Tspec: A Specification of Memory Access Traces

Sally A. McKee[**], D.A.B. Weikle[*], and Wm. A. Wulf[*]
[**]Intel Corporation
{smckee@cse.ogi.edu}
[*]University of Virginia
{daw4q@cs.virginia.edu | wwulf@nae.edu}

Abstract.
This document describes Tspec, a "little language" for specifying memory traces, and Tint, an interpreter for Tspec. Tspec can be used to generate synthetic traces, or can be used to represent an actual trace in a human-readable, compressed format. We do not address here how to convert a trace into a Tspec specification.

1. Introduction

This working paper describes *Tspec*, a little language for specifying memory address traces, and *Tint*, an interpreter for Tspec. Both the language and the interpreter are moving targets, but the general structure of both are well defined, and will not change much from what is described herein.

Tspec is intended for use with the MACE (Memory ACess Evaluation) system [Wei97]. It can be used to generate synthetic traces (or subtraces), or to represent an actual trace in a compressed format that is still human-readable (so some of the underlying structure of the program that produced the trace should be evident).

2. Overview of the Tspec Language

A memory trace is a sequence of <*trace atom*>s, each of which is a pair consisting of an integer and an optional tag. The integers are addresses, and may be specified in decimal or hexadecimal notation; the tags are not defined by TSpec, but are intended to connote code vs. data, read vs. write, system vs. user, etc. For the purpose of this document, non-null tags will be represented by an underscore and one or more letters appended to the end of an integer — thus `100_cr` is an integer/tag pair.

A <*trace specification*> is a formal rule that generates a trace when *executed*. To be executed, a trace specification must first be *instantiated*; when instantiated, a specification acquires a *control pointer* that, like the program counter of a CPU, points to the next part of the specification to be executed.

An instance of a <*trace specification*> may be executed in either of two modes — it may be *pulsed* or *run*. If pulsed, the trace instance will generate the sequence corresponding to advancing its control pointer one unit (one <*generating item*> or <*control item*>, as defined below). If run, the instance will generate the sequence corresponding to advancing its control pointer to the end of the instance.

A memory address trace is represented by a <*trace specification*>. The simplest <*trace specification*> is just a list of <*trace atom*>s and some syntactic sugar. Specifications may be made more compact, however, by introducing three notions from programming languages:

- variables (that hold atoms),
- definite iteration, and
- subprogram-like definitions.

3. Tspec Grammar

In this section, we interleave description of the syntax with semantics of the specifications. In the syntactic definition, single quotes surround the delimiters of the language being defined. Thus, in the first definition below, a <*trace specification*> starts with an opening curly brace ('{') and consists of a semicolon-separated sequence of <*declaration*>s followed by a <*trace*>, a semicolon (';'), and a closing curly brace ('}').

<*trace specification*> *::= 'TRACE'*
{<declaration>';'}<trace>';' 'ECART'*

<*declaration*> *::=*
'VAR' {<variable defs>','}<variable def>* |
'SUB' <trace id> '(' <instance names> ')' '=' <trace>

<*instance names*> *::= {<instance names>*
','}<instance name>*

<*instance name*> *::= <identifier>*

<*variable def*> *::=*
<trace variable> '(' <trace atom> ',' <integer> ')'

<*trace variable*> *::= <identifier>*

<*trace atom*> *::= <integer><tag>*

The <*trace*> at the end of a <*trace specification*> will be instantiated and run — and presumably it will use the variables and subtraces declared at the beginning.

A <*trace variable*> declaration has the form:

```
VAR x(100cr,4), y(500,-1)
```

Capitalization of keywords is optional, so `var` and `VAR` are both recognized.

The quantities inside the parentheses provide initial values (the <*trace atom*>s) and increments (the <*integer*>s) associated with the variables; note that tags may be null, and negative increments are permitted.

