **Service**. Performs the execution phase, executing the program stored in $absmach$ and managing the run-time user interface, batch or interactive.

- **Secret**. The way in which the other modules are used, and the format and content of the exception messages.

$absmach$ Module—Service and Secret

- **Service**. Implements the mem, acc, and pc state variables, as well as the Language Semantics Table from the SHAM Requirements Specification. Following each instruction execution, the user is given a status indicator and access to the state variables.

- **Secret**. The SHAM language semantics, including the execution-phase exceptions.

token Module—Service and Secret

- **Service**. Extracts tokens from a string supplied by the user. Tokens are retrieved sequentially, in the order they occur in the user's string. The user is given access to the token value (a string) and the token type (integer, identifier, or unknown).

- **Secret**. The rules governing token types and token separators.

load Module—Service and Secret

- **Service**. Performs the load phase. Issues exception messages for incorrect input and, for correct input, stores the resulting object code in the *absmach* module.

- **Secret**. The details of the load-phase user interface, including the source language concrete syntax and the exception messages.

Modules for State Variables

- mem, acc, pc
  - Software-decision hiding: $absmach$
  - State-driver: $load$, $absmach$

    $load$: initial values

    $absmach$: execution-phase transitions
- $load$ changes if even-word alignment required
- $absmach$ changes if index register added

Modules for Output Variables

- *stdout*
  - Machine-hiding: *stdio*
  - Output-format: none

- Output-driver: *load*, *exec*
  - *load*, *exec*: exception rules and messages
  - *exec*: normal case output

- *load*, *exec* change if messages changed to French

Modules for Input Variables

- *srcfil*
  - Machine-hiding: stdio
  - Input-format: *token*
  - Input-format: *load*
- *stdio* changes if underlying file I/O does
- *token* changes if tab accepted as token separator
- *load* changes if symbolic addresses added

SHAM Module Decomposition

- For now, focus is on BSHAM
  - with ISHAM needs accommodated
- First decomposition step: $load$, $exec$ modules
  - models RS load, exec phases
  - simple coordinator module also needed: $sham$
- This decomposition is
  - frequently used in translators
  - strongly suggested by the RS
  - not required by the RS

Work Product Definition

- Module summary
  - long name: descriptive name in English
  - short name: short mnemonic used for file names
  - prefix: short string used in exported C identifiers
- Module service and secret
  - service: brief description of features offered
  - secret: likely change encapsulated

Modules for each State Variable

- Software-decision hiding module
  - provides special-purpose memory, implemented in software
  - hides internal data structures and algorithms
- Behavior hiding state-driver module
  - maintains state using the SDH module
  - hides the rules that govern required state values

Modules for each Output Variable

- Machine-hiding module
  - accesses the device or OS service device used
  - hides the changeable characteristics of the device or OS service

- Output-format module
  - writes information to the Machine hiding module
  - hides the output format

- Output-driver module
  - writes information to output-format module
  - hides the rules that govern required output values

Modules for each Input Variable

- Machine-hiding module
  - accesses the device or OS service device used
  - hides the changeable characteristics of the device or OS service

- Behavior-hiding input-format module
  - processes raw input from machine-hiding module
  - hides the input format

Grouping Modules by Secret Type

- Machine-hiding (MH) module
  - hides characteristics of underlying machine
  - physical machine: hardware devices
  - virtual machine: OS services
- Software-decision hiding (SDH) module
  - hides internal data structures and algorithms
- Behavior-hiding (BH) module
  - hides behavior observable to end user
  - the behavior described in the RS

The Information Hiding Procedure

1. Identify the expected changes
   - begin with the **Expected Changes** in the RS

2. Create one module to encapsulate each change
   - we call that change the *module secret*

3. Design an interface which “hides” the secret
   - the interface need not change even if the expected change is made

Benefits of a Good Decomposition

- Shorter development time
  - through reduced interpersonal communication
  - through increased parallel development

- Verification simpler, more reliable
  - through separation of concerns

- Reduced maintenance cost
  - through greater product flexibility

- Critical: minimize module interdependence
  - ideally modules are completely independent

Module Decomposition

- Context
  - multi-person, multi-version programming
- Problem
  - programs too large to develop "all at once"
- Solution: module decomposition
  - division of the task into modules
- Module
  - a programming work assignment

Summary

- ISHAM RS has same format as BSHAM RS
  - same document sections
  - same focus on FSM

- Main differences
  - $stdin$ and scn environment variables
  - screen format
  - execution phase FSM handles
    - —keyboard input
    - —screen updates

Expected Changes

1. The field positions and the contents of the fixed fields will change.

2. $MEMSIZ$ will exceed 100 and vertical scrolling will be supported.

3. Different forms of stepping through the instructions will be supported, such as executing a specified number of instructions or executing until a specified instruction is reached.

Execution-phase FSM (con.)

Notes on screen updating:

- Initially and between transitions, ensure that:
  1. The fixed fields shown in the ISHAM screen format are displayed.
  2. MEM, PC, and ACC are such that
  $$(\forall r, c \in [0..9])(MEM[r, c] = mem[10 \times r + c]) \wedge$$
  $$ACC = acc \wedge PC = pc$$
  3. $MEM[pc/10, pc \; mod \; 10]$ is displayed in inverse video.

- Initially the MSG and PRT fields are blank

Execution-phase FSM (con.)

Transitions and outputs

For each character, c, from $stdin$
if $c = EXIT$ then
  clear scn
  halt ISHAM execution
else if $c = STEP$ then
  if the BSHAM Execution-phase Exception Table specifies
  an exception for $mem[pc]$ then
    MSG := the specified message
  else if $mem[pc] = HALT.object$ then
    $MSG := HALTMSG$
  else
    MSG := ""
    if $mem[pc] = PRINT.object$ then
      $PRT := acc$
    modify mem, acc, pc, as per the BSHAM Lang. Sem. Table
else
  $MSG := CMDERRMSG$

Execution-phase FSM

Inputs

Keystrokes from $stdin$.

Outputs

All outputs are to scn and its fields.

States

Same as for the load phase FSM.

Initial state

The final state for the load-phase FSM.