A subtrace declaration is of the form:

```
SUB z (z1, z2) = <trace>
```

The <*trace id*>, z in this example, currently only serves as documentation; the name should indicate something about the nature of the subtrace.

We'll define a <*trace*> below, but, for now, note that the declaration contains one or more <*instance name*>s. A separate instantiation is created and associated with each instance name (each <*instance name*> may be thought of as a distinct control pointer).

<*trace*> *::= <trace item>*

| <*trace*> <*trace item*>

<*trace item*> ::= <*generating item*>
| <*control item*>

When executed, the *generating item*s produce <*trace atom*>s in the resulting trace; the control items exist to control the execution of the <*trace specification*>, and do not generate <*trace atom*>s. The control items are:

<*control item*> ::= '!'<*trace variable*>
| '!'<*instance name*>
| '('<*trace*>')'
| <*trace item*>'*'<*integer*>
| <*trace variable*>'#'<*integer*>

The exclamation mark denotes initialization. In the case of a <*trace variable*>, it causes the variable to be set to the <*trace atom*> in its definition. In the case of an <*instance name*>, it causes the control pointer to be set to the beginning of its defining <*trace*>.

A <*trace*> (a sequence of <*trace item*>s) can be parenthesized to form a <*trace item*>. This grouping is useful with the iteration constructs defined next.

By loose analogy with Kleene's Star, the construct <*trace item*>'*'<*integer*> denotes definite iteration — the <*trace item*> is executed the number of times denoted by the <*integer*>. Specifying `100*4*4` causes `100` to be generated 16 times.

The <*trace variable*>'#'<*integer*> construct overrides the post-increment defined in the <*trace variable*>'s declaration.

The <*generating item*>s produce <*trace atom*>s:

<*generating item*> ::= <*trace atom*>
| <*trace variable*>
| <*instance name*>
| '@'<*instance name*>

Execution of a <*trace atom*> causes that atom to be generated. The execution of a <*trace variable*> generates the current value of the variable and post-increments the variable by its increment, the second value in its definition (the tag is not changed, of course).

An <*instance name*> may be prefixed by an at sign ('@'). The @ symbol causes a pulsed execution, executing only the next item in the trace; without it, the <*trace*> will be run in its entirety.

Consider the following example (using null tags, for simplicity):

```
{
    VAR a(100,4) b(500,-4);
    SUB s(s1) = (800 900 430 164);
    !a !b !s1 (a @s1 b)*4
}
```

When executed, this will produce the trace:

```
100 800 500 104 900 496 108 430 492 112 164 488
```

As a second example, consider the trace for a loop that copies a vector from one location in memory to another. Using the tag `cr` to denote reading code, and `dr` and `dw` for reading and writing data, respectively, the *trace specification* might look something like:

```
{
    VAR c(100_cr,4) f(200_dr,4) t(300_dw,4);
    !f !t (!c c f c t c)*50
}
```

Note that the placement of the initialization of `f` and `t` are outside the iteration, and that of `c` is inside it. When executed, this will produce the trace:

```
100_cr  200_dr  104_cr  300_dw  108_cr  100_cr
204_dr 104_cr 304_dw 108_cr 100_cr …
```

4. Comments

4.1 Probabilistic Generation

Another useful construct is a <*control item*> that allows the <*generating item*> to which it refers to be generated probabilistically, to model conditional paths of execution synthetically. The proposed syntax below is far from elegant, and is likely to be changed in subsequent versions:

<*trace item*> '?' <*integer*>:<*integer*>

where the <*trace item*> would be generated (or fired) with probability n/m, where n and m are the <*integer*>s given in the item. If only one integer is given, it is assumed to be the second, m, and n defaults to 1. As a special case, <*trace item*> '?' 0 means that the item is evaluated but never generated. This is useful in conjunction with the # operator to adjust the address of a data stream. To see this, consider matrix multiplication:

```
for (i = 0; i < N; i++) {
    for (k = 0; k < N; k++) {
        r = x[i][k];
        for (j = 0; j < N; j++) {
            z[i][j] += r * y[k][j];
        }
    }
}
```

Arrays *x* and *y* are accessed linearly, but the address or pointer for *y* needs to be reset back to the beginning of the array at each completion of the *k*-loop. Array *z*'s pointer needs to be reset to the beginning of the current row after each completion of the *j*-loop, and advanced to the next (i^{th}) row after each completion of the *k*-loop. This computation involves four streams: *xin*, *yin*, *zin*, and *zout*. The construct *zout#-ROWSIZE?0* adjusts the address for the *z* output stream without generating an address in the trace. Similar constructs can be used to adjust the *zin* and *yin* pointers.

4.2 Post-Execution Control Pointer State

The control pointer for a subtrace could be positioned back at the beginning of the subtrace after being run, or it could be left pointing to the (null) end of the trace. Currently, the effect of re-running a trace without reinitializing the control pointer is undefined.

4.3 Error Processing and Recovery

Undeclared variables signal an error. Other error conditions need to be defined, and error processing and recovery need to be built into the interpreter.

5. Tint: the Tspec Interpreter

The current incarnation of Tint is a simple filter that reads the Tspec source file from the standard input, producing the corresponding trace on the standard output. The program is written in C, and uses lex [Les90] to tokenize the input and yacc [Joh90] to parse it.

The `TBEGIN` and `TEND` tokens have been defined to be opening and closing curly braces in the lex source file, tint.lex. These can be redefined to be `TRACE` and `ECART` or *m4* macros for TRACE and ECART can be defined, as described in the next paragraph.

Macro definitions can be used in Tspec source "programs" by running the source file through *m4* and piping the output through tint. We provide a shellscript, Tint, that does precisely this. Symbolic constants attempt to improve the readability of Tspec synthetic trace specifications. Unfortunately, the specifications are still not as easy to read as we would like them to be.

The present implementation of tint detects an undeclared or uninitialized variable or subprogram, and announces that such an error condition exists by naming the offending item. It does not provide any information as to the line number in the source specification, nor does it attempt any error recovery. Many other errors go undetected. Improper specifications often produce no output. The software (in its current incarnation) is provided for use "as is"; disclaimers apply.

Note that when using *m4*, it is necessary to include the "changecom" macro directive if a trace includes the # operator, since this is interpreted as a comment by *m4* (and the rest of the line that contains the # will not be passed to tint). A standard include file should be provided, but is not available as yet.

0.1. References

[Joh90] S.C. Johnson and R.Sethi, "Yacc: A Parser Generator", *Unix Research System Papers, 10th Edition*, AT&T Bell Laboratories, Murray Hill, NJ, 1990, pp. 347-374.

[Les90] M.E. Lesk, E. Schmidt, "Lex — A Lexical Analyzer Generator", *Unix Research System Papers, 10th Edition*, AT&T Bell Laboratories, Murray Hill, NJ, 1990, pp. 375-387.

[Wei97] D.A.B. Weikle, S.A. McKee, Wm.A. Wulf, "A Stake In The Ground: A New Approach to Cache Analysis", University of Virginia Department of Computer Science Technical Report CS-97-20, December 1997.

Appendix A: Sample Specifications

The following examples should help demonstrate how Tspec can be used to specify sample traces. The first specification is for a daxpy operation in which only a data address stream is generated. The "dnl" construct denotes a comment (it instructs *m4* to delete all characters from the current position to the end of line).

```
dnl -- daxpy.data.10000.t
define(STRIDE1_DOUBLE,8)
define(SCALAR,0)
{
        var x(0x1000000_dr, STRIDE1_DOUBLE),
           yin(0x2000000_dr, STRIDE1_DOUBLE),
           yout(0x2000000_dw, STRIDE1_DOUBLE),
           a(0x3000000_dr, SCALAR);
        sub daxpy(pc) = (a x yin yout);
        !x !yin !yout !a !pc
        pc*10000
}
```