ISHAM RS—Screen Format

```
0123456789012345678901234567890123456789012345678901234567890123456...
*******************************************************************...
                                   SHAM


                 0     1     2     3     4     5     6    ...     9


Main        0   MEM   MEM   MEM   MEM   MEM   MEM   MEM   ...   MEM
memory:    10   MEM   MEM   MEM   MEM   MEM   MEM   MEM   ...   MEM
           20   MEM   MEM   MEM   MEM   MEM   MEM   MEM   ...   MEM
          ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
           90   MEM   MEM   MEM   MEM   MEM   MEM   MEM   ...   MEM

                           Program counter: PC-
                               Accumulator: ACC
                        Last value printed: PRT



Enter command: "s" to single step; "e" to exit
Message: MSG-------------------------------------------------------...
*******************************************************************...
```

Environment Variables

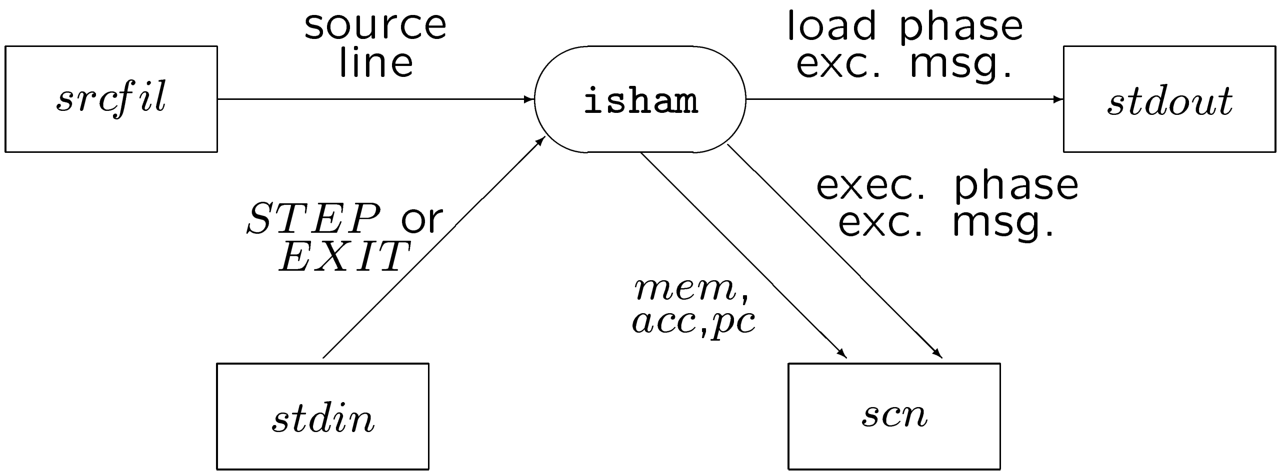
$stdin$: $string$

Unix standard input

scn: $sequence$[24][80] of $char$

$scn[r][c]$ is the character at screen row r and column c, with numbering zero-relative and beginning at the upper left corner

ISHAM RS—Overview



- New environment variables
  - for keyboard input and screen output
- New execution-phase FSM
  - single-stepped execution
  - screen updating

RS Chapter Summary

- Goals
  - precisely specify required behavior
  - support system users, developers, verifiers
- Work product format
  - standard formats and notations critical
  - over time, standardized approach pays off
  - real RSs much longer; the same techniques apply
- Verification
  - RS is verifiable in critical ways: completeness is key

Verification Procedure

- Well-formed
  - readily inspected in a checklist fashion
- Precise and comprehensible
  - difficult to inspect for directly
- Complete
  - assumptions, exceptions, normal case boundaries
- Feasible
  - estimating difficult; detailed RS very helpful

Work Product Criteria (con.)

- Well-formed
  - all required sections present
  - all terms, identifiers defined
- Precise and comprehensible
- Complete
  - with respect to the specification trichotomy
- Feasible
  - implementable at reasonable cost

Work Product Criteria

- Audience
  - system users or their representatives
  - system developers
- Prerequisites
  - users: task to be performed; user environment
  - developers: software technology; hardware/OS
- Purpose
  - describe the characteristics of the system that are required by the customer: no more, no less

BSHAM RS—Summary

- The BSHAM program offers a simple service
  - still, a complete RS is challenging to write
- The BSHAM RS is complicated
  - but the structure is simple
- Load-phase FSM is short and simple
  - details in Language Syntax and Exception tables
- Execution-phase FSM is short and simple
  - details in Language Semantics and Exception tables

Expected Changes

Input/output format

- command line parameters besides $srcfil$
- robust handling of overlong input lines

Abstract machine

- new or extended data types
- more registers, e.g., index registers
- symbolic data and branch addresses

Platform

- Different operating system: Unix or MS/DOS

Exception handling

- limits on the number of exceptions
- changes in the exception conditions or message text

Execution-Phase Exceptions—Example

Source Code	Object code		'After' values		
	Address	**Value**	pc	acc	$mem[5]$
`loadcon 500`	0	7	2	500	0
	1	500			
`store 5`	2	1	4	500	500
	3	5			
`br 0`	4	4	4	500	500
	5	0			

Execution-Phase Exception Table

Let $i = mem[pc]$ and $op = mem[pc + 1]$

Condition	Message
$i \in op0objectT$	**Normal case**
$i \in op1objectT$	
$\quad pc \in [0, MEMSIZ - 2]$	
$\quad\quad i = LOADCON.object$	**Normal case**
$\quad\quad i \neq LOADCON.object$	
$\quad\quad\quad op \in shamaddrT$	
$\quad\quad\quad\quad i = ADD.object$	
$\quad\quad\quad\quad\quad acc + mem[op] \in shamintegerT$	**Normal case**
$\quad\quad\quad\quad\quad acc + mem[op] \notin shamintegerT$	$excmsg(ARITHEXC, pc, "")$
$\quad\quad\quad\quad i = SUBTRACT.object$	
$\quad\quad\quad\quad\quad acc - mem[op] \in shamintegerT$	**Normal case**
$\quad\quad\quad\quad\quad acc - mem[op] \notin shamintegerT$	$excmsg(ARITHEXC, pc, "")$
$\quad\quad\quad\quad true$	**Normal case**
$\quad\quad\quad op \notin shamaddrT$	$excmsg(ADDREXC, pc, op)$
$\quad pc = MEMSIZ - 1$	$excmsg(NOOPEXC, pc, "")$
$i \notin objectT$	$excmsg(OBJECTEXC, pc, i)$

Object Code Types

$$objectT = \{LOAD.object, STORE.object, ADD.object,$$
$$SUBTRACT.object,$$
$$BRANCH.object, BRANCHZERO.object,$$
$$BRANCHPOS.object,$$
$$LOADCON.object, PRINT.object, HALT.object\}$$

$$op0objectT = \{HALT.object, PRINT.object\}$$

$$op1objectT = objectT - op0objectT$$

The 2+2 Program

srcfil	Object code Address	Value	'After' values *pc*	*acc*	*mem*[8]	*stdout*
`loadcon 2`	0	7	2	2	0	$\langle\rangle$
	1	2				
`store 8`	2	1	4	2	2	$\langle\rangle$
	3	8				
`add 8`	4	2	6	4	2	$\langle\rangle$
	5	8				
`print`	6	8	7	4	2	$\langle 4\rangle$
`halt`	7	9	7	4	2	$\langle 4\rangle$

Execution-Phase FSM (con.)

Transitions and output

for the instruction beginning at $mem[pc]$:

if the E. E. Table specifies an exception then

write the specified message to $stdout$

terminate SHAM

else if $mem[pc] = HALT.object$ then

terminate SHAM

else

if $mem[pc] = PRINT.object$ then

write to $stdout$: $acc \,||$ newline

modify mem, acc, and pc as shown in the L. S. Table

Execution-Phase FSM

Inputs

None

Outputs

Normal-case output and exception messages are written to $stdout$

States

Same as for the load phase FSM

Initial state

The final state for the load phase FSM

Language Semantics Table

Let $op = mem[pc+1]$

Instr. at $mem[pc]$	**Effect on mem, acc, and pc**
	Memory access
$LOAD.object$	$acc, pc := mem[op], (pc+2)$ mod $MEMSIZ$
$STORE.object$	$mem[op], pc := acc, (pc+2)$ mod $MEMSIZ$
	Arithmetic
$ADD.object$	$acc, pc := acc + mem[op], (pc+2)$ mod $MEMSIZ$
$SUBTRACT.object$	$acc, pc := acc - mem[op], (pc+2)$ mod $MEMSIZ$
	Branch
$BRANCH.object$	$pc := op$
$BRANCHZERO.object$	$pc := (acc = 0 \Rightarrow op$ $\mid acc > 0 \Rightarrow (pc+2)$ mod $MEMSIZ)$
$BRANCHPOS.object$	$pc := (acc > 0 \Rightarrow op$ $\mid acc = 0 \Rightarrow (pc+2)$ mod $MEMSIZ)$
	Miscellaneous
$LOADCON.object$	$acc, pc := op, (pc+2)$ mod $MEMSIZ$
$PRINT.object$	$pc := (pc+1)$ mod $MEMSIZ$
$HALT.object$	no change to acc, pc, mem

Load-Phase Exceptions—Example

srcfil	**Exception message**
`br 115`	`Load exception at 1. Illegal operand: 115`
`lode 7`	`Load exception at 2. Illegal instruction: lode`
`add`	`Load exception at 3. Operand missing`
`add 5 8`	

Load-Phase Exception Table

Let L be the current line, with line number n
Let $T_1, T_2, \ldots, T_K$ be the tokens in L

Condition	Message
$K = 0$ (L is blank)	$excmsg(BLANKLINEEXC, n, \texttt{""})$
$K > 0$	
$T_1 \in op0sourceT$	**Normal case**
$T_1 \in op1sourceT$	
$K = 1$	$excmsg(MISSINGOPEXC, n, \texttt{""})$
$K > 1$	
$T_1 = LOADCON.source$	
$T_2 \in shamintegerT$	**Normal case**
$T_2 \notin shamintegerT$	$excmsg(OPFMTEXC, n, T_2)$
$T_1 \neq LOADCON.source$	
$T_2 \in shamaddrT$	**Normal case**
$T_2 \notin shamaddrT$	$excmsg(OPFMTEXC, n, T_2)$
$T_1 \notin sourceT$	$excmsg(SOURCEEXC, n, T_1)$

Source Code Types

$$
\begin{aligned}
sourceT = \{&LOAD.source, STORE.source, ADD.source,\\
&SUBTRACT.source,\\
&BRANCH.source, BRANCHZERO.source,\\
&BRANCHPOS.source,\\
&LOADCON.source, PRINT.source, HALT.source\}
\end{aligned}
$$

$$op\mathsf{0}sourceT = \{HALT.source, PRINT.source\}$$

$$op\mathsf{1}sourceT = sourceT - op\mathsf{0}sourceT$$

The 2+2 Program

srcfil	Object code Address	Value	'After' values pc	acc	$mem[8]$	$stdout$
`loadcon 2`	0	7	2	2	0	$\langle\rangle$
	1	2				
`store 8`	2	1	4	2	2	$\langle\rangle$
	3	8				
`add 8`	4	2	6	4	2	$\langle\rangle$
	5	8				
`print`	6	8	7	4	2	$\langle 4\rangle$
`halt`	7	9	7	4	2	$\langle 4\rangle$

Load Phase FSM (con.)

Transitions and outputs

For line L, with line number n:

if the Load-phase Exc. Table specifies an exception then

write the specified message to $stdout$

else

if no previous line had an exception then

if there is room in mem then

load the object-code form of L into mem

else

write $excmsg(NOMEMEXC, n, "")$ to $stdout$

Load Phase FSM

Inputs

Each input is a line from $srcfil$,
read in the order it appears in $srcfil$

Outputs

Normal-case output and exception messages are written to $stdout$

States

mem : $sequence$ [0..$MEMSIZ - 1$] of $shamintegerT$

acc : $shamintegerT$

pc : $shamaddrT$

Initial state

acc, pc, and every element of mem are set to 0

Language Syntax Table

Mnemonic	$I.source$	$I.object$	Operand type
		Memory access	
$LOAD$	`load`	0	$shamaddrT$
$STORE$	`store`	1	$shamaddrT$
		Arithmetic	
ADD	`add`	2	$shamaddrT$
$SUBTRACT$	`sub`	3	$shamaddrT$
		Branch	
$BRANCH$	`br`	4	$shamaddrT$
$BRANCHZERO$	`brz`	5	$shamaddrT$
$BRANCHPOS$	`brp`	6	$shamaddrT$
		Miscellaneous	
$LOADCON$	`loadcon`	7	$shamintegerT$
$PRINT$	`print`	8	
$HALT$	`halt`	9	

Types and Constants

- Constants

Name	**Value**
$MAXLINLEN$	100
$MAXINT$	999
$MEMSIZ$	100

- Types

$$shamaddrT = [0, MEMSIZ - 1]$$

$$shamintegerT = [0, MAXINT]$$

Exception Messages

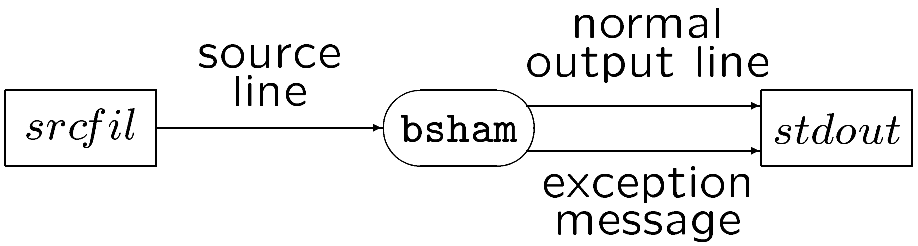
$excmsg : excidT \times integer \times string \to string$

if id **is**	**then** $excmsg(id, loc, tok)$ **is**
Command line messages	
$FILSYSEXC$	`Command line error. Cannot open file:` tok
$NOFILEXC$	`Command line error. No file name specified`
Load-phase messages	
$BLANKLINEEXC$	`Load exception at` loc`. Blank line illegal`
$MISSINGOPEXC$	`Load exception at` loc`. Operand missing`
$NOMEMEXC$	`Load exception at` loc`. Program too large`
$OPFMTEXC$	`Load exception at` loc`. Illegal operand:` tok
...	...
Execution-phase messages	
...	...