Here is the specification for a daxpy trace that models both code and data addresses. This specification does not use the *m4* macro package, but instead models each instruction as a scalar variable with tag _cr and increment 0:

```
dnl -- daxpy.cd.10000.t
{
        var x(0x1000000_dr, 8),
           yin(0x2000000_dr, 8),
           yout(0x2000000_dw, 8),
           a(0x3000000_dr, 0);
        var loada(0x001000_cr, 0),
           loadx(0x001004_cr, 0),
           multax(0x001008_cr, 0),
           loady(0x00100C_cr, 0),
           plusyax(0x001010_cr, 0),
           storey(0x001014_cr, 0),
           incri(0x001018_cr, 0),
           test(0x00101C_cr, 0),
           jump(0x001020_cr, 0);
        sub daxpyd(data) = (a x yin yout);
        sub daxpyc(code) = (loada loadx multax loady plusyax storey incri
test jump);
                !x !yin !yout !a !data !code
                !loada !loadx !multax !loady !plusyax !storey !incri !test !jump
                (!code !data @code @data @code @data @code @code @data
@code @code @data @code @code @code)*10000
        }
```

The final specification produces the example reference stream from [Wei97]:

$$[0^k, 1, N^k, 2, 0^k, 3, N^k, 4, 0^k, \ldots, (N-1), N^k]^*$$

where k=1 and N=8192.

```
define(N,8192)
define(halfminusone,4095)
define(k,1)
{
        var x(1, 1);
        (!x (0*k x N*k x)*halfminusone 0*k x N*k)*2
}
```

Appendix B: *tint* Source

B.1. File *tint.h*:

```
/* tint.h */

#ifndef __TINT_H__
#define __TINT_H__

enum Operator {
        Pulse,
        Repeat,
        Increment,
        Initialize,
        Generate,
        Declare,
        Probability
};

enum Type {
        Variable,
        Subtrace,
        Trace,
        Instance,
        Atom,
        Unknown,
};

struct descriptor;
struct variable;
struct instance;
struct list;
struct symbol;


typedef struct atom {
        int value;
        char tag[1];
} atom;

typedef struct list {/* works for tlists, ilists, vlists */
        struct list *next;/* symbol table */
        struct descriptor *item;
} list;

typedef struct symbol {
        struct symbol *next;
        char *name;
        struct descriptor *guts;
} symbol;

typedef struct variable {
        atom *a;    /* holds current value and tag */
        int initial,    /* initial value */
            stride;    /* increment/decrement */
} variable;

typedef struct instance {
        list *trace,   /* start of trace */
             *pc;      /* pointer into list */
} instance;
```

```
typedef struct descriptor {
        enum Type type;/* Atom, Trace, Instance, etc. */
        enum Operator op;/* Pulse, Run, Initialize, etc. */
        int n;          /* count or increment */
        int m;          /* additional integer when needed */
        symbol *sym;
        union {
                atom *a;
                variable *var;
                instance *inst;
                list *trace;
        } val;
} descriptor;

typedef void (*proc)(descriptor *);

typedef struct proctab {
        enum Operator op;
        proc p;
} proctab;

extern symbol *symtab;
extern int state;
extern list *tp;

extern void init();
extern void exec();

#endif
```

B.2. File *misc.h*:

```
/* misc.h */

#ifndef __MISC_H__
#define __MISC_H__

#include "tint.h"

extern void yyerror(char *s);

/* declare a fatal run-time error */
#define fatal(fmt, args...)  _fatal(__FILE__,\
                __FUNCTION__,       \
                __LINE__,           \
                fmt,                        \
                ## args)

extern void _fatal(char *file, char *func, int line, char *fmt, ...)
        __attribute__ ((noreturn));

/* print run-time waring */
#define warn(fmt, args...)  _fatal(__FILE__,\
                __FUNCTION__,       \
                __LINE__,           \
                fmt,                        \
                ## args)

extern void _warn(char *file, char *func, int line, char *fmt, ...);

extern symbol *lookup(char *);
extern char *alloc();
extern descriptor *newd(enum Type type,
        enum Operator op,
        int n,
        int m,
        symbol *sym,
        void *p);
extern atom *newa(int value, char *tag);
extern list *newl(descriptor *item);
extern variable *newv(atom *a, int value, int stride);
extern instance *newi();
extern void *setitems(list *p, list *t);
extern list *rev(list *p);

#endif
```