Exception Identifiers

$$
\begin{aligned}
excidT = \{&FILSYSEXC, NOFILEXC,\\
&BLANKLINEXC, MISSINGOPEXC,\\
&\quad NOMEMEXC, OPFMTEXC, SOURCEEXC,\\
&ADDREXC, ARITHEXC, NOOPEXC,\\
&\quad OBJECTEXC\}
\end{aligned}
$$

BSHAM RS—Overview



- Command line invocation

  % **bsham** $srcfil$

- Environment variables

  $srcfil : string$

  The file name passed on the command line

  $stdout : string$

  Unix stdout

The State Machine Section

- Inputs
  - system inputs, in terms of environment variables
- Outputs
  - system outputs, in terms of environment variables
- State variables
  - state space expressed as variable declarations
- Initial state
  - an initial value for each state variable
- Transitions and outputs
  - outputs and new state for each input and old state

Document Sections

- Overview
  - of the system and the RS document
- Environment variables
  - declarations of variables modeling the environment
- State machine
  - definition of the FSM on which the RS is based
- Constants, Types, Functions
  - auxiliary constants, types, and functions

Requirements Specification Goals

- Serve as a contract between users and developers
- Ensure that developers do not decide users need
- Provide support for independent verification
- Support estimates of time and resources
- Provide protection against personnel turnover
- Support the maintainer

Four Requirements Specification Roles

- Designer
  - decides on the required system behavior
  - records it in the RS
- Developer
  - develops a system as specified by the RS
- Verifier
  - shows that the system is correct w.r.t. the RS
- User—the end user
  - depends on behavior as specified in the RS

Four Specification Roles

- Designer
  - decides on the required behavior
  - records it in specification S
- Developer
  - creates an implementation I as specified by S
- Verifier
  - shows that I is correct with respect to S
- User
  - reads S to learn how to use I

SHAM Work Products

- ISHAM and BSHAM Requirements Specifications (RS)
- Module Guide (MG)
- For each of the 9 modules
  - Module Interface Specification (MIS)
  - Module Internal Design (MID)
  - Module Implementation (MI)
  - Test Plan (TP) and Test Implementation (TI)
- System Test Plan (TP) and Test Implementation (TI)

SHAM Instruction Set

Instruction	**Operand**	**Description**
LOAD	a	load value at a into acc
STORE	a	store acc at a
ADD	a	add value at a to acc
SUBTRACT	a	subtract value at a from acc
BRANCH	a	branch to a
BRANCHZERO	a	branch to a if $acc = 0$
BRANCHPOS	a	branch to a if $acc > 0$
LOADCON	i	load integer i into acc
PRINT		print value of acc
HALT		terminate SHAM

Service Overview

- Register and memory model
  - registers: program counter and accumulator
  - one-dimensional memory; absolute addressing
- Instruction format
  - one word instruction code; one word operand
- Instruction set
  - only ten instructions in all
- User interface
  - batch and interactive interface

Purpose of the SHAM System

- As a teaching vehicle
  - complex enough to demonstrate the methods
  - simple enough to learn easily
  - somewhat entertaining

- Commercial application
  - none: it is truly a sham

- Hypothetical application
  - teach basics of machine language programming
  - provide the simplest services imaginable

$elist$ MSM—Execution Tables

	Call	New state	Output	Exception
(1)	`s_init`	$\langle\rangle$	—	—
(2)	`s_add`$(0, 10)$	$\langle 10\rangle$	—	—
(3)	`s_add`$(1, 20)$	$\langle 10, 20\rangle$	—	—
(4)	`s_add`$(2, 10)$	$\langle 10, 20, 10\rangle$	—	—
(5)	`g_pos`(10)	$\langle 10, 20, 10\rangle$	0	—

	Call	New state	Output	Exception
(1)	`s_init`	$\langle\rangle$	—	—
(2)	`s_add`$(0, 10)$	$\langle 10\rangle$	—	—
(3)	`s_add`$(1, 20)$	$\langle 10, 20\rangle$	—	—
(4)	`s_add`$(2, 10)$	$\langle 10, 20, 10\rangle$	—	—
(5)	`s_delval`(10)	$\langle 10, 20\rangle$	—	—

elist MSM—Execution Tables ($N = 3$)

	Call	New state	Output	Exception
(1)	`s_init`	$\langle\rangle$	—	—
(2)	`s_add`(0, 10)	$\langle 10 \rangle$	—	—
(3)	`s_add`(1, 20)	$\langle 10, 20 \rangle$	—	—
(4)	`s_add`(2, 30)	$\langle 10, 20, 30 \rangle$	—	—
(5)	`s_add`(3, 40)	$\langle 10, 20, 30 \rangle$	—	`full`

	Call	New state	Output	Exception
(1)	`s_init`	$\langle\rangle$	—	—
(2)	`s_add`(0, 10)	$\langle 10 \rangle$	—	—
(3)	`s_add`(1, 20)	$\langle 10, 20 \rangle$	—	—
(4)	`g_val`(2)	$\langle 10, 20 \rangle$	0	`pos`

$\boxed{elist \text{ MSM—Semantics}}$

$\texttt{s\_delval}(x)$:

transition: $s := s[0..i-1] \parallel s[i+1..|s|-1]$ where $pre(s)[i] = x$

exceptions: $exc := (x \notin s \Rightarrow \texttt{val})$

$\texttt{sg\_val}$:

transition-output: $s, out :=$

$s[0..i-1] \parallel s[i+1..|s|-1], s[i]$

where $i \in [0..|pre(s)|-1]$

exceptions: $exc := (|s| = 0 \Rightarrow \texttt{empty})$

$elist$ MSM—Semantics

access routine semantics

$\texttt{g\_exval}(x)$:

output: $out := x \in s$

exceptions: none

$\texttt{g\_pos}(x)$:

output: $out := i$ where $(i \in [0..|s|-1] \wedge s[i] = x)$

exceptions: $exc := (x \notin s \Rightarrow \texttt{val})$

elist MSM—Access Routines

Routine name	Input	Output	Exception
g_exval	*integer*	*boolean*	
g_pos	*integer*	*integer*	val
s_delval	*integer*		val
sg_val		*integer*	empty

Summary Table

	Normal case	**Exceptions**
set call	one or more new states	zero or one exception(s) no state change
get call	one or more outputs	zero or one exception(s) any output permitted
set-get call	one or more new state output pairs	zero or one exception(s) no state change; any output permitted

Examining MSMs More Carefully

- What variations are permitted on MSMs?
- Nondeterminism
  - is there more than one acceptable new state?
  - is there more than one acceptable output?
  - is there more than one acceptable exception?
- Exception Semantics
  - when an exception occurs, are there any constraints on the new state or output?
- Completeness
  - are there traces for which there is no behavior specified?

Transitions and Outputs

- Set routines
  - transition

    change in state caused by the call

    specified by assignment to state variables
  - exceptions

    conditions under which exceptions are signaled

    specified by assignment to exc or "none"
- Get routines
  - output

    value returned by the call

    specified by assignment to out
  - exceptions—same as for set routines

list MSM—Execution Tables ($N = 3$)

	Call	New state	Output	Exception
(1)	`s_init`	$\langle\rangle$	—	—
(2)	`s_add`$(0, 10)$	$\langle 10\rangle$	—	—
(3)	`s_add`$(1, 20)$	$\langle 10, 20\rangle$	—	—
(4)	`s_add`$(2, 30)$	$\langle 10, 20, 30\rangle$	—	—
(5)	`g_val`(2)	$\langle 10, 20, 30\rangle$	30	—

	Call	New state	Output	Exception
(1)	`s_init`	$\langle\rangle$	—	—
(2)	`s_add`$(0, 10)$	$\langle 10\rangle$	—	—
(3)	`s_add`$(0, 20)$	$\langle 20, 10\rangle$	—	—
(4)	`s_add`$(0, 30)$	$\langle 30, 20, 10\rangle$	—	—
(5)	`g_val`(2)	$\langle 30, 20, 10\rangle$	10	—

$\boxed{list \text{ MSM—Semantics}}$

$\texttt{s\_del}(i)$:

transition: $s := s[0..i-1] \parallel s[i+1..|s|-1]$

exceptions: $exc := (i \notin [0..|s|-1] \Rightarrow \texttt{pos})$

$\texttt{s\_val}(i, x)$:

transition: $s[i] := x$

exceptions: $exc := (i \notin [0..|s|-1] \Rightarrow \texttt{pos})$

$\texttt{g\_val}(i)$:

output: $out := s[i]$

exceptions: $exc := (i \notin [0..|s|-1] \Rightarrow \texttt{pos})$

$list$ MSM—Semantics

state variables

s: $sequence\ of\ integer$

access routine semantics

`s_init`:

transition: $s := \langle\rangle$

exceptions: none

`s_add`(i, x):

transition: $s := s[0..i-1] \parallel \langle x \rangle \parallel s[i..|s|-1]$

exceptions: $exc := (|s| = N \Rightarrow \texttt{full} \mid i \notin [0..|s|] \Rightarrow \texttt{pos})$

list MSM—Access Routines

Routine name	Input	Output	Exception
`s_init`			
`s_add`	*integer* *integer*		`full` `pos`
`s_del`	*integer*		`pos`
`s_val`	*integer* *integer*		`pos`
`g_val`	*integer*	*integer*	`pos`

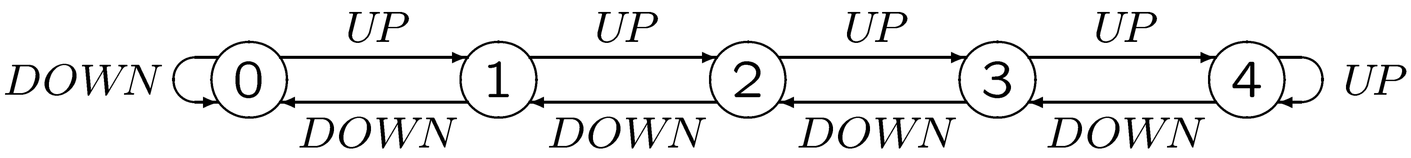
Module State Machines

- A special kind of finite state machine
  - for specifying the behavior of software modules
- Inputs: access routine calls
- States: derived from variable declarations
- Initial state: the one produced by `s_init`
  - every module must provide `s_init`
- Transitions: specified by assignments to state variables
- Event outputs only
  - normal case: access routine return values
  - exception: names of exceptions signaled

Summary

- Every FSM has
  - a set of inputs, a set of states, and an initial state
  - a state-transition function
  - event outputs, condition outputs, or both
- Event outputs model function output well
- Condition outputs model screen output well
- Nondeterminism may be required
  - more than one transition or output is allowed
  - transition and output relations are required

Up/Down Counter—Outputs



Condition	$E(s, x)$
$s = 0$	
$\quad x = DOWN$	$ALARM$
$\quad x = UP$	$NORMAL$
$s \in [1..3]$	$NORMAL$
$s = 4$	
$\quad x = DOWN$	$NORMAL$
$\quad x = UP$	$ALARM$

Condition	$C(s)$
$s = 0$	$WARNING$
$s \in [1..3]$	$NORMAL$
$s = 4$	$WARNING$

Event and Condition Outputs

◯ Event Outputs

- outputs are associated with transitions
- O_E : set of event outputs
- $E: \; S \times I \to O_E$
- $E(s, x) =$ output associated with transition $\langle s, x \rangle$

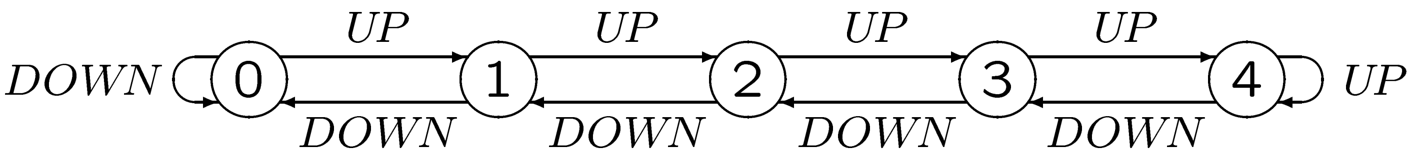
◯ Condition Outputs

- outputs are associated with states
- O_C : set of condition outputs
- $C: \; S \to O_C$
- $C(s) =$ output associated with state s

Outputs

- FSMs are often used for software specification
- Only the input/output behavior is important
- For each input sequence the software and the FSM must have the same output
- The states and transitions need not be the same

Up/Down Counter



- $S:\ \{0, 1, 2, 3, 4\}$ $\quad s_0 = 0 \quad$ $I:\ \{UP, DOWN\}$
- $T:\ S \times I \to S$

Condition	$T(s, x)$
$x = DOWN$	
$s = 0$	0
$s \in [1..4]$	$s - 1$
$x = UP$	
$s \in [0..3]$	$s + 1$
$s = 4$	4

States and Transitions

- S : set of states
- s_0 : initial state
- I : set of inputs
- $T: \; S \times I \to S$
  - $T(s, x) =$ new state for input x while in state s

Finite State Machines (FSM)

- FSMs are powerful constructs for software specification

- Basic components of a FSM:
  - a set of inputs, a set of states, and an initial state
  - a state-transition function
  - a set of outputs and an output function

- A variety of inputs is required
  - to model keyboard input, function calls, etc.

- A variety of outputs is required
  - to model screen output, function return values, etc.