B.3. File *grammar.y*:

In the *yacc* specification below, the functions newa(), newv(), newd() and newl() create new atom, variable, descriptor, or list structures, respectively. Each production creates or modifies a list element, which is linked into the trace being created.

```
%{
#include <stdio.h>
#include "tint.h"
#include "misc.h"
#define yyerror(s)
%}

%start spec

%union {
        int number;
        char *string;
        symbol *sym;
        atom *a;
        instance *inst;
        list *l;
        descriptor *d;
        void *v;
};
%token <sym> TID VID IID
%token <number> NUMBER
%token <string> TAG UNKNOWN SUB VAR TBEGIN TEND

%type<a> atom
%type<l> namelist trace item
%type<d> iname


%%

spec  : /* empty */
        | TBEGIN declist trace TEND {tp = rev($3); }
        ;

declist:
        | declist decl
        ;

decl  : VAR
                                {       state = VID; }
          varlist ';'
        | SUB
                                {       state = TID; }
          TID '('
                                {       state = IID; }
          namelist ')' '=' '(' trace ')'
                                {       setitems($6, rev($10)); }
          ';'
        ;


varlist: vardef                 {       state = VID; }
        | varlist ',' vardef    {       state = VID; }
        ;

vardef: VID '(' atom ',' NUMBER ')' {$1->guts = newd(Variable,
                                        Declare,
                                        0,
                                        0,
                                        $1,
                                        (void *)newv($3,
                                                $3->value,
                                                $5));
                                }
        ;

namelist: iname                 {       $$ = newl($1);
                                        state = IID;
                                }
        | namelist ',' iname    {       $$ = newl($3);
                                        state = IID;
                                        $$->next = $1;
                                }
        ;

iname:IID                       {       $$ = $1->guts = newd(Instance,
                                                Declare,
                                                0,
                                                0,
                                                $1,
```

```
                        (void *)newi());
                }

atom : NUMBER TAG       {       $$ = newa($1, $2); }
        | NUMBER        {       $$ = newa($1, NULL); }
        ;

trace : item            {       $$ = $1; }
        | trace item    {       $$ = $2;
                                $$->next = $1;
                        }
        ;
item  : atom            {       $$ = newl(newd(Atom,
                                        Generate,
                                        1,
                                        0,
                                        NULL,
                                        (void *)$1));
                        }
        | VID           {       if (!$1->guts) {
                                        fatal("undeclared/uninitialized
variable %s\n",
                                                $1->name    ?    $1->name    :
"nameless");
                                }
                                $$ = newl(newd(Variable,
                                        Generate,
                                        1,
                                        0,
                                        $1,
                                        (void *)$1->guts->val.var));
                        }
        | IID           {       if (!$1->guts) {
                                        fatal("undeclared/uninitialized    item
'%s'\n",
                                                $1->name    ?    $1->name    :
"nameless");
                                }
                                $$ = newl(newd(Instance,
                                        Generate,
                                        1,
                                        0,
                                        $1,
                                        (void *)$1->guts->val.inst));
                        }
        | '@' IID       {       if (!$2->guts) {
                                        fatal("undeclared/uninitialized    item
'%s'\n",
                                                $2->name    ?    $2->name    :
"nameless");
                                }
                                $$ = newl(newd(Instance,
                                        Pulse,
                                        1,
                                        0,
                                        $2,
                                        (void *)$2->guts->val.inst));
                        }
        | '!' VID       {       if (!$2->guts) {
                                        fatal("undeclared variable '%s'\n",
                                                $2->name    ?    $2->name    :
"nameless");
                                }
                                $$ = newl(newd(Variable,
                                        Initialize,
                                        1,
                                        0,
                                        $2,
                                        (void *)$2->guts->val.var));
                        }
        | '!' IID       {       if (!$2->guts) {
                                        fatal("undeclared item '%s'\n",
                                                $2->name    ?    $2->name    :
"nameless");
                                }
                                $$ = newl(newd(Instance,
                                        Initialize,
                                        1,
                                        0,
                                        $2,
                                        (void *)$2->guts->val.inst));
                        }
        | '(' trace ')'         {       $$ = newl(newd(Subtrace,
                                        Generate,
                                        1,
                                        0,
                                        NULL,
                                        (void *)rev($2)));
                                }
        | item '*' NUMBER       {       $$ = newl(newd(Subtrace,
                                        Repeat,
                                        $3,
                                        0,
                                        NULL,
                                        (void *)$1));
                                }
        | VID '#' NUMBER        {       if (!$1->guts) {
                                        fatal("undeclared/uninitialized
variable %s\n",
                                                $1->name    ?    $1->name    :
"nameless");
                                }
                                $$ = newl(newd(Variable,
                                        Increment,
                                        $3,
                                        0,
                                        $1,
                                        (void *)$1->guts->val.var));
                                }
        | item '?' NUMBER       {       $$ = newl(newd(Variable,
                                        Probability,
                                        $3,
                                        1,
                                        NULL,
                                        (void *)$1));
                                }
        | item '?' NUMBER ':' NUMBER{$$ = newl(newd(Variable,
                                        Probability,
                                        $3,
                                        $5,
                                        NULL,
                                        (void *)$1));
                        }
        ;
%%
```