Tabular Form—Nested Rules

- For nested rules, use indenting in the condition column
- $\langle x_0, y_0 \rangle < \langle x_1, y_1 \rangle =$

$$
\begin{aligned}
&(x_0 < x_1 \Rightarrow \mathit{true} \\
&\mid x_0 = x_1 \Rightarrow \\
&\qquad (y_0 < y_1 \Rightarrow \mathit{true} \mid y_0 \geq y_1 \Rightarrow \mathit{false}) \\
&\mid x_0 > x_1 \Rightarrow \mathit{false})
\end{aligned}
$$

Condition	$\langle x_0, y_0 \rangle < \langle x_1, y_1 \rangle$
$x_0 < x_1$	true
$x_0 = x_1$	
$\quad y_0 < y_1$	true
$\quad y_0 \geq y_1$	false
$x_0 > x_1$	false

Tabular Form—Examples

- $two\_sort = (x < y \Rightarrow x, y := x, y \mid x \geq y \Rightarrow x, y := y, x)$

Condition	$x :=$	$y :=$
$x < y$	x	y
$x \geq y$	y	x

Conditional Rules—Tabular Form

- Tabular form is clearer for long conditional rules
  - use one row for each component
  - place conditions in column one
  - place rules in columns two, three , ...

- $min(x, y) = (x \leq y \Rightarrow x \mid x > y \Rightarrow y)$

Condition	$min(x, y)$
$x \leq y$	x
$x > y$	y

Conditional Rules—Examples

- Simple rules
  - $min(x, y) = (x \leq y \Rightarrow x \mid x > y \Rightarrow y)$
- Rules with embedded assignment—two-sort
  - $(x \leq y \Rightarrow x, y := x, y \mid x > y \Rightarrow x, y := y, x)$
- Nested rule—lexicographic order
  - $\langle x_0, y_0 \rangle < \langle x_1, y_1 \rangle =$

$$
\begin{aligned}
&(x_0 < x_1 \Rightarrow true \\
&\mid x_0 = x_1 \Rightarrow \\
&\qquad (y_0 < y_1 \Rightarrow true \mid y_0 \geq y_1 \Rightarrow false) \\
&\mid x_0 > x_1 \Rightarrow false)
\end{aligned}
$$

Conditional Rules

- Syntax
  - $(c_1 \Rightarrow r_1 \mid c_2 \Rightarrow r_2 \mid \ldots \mid c_n \Rightarrow r_n)$
  - c_i: condition—any boolean expression
  - r_i: rule—any expression

- Semantics
  - evaluate c_1; if true, r_1 applies
  - otherwise, evaluate c_2; if true, r_2 applies

    $\ldots$
  - otherwise, evaluate c_n; if true, r_n applies
  - otherwise

    $(c_1 \Rightarrow r_1 \mid c_2 \Rightarrow r_2 \mid \ldots \mid c_n \Rightarrow r_n)$ is undefined

The Underlying Function

- A function from old state to new state
- Let $x, y, z : integer$
- $x, y := 0, 10$
  - $\{\langle\langle x_1, y_1, z_1\rangle, \langle x_2, y_2, z_2\rangle\rangle \mid x_2 = 0 \land y_2 = 10 \land z_2 = z_1\}$
- $x, y := 10, x$
  - $\{\langle\langle x_1, y_1, z_1\rangle, \langle x_2, y_2, z_2\rangle\rangle \mid x_2 = 10 \land y_2 = x_1 \land z_2 = z_1\}$
- $x, y := y, x$
  - $\{\langle\langle x_1, y_1, z_1\rangle, \langle x_2, y_2, z_2\rangle\rangle \mid x_2 = y_1 \land y_2 = x_1 \land z_2 = z_1\}$

Multiple Assignment—Examples

Assignment	Before x	Before y	After x	After y
$x, y := 0, 10$	1	2	0	10
	4	7	0	10
$x, y := 10, x$	1	2	10	1
	4	7	10	4
$x, y := y, x$	1	2	2	1
	4	7	7	4

The Multiple Assignment Statement

- Syntax
  - $v_1, v_2, ..., v_n := e_1, e_2, ..., e_n$
  - the v_i's are distinct variables
  - each e_i is an expression of the same type as v_i

- Semantics
  - evaluate $e_1, e_2, .., e_n$
  - assign e_i's value to v_i
  - leave variables other than $v_1, v_2, ..., v_n$ unchanged

Types—More Examples

- Let $T_0 = \mathit{tuple\ of}\ (\mathit{id}\colon\ \mathit{integer},$
  $\mathit{val}\colon\ \mathit{string})$

- The following are equivalent definitions
  - $T_1 = \mathit{set\ of}\ T_0$
  - $T_1 = \mathit{set\ of\ tuple\ of}\ (\mathit{id}\colon\ \mathit{integer},$
    $\mathit{val}\colon\ \mathit{string})$

- Let $x\colon\ T_1$; possible values for x
  - $\{\}$
  - $\{\langle 1, \texttt{"cat"}\rangle, \langle 3, \texttt{"dog"}\rangle, \langle 1, \texttt{"mouse"}\rangle\}$

Tuple Operations

Operation	**Meaning**
$=, \neq$	equal, not equal
$t.f$	value of field f of tuple t

- Let $T = tuple\ of\ (id\colon\ integer,$
  $val\colon\ string)$
- $x\colon\ T$
- Let $x = \langle 4, \texttt{"cat"}\rangle$
  - $x.id = 4$
  - $x.val = \texttt{"cat"}$

Tuples

Definition: A *tuple* is a collection of elements of possibly different types. Each tuple has one or more fields, each with a unique field identifier.

- Tuple definition
  - $$\begin{aligned} T = \mathit{tuple\ of}\ (&f_1 : T_1, \\ &f_2 : T_2, \\ &\ldots \\ &f_n : T_n) \end{aligned}$$
- Specifying a particular tuple
  - $\langle x_1, x_2, \ldots, x_n \rangle$

Undefined Sequence Operations

- Let s: *sequence* [100] *of integer*

- Undefined operations
  - $s[-1], s[100], s[1000], s[90, 110]$

- $s[i]$ is undefined if $i \notin [0..|s| - 1]$

- $s[i..j]$ is undefined if $i \notin [0..|s| - 1] \vee j \notin [0..|s| - 1]$

- Express in logic "the elements of s are nonnegative"
  - flawed: $(\forall i \in integer)(s[i] \geq 0)$
  - satisfactory: $(\forall i \in [0..99])(s[i] \geq 0)$

Sequence Operations

Operation	**Meaning**
$s[i..j]$	subsequence of s from i to j
$s_1 \parallel s_2$	concatenation of s_1 and s_2
$=, \neq$	equal, not equal
$\in, \notin$	member, non-member
$s[i]$	i-th element of s
$\|s\|$	length of s

- Let $a = \langle 1, 2, 3, 4 \rangle$ and $b = \langle 5, 6 \rangle$

$a[0..3] = \langle 1, 2, 3, 4 \rangle$ $a[3..3] = \langle 4 \rangle$ $a[3..1] = \langle \rangle$
$a \parallel b = \langle 1, 2, 3, 4, 5, 6 \rangle$ $b \parallel a = \langle 5, 6, 1, 2, 3, 4 \rangle$
$3 \in x$ $3 \notin y$
$a[0] = 1$ $a[3] = 4$
$|a| = 4$ $|a[1..2]| = 2$ $|b| = 2$

Sequence Definition—Examples

- Let $T_1 = \textit{sequence of } \{red, green, blue\}$ and x_1: T_1
  - $T_1 = \{\langle\rangle, \langle red\rangle, \langle green\rangle, \langle blue\rangle,$
    $\langle red, red\rangle, \langle red, green\rangle, \langle red, blue\rangle,$
    $\langle green, red\rangle, \langle green, green\rangle, \langle green, blue\rangle, \ldots$
    $\langle red, red, red, red, green, green, blue\rangle, \ldots$
  - $\langle red, blue, blue\rangle$ is a possible value for x_1

- Let $T_2 = \textit{sequence of integer}$ and x_2: T_2
  - T_2 is the set of all sequences of integers
  - $\langle 1, 3, 1, 7, 9\rangle$ is a possible value for x_2

Sequences

Definition: A *sequence* is an ordered collection of elements of the same type.

- Variable length sequence definition
  - $T = \mathit{sequence\ of}\ T_0$
- Fixed length sequence definition
  - $T = \mathit{sequence}\ [i_0, i_1, \ldots, i_{n-1}]\ \mathit{of}\ T_0$
- Specifying a particular sequence
  - $\langle x_0, x_1, \ldots, x_{n-1} \rangle$
- Indexing is zero-relative

Set Operations

Operation	Meaning
$\cup, \cap, -, \times$	union, intersec., diff., Cartesian product
$=, \neq$	equal, not equal
$\in, \notin$	member, non-member
$\|s\|$	size of set s

- Let $a = \{1, 2, 3\}$ and $b = \{2, 3\}$

$a \cup b = b \cup a = \{1, 2, 3\}$
$a \cap b = b \cap a = \{2, 3\}$
$a - b = \{1\}$ $\qquad b - a = \{\}$
$a \times b = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 2\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle, \langle 3, 3\rangle\}$
$1 \in a$ $\qquad 1 \notin b$
$|a| = 3$ $\qquad |b| = 2$

Set Definition—Examples

- Let $T_1 = \mathit{set\ of}\ \{red, green, blue\}$ and x_1: T_1
  - $T_1 = \{\{\}, \{red\}, \{green\}, \{blue\}$ $\{red, green\}, \{red, blue\}, \{green, blue\},$ $\{red, green, blue\}\}$
  - T_1 is the power set of $\{red, green, blue\}$
  - $\{red, blue\}$ is a possible value for x_1

- Let $T_2 = \mathit{set\ of\ integer}$ and x_2: T_2
  - T_2 is the power set of the integers
  - $\{1, 3, 5, 7, 9\}$ is a possible value for x_2

Sets

Definition: A *set* is an unordered collection of elements of the same type.

- Direct set definition
  - enumeration: $\{x_1, x_2, \ldots, x_n\}$
  - logical condition: $\{x \mid f(x)\}$
  - integer range: $[L..U]$

- Set definition in terms of another type
  - $T = \mathit{set\ of}\ T_0$
  - T is the power set of T_0

Strings—Operations

Operation	Meaning
$s[i..j]$	substring of s from position i to position j
$s_1 \parallel s_2$	concatenation of s_1 and s_2
$=, \neq$	equal, not equal
$\in, \notin$	member, non-member
$s[i]$	i-th character of s
$\lvert s \rvert$	length of s

- Let $x =$ `"abcd"` and $y =$ `"ef"`

$x[0..3] =$ `"abcd"` $\quad x[3..3] =$ `"d"` $\quad x[3..1] =$ `""`

$x \parallel y =$ `"abcdef"` $\quad y \parallel x =$ `"efabcd"`

`'c'` $\in x \quad$ `'c'` $\notin y$

$x[0] =$ `'a'` $\quad x[3] =$ `'d'`

$|x| = 4 \quad |x[1..2]| = 2 \quad |y| = 2$

Primitive Types

- *integer*: $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - operations: $+, -, \times, /, =, \neq, <, \leq, >, \geq$
- *boolean*: $\{false, true\}$
  - operations: logical operators
- *char*: ASCII characters
  - operations: $=, \neq$
  - constants: enclosed in single quotes
- *string*: finite sequence of ASCII characters

Types and Variables

Definition: A *type* is a set.

- Variable definition
  - $x : T$ defines the variable x of type T
  - at all times: $x \in T$
- $x : integer$
- $c : \{red, green, blue\}$

Formality in Logical Expressions

- Logical expressions may be formal or informal
  - base your choice on simplicity and clarity
  - mix notation when it helps
  - avoid vagueness and ambiguity

- Let $s = \langle s[0], s[1], \ldots, s[n-1] \rangle$
  - s contains no duplicates
  - $(\forall i, j \in [0, n-1])(i \neq j \rightarrow s[i] \neq s[j])$

Quantifiers—Examples

- $(\forall x \in S)(\exists y \in S)(y < x)$

- $(\exists x \in S)(\forall y \in S)(x \leq y)$

- $\lim_{x \to x_0} f(x) = L$

  - $(\forall \epsilon \in$ positive reals$)(\exists \delta \in$ reals$)$
    $(|x - x_0| < \delta \to |f(x) - L| < \epsilon)$

Quantifiers

- Universal quantifier
  - $(\forall x \in S)(p(x))$
  - for all $x \in S$, $p(x)$ is $true$

- Existential quantifier
  - $(\exists x \in S)(p(x))$
  - there exists an $x \in S$ such that $p(x)$ is $true$

Logical Expressions—Examples

- $p \wedge q \rightarrow r$
- $\{x \mid x$ is an integer and $x \geq 1 \wedge x \leq 10\}$
- $\{x \mid x$ is an integer and $x \geq 1 \vee x \leq 10\}$
- $p \wedge q \vee r \wedge s$

Logical Expressions

Definition: A *logical expression* is a statement whose truth can be determined.

○ Logical operators (Boolean operators)

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
$true$	$true$	$false$	$true$	$true$	$true$	$true$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$false$	$false$	$true$	$false$	$false$	$true$	$true$

Functions—Examples

- Let $R_1 = \{\langle x, y \rangle \mid x$ is an int. and $y = x^2\}$ and $f_1(x) = x^2$
  - R_1 and f_1 are the same function
- Let $R_2 = \{\langle x, y \rangle \mid x$ is an integer and $y^2 = x\}$
  - R_2 is *not* a function
  - $\langle 9, 3 \rangle \in R_2$
  - $\langle 9, -3 \rangle \in R_2$
- Let $R_3 = \{\langle true, false \rangle, \langle false, true \rangle\}$ and $f_3(x) = \neg x$
  - R_3 and f_3 are the same function

Functions

Definition: A *function* is a binary relation in which each domain element appears in exactly one ordered pair.