B.4. File *tint.lex*

```
/* tint.lex */


%{
#include "tint.h"
#include "misc.h"
#include "y.tab.h"

%}

L               [a-zA-Z]
A               [_a-zA-Z0-9]
D               [0-9]
X               [0-9a-fA-F]

%%

"\{"            {
                        return(TBEGIN);
                }

"\}"            {
                        return(TEND);
                }

VAR|var         {
                        return(VAR);
                }

SUB|sub         {
                        return(SUB);
                }

[+-]?{D}+{
                        yylval.number = atoi(yytext);
                        return(NUMBER);
                }
```

```
0x{X}+      {
                yylval.number = strtoul(yytext, NULL, 16);
                return(NUMBER);
            }

_{L}{L}*{
                yylval.string = yytext;
                return(TAG);
            }

{L}{A}*     {
                yylval.sym = lookup(yytext);
                if (yylval.sym->guts) {
                    switch (yylval.sym->guts->type) {
                        case Variable:return(VID);
                        case Trace:return(TID);
                        case Subtrace:return(IID);
                        case Instance:return(IID);
                        default:return(UNKNOWN);
                    }
                }
                else if (state) {
                    return(state);
                }
                return(UNKNOWN);
            }

[ \t\n]*;   /* ignore whitespace */


.           {
                return(*yytext);
            }

%%
```