○ $R_1 = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle\}$

○ $R_2 = \{\langle 0, 1\rangle, \langle 1, 0\rangle, \langle 2, 0\rangle\}$

Relations—Examples

- Let $R_1 = \{\langle 0,1\rangle, \langle 0,2\rangle, \langle 2,3\rangle\}$
  - the domain of $R_1 = \{0,2\}$
  - the range of $R_1 = \{1,2,3\}$

- Let $R_2 = \{\langle x,y\rangle \mid x$ and y are integers and $x < y\}$
  - the domain of R_2 is the set of integers
  - the range of R_2 is the set of integers

Relations

Definition: A *binary relation* is a set of ordered pairs.

Definition: The *domain* of a binary relation R is the set of all first elements of the ordered pairs in R.

Definition: The *range* of a binary relation R is the set of all second elements of the ordered pairs in R.

Sets

Definition: A *set* is an unordered collection of elements.

A set may be defined as follows:

- Enumeration
  - $\{10,1,6,7,3,8,2,4,9,5\}$
- Logical condition
  - $\{x \mid x$ is an integer between 1 and 10 inclusive$\}$
- Integer range
  - $[1,10]$

Document-driven Software Estimating

- New development
  - a good RS is extremely helpful
  - module decomposition is the critical task
  - for each module, five subtasks are immediate:

    development of MIS, MID, MI, TP, TI

- Maintenance
  - estimating is much simpler
  - sketch RS changes and review with the user
  - determine the modules affected: use the MG
  - for each affected module, five subtasks immediate:

    modification of MIS, MID, MI, TP, TI

Estimating Principles

- A procedure for estimating the cost of a task
  1. break the task into subtasks, recursively
  2. provide an estimate for each "leaf" subtask
  3. sum the subtask estimates

- Improving the accuracy of your estimates
  - sketch—partially complete—the task
  - begin with actual figures from a previous project

The Need for Accurate Estimates

- The driving questions:
  - How much will it cost?
  - When will it be done?
- Managers desperately need reliable answers
- Programmers do not have them

Testing versus Inspection

	Testing	Inspection	
+	Many tasks automatable Results are guaranteed	Automation difficult Results suspect	−
−	Results are specific Debugging still required Executability req'd	Results are general No debugging required Executability not req'd	+

Testing

- Dijkstra's law of testing
  - program testing can be used to show the presence of bugs, but never their absence
- Systematic testing standard practice
  - ad hoc testing is expensive and ineffective
- Isolate the module under test
  - otherwise, module testing very difficult
- Apply design for testability
  - facilitates effective testing
  - poor testability is a design weakness just like poor performance

The Inspection Process

- Overview meeting
  - introduce the work product to be inspected
- Preparation
  - all members study the work product
- Inspection meeting
  - careful line-by-line review
  - questions are pursued until a fault is found
- Rework
  - remove faults found during inspection meeting
- Follow-up
  - moderator is responsible for rework completion
  - rework $> 5\%$: reinspect the entire work product

Inspection Roles

- Moderator
  - controls all inspection meetings
  - ensures that the criticism is polite and constructive
- Reader
  - paraphrases the work product line-by-line
  - attempts to demonstrate that it is fault-free
  - usually not the author of the work product
- Other inspectors
  - designer, implementor, tester
  - listens to reader; attempts to find faults

Verification Terminology

Definition: A *fault* is a flaw in a work product.

Definition: A *failure* is the occurrence of incorrect run-time behavior

○ Inspection

- a disciplined review of a work product
- the goal is to find faults

○ Testing

- program execution
- the goal is to cause failures

Systematic Verification

Definition: Demonstrating that a work product is free of errors is called *verification*.

- Without systematic verification:
  - errors abound
  - software is sensitive to even trivial errors
  - standards are not enforced and so, are not achieved
- It pays to discover errors early
  - order of magnitude increase per phase
- Two complementary approaches to verification
  - review of work products: inspection
  - examination of execution behavior: testing

Sources of Assumptions

- Undetectable violation
  - illegal address
  - typographical error resulting in a legal command
- Cost and performance considerations
  - bounds checking omitted from production code
  - function parameter checking omitted by C compiler

Sources of Abnormal Behavior

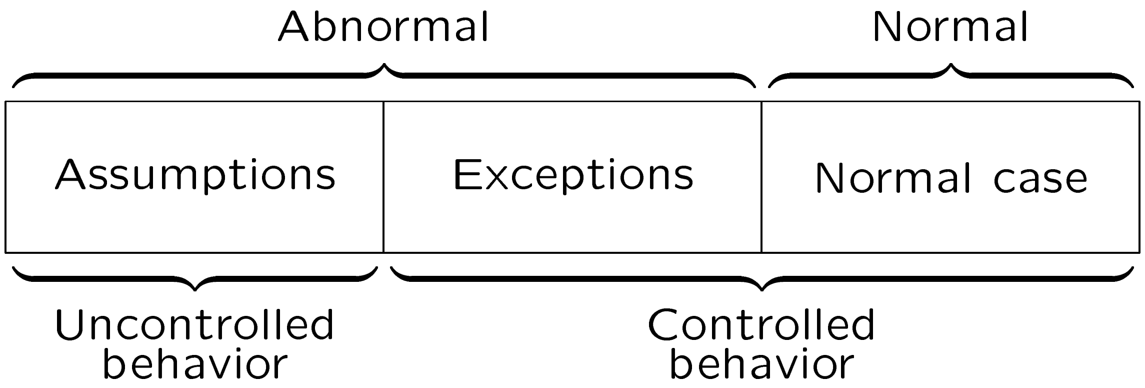
- Inherently illegal requests
  - divide by zero
  - fetch top element from an empty stack
  - GPA for a student whose number is not on file

- Resource restriction
  - push on full stack
  - write to full disk

The Specification Trichotomy

- Good engineers plan for the unexpected
  - because it happens so often
  - with planning the damage can be minimized

- It is critical to establish:
  - the normal operating range
  - the required behavior outside this range



Work Products

- Requirements Specification
  - specify required system behavior
- Module Guide
  - describe and motivate the module decomposition
- Module Interface Specification
  - specify the required behavior of each module
- Module Internal Design
  - specify the module internal data structures
- Module Implementation
  - production source code
- Test Plan and Test Implementation
  - strategy for selecting and executing tests
  - source code, data files, manual procedures

Triple Purpose Documentation

- Design documents
  - focus of design effort
  - record design decisions carefully
  - serve as the basis for design review
- Implementation documents
  - tell implementors what must be done
  - support parallel development
  - serve as a standard for correctness during testing
- Maintenance documents
  - support analysis of change requests
  - provide structure in which to record changes
  - aid in training new staff

Planned Documentation

- Plan the documentation carefully
  - provide a location for each relevant fact
- Criteria for good documentation
  - designed for reference use
  - easy to check for completeness and correctness
  - easy to change
- Three decisions for each document
  - audience: who are the readers?
  - prerequisites: what must the reader know?
  - purpose: what should the reader learn?

Documentation Today

- Current practice
  - documentation is viewed as a necessary evil
  - often, there is little or no documentation
  - different sets of doc. for each development phase

- The resulting documentation is:
  - incomplete and inaccurate
  - unplanned and undisciplined
  - hard to use
  - expensive to maintain