B.5. File *misc.c*

```
/* misc.c -- allocation routines, symbol table, etc. */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "tint.h"
#include "misc.h"
#include "y.tab.h"


static symbol *symtab;

void yyerror(char *s)
{
    printf(s);
}

void _fatal(char *file, char *func, int line, char *fmt, ...)
{
    va_list v;
    va_start(v, fmt);

    fprintf(stderr, "fatal: ");
    vfprintf(stderr, fmt, v);
    fprintf(stderr, "\n");
    exit(-1);
}

void _warn(char *file, char *func, int line, char *fmt, ...)
{
    va_list v;
    va_start(v, fmt);

    fprintf(stderr, "warning: ");
    vfprintf(stderr, fmt, v);
    fprintf(stderr, "\n");
}

char *alloc(int n)
{
    char *p;

    p = (char *) calloc(1, n);
    if (!p) {
        fatal("out of memory");
    }
    return(p);
}

/*
 * looks up name in symbol table
 * if not found, it inserts the name
 * but can't insert the info for it (haven't seen that yet)
 */

symbol *lookup(char *s)/* change to hash table later */
{
    symbol *p;

    for (p = symtab; p; p = p->next) {
        if (p->name && !strcmp(p->name, s)) {/* found it */
            return(p);
        }
    }
    p = (symbol *)alloc(sizeof(symbol));/* make it */
    p->guts = NULL;
    p->name = alloc(strlen(s) + 1);
    strcpy(p->name, s);
    p->next = symtab;
    symtab = p;
    return(p);
}

#if 0
/* don't need this? */
symbol *update(char *s, descriptor *guts)
{
    symbol *p;

    for (p = symtab; p; p = p->next) {
        if (p->name && !strcmp(p->name, s)) {/* found it */
            if (p->guts) {
                warn("update: overwriting previous info for %s", s);
            }
            p->guts = guts;
        }
    }
    fatal("update: symbol %s not in table", s);
}
#endif

descriptor *newd(enum Type type, enum Operator op, int n, int m, symbol
*sym, void *p)
{
    descriptor *guts;

    guts = (descriptor *)alloc(sizeof(descriptor));
    guts->type = type;
    guts->op = op;
    guts->n = n;
    guts->m = m;
    guts->sym = sym;
    switch (type) {
        case Atom:guts->val.a = (atom *) p;
                  break;
        case Instance:guts->val.inst = (instance *)p;
                  break;
        case Subtrace:
        case Trace:guts->val.trace = (list *)p;
                  break;
        case Variable:guts->val.var = (variable *)p;
                  break;
        default:fatal("newd: unrecognized descriptor type");
    }
    return(guts);
}

atom *newa(int value, char *tag)
{
    atom *p;

    if (!tag) {
        tag = "";
    }
    p = (atom *)alloc(sizeof(atom)+strlen(tag));
    p->value = value;
    strcpy(p->tag, tag);
    return(p);
```

```
}

list *newl(descriptor *item)
{
        list *p;

        p = (list *)alloc(sizeof(list));
        p->next = NULL;
        p->item = item;
        return(p);
}

variable *newv(atom *a, int value, int stride)
{
        variable *p;

        p = (variable *)alloc(sizeof(variable));
        p->a = a;
        p->initial = value;
        p->stride = stride;
        return(p);
}

instance *newi()
{
        instance *p;

        p = (instance *)alloc(sizeof(instance));
        p->trace = NULL;
        p->pc = NULL;
        return(p);
}

void *setitems(list *p, list *t)
{
        for (; p; p = p->next) {
                p->item->val.inst->trace = t;
                p->item->val.inst->pc = t;
        }
}

list *rev (list *p)
{
        list *q,
             *r;

        for (r = NULL; p; p = q) {
                q = p->next;
                p->next = r;
                r = p;
        }
        return(r);
}
```

B.6. File *tint.c*

```
/* tint.c */

#include <stdio.h>
#include <stdlib.h>
#include "tint.h"
#include "misc.h"
#include "funs.h"
#include "y.tab.h"

int state;
int silent = 0;
list *tp;

static proctab table[] = {
        Pulse,        pulse,
        Repeat,       repeat,
        Increment,increment,
        Initialize,initialize,
        Generate,generate,
        Declare,declare,
        Probability,probability,
        -1,           0,
};

/* should make associative table for these, and have a fun return char * */
static char *opnames[] = {
        "Pulse",
        "Repeat",
        "Increment",
        "Initialize",
        "Generate",
        "Declare",
        "Probability",
        0,
};


main()
{
        state = -1;
        yyparse();
        exec(tp);
}

void exec(list *p)
{
        descriptor *d;
        int i;

        for (; p; p = p->next) {
                d = p->item;
                for (i = 0; table[i].op != -1; i++) {
                        if (d->op == table[i].op) {
                                (*table[i].p)(d);
                                break;
                        }
                }
                if (table[i].op == -1) {
                        fatal("exec: bad operator %s %d (%s)\n",
                                opnames[d->op],
                                d->op,
                                d->sym ? d->sym->name : "anonymous");
                }
        }
}

void output(int val, char *tag)
{
        if (!silent) {
                printf("%d%s\n", val, tag);
        }
}

void initialize(descriptor *d)
{
        switch (d->type) {
                case Instance:d->val.inst->pc = d->val.inst->trace;
                        break;
                case Variable:d->val.var->a->value = d->val.var->initial;
                        break;
                default:fatal("initialize: bad descriptor type %d (%s)",
                                d->type,
                                d->sym ? d->sym->name : "anonymous");
        }

}

void generate(descriptor *d)
{
        switch (d->type) {
                case Atom:output(d->val.a->value,
                                d->val.a->tag);
                        break;
                case Instance:exec(d->val.inst->pc);/* go to end */
                        d->val.inst->pc = d->val.inst->trace;
                        /* define pc to wrap */
                        break;
                case Variable:output(d->val.var->a->value,
                                d->val.var->a->tag);
                        d->val.var->a->value += d->val.var->stride;
                        break;
                case Subtrace:exec(d->val.trace);/* ever? */
                        break;
                default:fatal("generate: bad descriptor type %d",
                                d->type);
        }
}

void pulse(descriptor *d)
{
```

```
        list element;

        switch (d->type) {
                case Instance:element.item = d->val.inst->pc->item;
                        element.next = NULL;
                        d->val.inst->pc = d->val.inst->pc->next;
                        exec(&element);
                        break;
                default:fatal("pulse: bad descriptor type %d",
                                d->type);
        }
}

void increment(descriptor *d)
{
        switch (d->type) {
                case Variable:output(d->val.var->a->value,
                                d->val.var->a->tag);
                        d->val.var->a->value += d->n;
                        break;
                default:fatal("increment: bad descriptor type %d",
                                d->type);
        }
}

void repeat(descriptor *d)
{
        int i;

        if (d->type != Subtrace) {
                fatal("repeat: bad descriptor type %d", d->type);
        }
        for (i = 0; i < d->n; i++) {
                exec(d->val.trace);
        }
}

void declare(descriptor *d)
{
        fatal("declare: should never be executed (%s)\n", d->sym ? d->sym-
>name : "anonymous");
}

void probability(descriptor *d)
{
        int i;
        long r;

        if (!d->n) {
                silent = 1;                     /* global flag */
                exec(d->val.trace);
                silent = 0;                     /* turn printing on */
        }
        else {
                r = random();
                if ((r % d->m) >= (d->m - d->n)) {/* fire w/ prob */
                        exec(d->val.trace);/*   d->n/d->m */
                }
        }
}
```

B.7. File *makefile*

```
CC=/usr/local/bin/gcc
#CFLAGS=-g
CFLAGS=-O2

.SUFFIXES: .c

.c.o:
        $(CC) $(CFLAGS) -c $*.c

tint:   y.tab.o lex.yy.o tint.o misc.o
        $(CC) -o tint tint.o misc.o y.tab.o lex.yy.o -ly -ll -lm

lex.yy.o:tint.lex
        lex tint.lex
        $(CC) $(CFLAGS) -c lex.yy.c

y.tab.o:grammar.y
        yacc -d -t grammar.y
        $(CC) $(CFLAGS) -c y.tab.c

clean:
        -rm *.o y.tab.h lex.yy.c core
```

B.8. File *Tint*

```
#!/bin/sh
cat $1 | m4 | tint
```