**Description**

Slang is a discrete event simulation language. It allows for a programmer to schedule events at discrete times, and have those events executed based on an event queue ordered by start time.

**Program**

A program is a single file consisting of functions, defined and written above the main function, and main function. The main block consists of zero, or at least one (or more) of an init block and/or an always block. An init block consists of statements that are to be executed sequentially from time 0. An always block consists of code to be continually run until program termination. init and always blocks can only exist within the main function, not within any other function. The program ends when the Terminate keyword is used.

**Scoping**

Slang uses block scoping, such that something that is defined within a block is defined within all blocks following the line in which it was defined, within that block.

Variables can exist within any function, and exist within functions that are called within the outer function. Local variables take precedence over variables defined outside the function.

There are no global variables, but variables defined within main, above the init and always blocks, are visible within all blocks.

**Execution**

The simulator processes events in the event queue by removing all the events for the current time and processing them. During the processing, more events may be created (such as by functions) and placed in the proper place in the queue for later processing. When all the events of the current time have been processed, the simulator advances time and processes the next events at the front of the queue.

**Whitespace**

Slang is whitespace-ambivalent, meaning that whitespace does not affect the program.

**Identifiers**

An identifier is used to refer to an object, data structure, or primitive type, in declarations, assignments, and general statements.

```
int decl_ident; /* declaration */
int ident_1 = 5; /* assignment */
ident_1 + 2 == 7; /* general statement; evaluates to true */
```

An identifier must start with with a character letter and can contain any combination of numbers, letters, and the underscore symbol `_`. Whitespace signals the end of the identifier.

**Punctuation**

**Parenthesis**

Expressions can include expressions inside parentheses. Parentheses can also indicate a function call.

**Braces**

Braces indicate a statement in blocks.

**Semicolon**

Used to separate statement and expression in a for loop. Used at the end of every statement.
Colons are used specifically for property lists when declaring an object.

**Escape Sequences**

A backslash followed by a character is used to represent certain characters within strings.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>\</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>\n</td>
<td>linefeed</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>horizontal tabulation</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
</tbody>
</table>

**Comments**

Multiline comments in Slang start with /* and terminate with the next */. Multiline comments do not nest.

**Data Types**

**Int**

An int is a 32-bit signed integer.

**Float**

A float is a 64-bit signed floating-point number.

**Boolean**

A boolean value defined using the keywords true or false.

```slang
bool x = true;
```

**String**

A string is a sequence of characters. String literals are placed between double quotes.

**Objects**

In Slang, an object is an encapsulation of a set of user defined properties. An object can be declared and not defined as:

```slang
object person;
```

Alternatively, an object can be defined as:

```slang
object person = object(string name="Bob", int age=25, string nicknames[] = ["Bobby", "Lil' B"]);
```

The user also has the option to not initialize the values of the properties (such as name and age) but is required to list out all of the properties of the Object on definition.
operators

Some operators (such as +) are overloaded. They must be used on expressions of the same type, as Slang will not automatically promote.

Arithmetic

- + addition, addition, and String concatenation.
- - subtraction and unary negation.
- * multiplication.
- / division
- % modulo

*, /, and % operators have precedence over + and -

Assignment

= Assigns value of right hand side to the left hand side. Assignment has right to left precedence.

Comparison

- == equal to. Compares values of two items. In order for two Objects to be equal, they must be the same object.
- != not equal to
- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

Logical

- ! unary not
- & logical and
- | logical or

Precedence Order

Operators within the same row share the same precedence. Higher rows indicate higher precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>--, ++, -</td>
<td>right</td>
</tr>
<tr>
<td>!</td>
<td>right</td>
</tr>
<tr>
<td>*, -, %</td>
<td>left</td>
</tr>
<tr>
<td>+, -</td>
<td>left</td>
</tr>
<tr>
<td>&amp;</td>
<td>left</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>left</td>
</tr>
<tr>
<td>==, !=</td>
<td>left</td>
</tr>
<tr>
<td>=</td>
<td>right</td>
</tr>
</tbody>
</table>

Declarations

Variables

Variables can be defined within the main function, within individual functions, or within an init/always block. Variables may be declared and then defined, or done both at the same time. Declaration:
int a;

Definition & Declaration:

int a = 5;

**Arrays**

In Slang, you can have arrays of any type. You declare an array as follow:

```plaintext
type name_of_array[];
```

You can also create an array and fill it with values, as in:

```plaintext
type name_of_array[] = [5,6,7]
```

Arrays are dynamically sized.

**Functions**

```plaintext
func returntype func_name(type var1, type var2, …) {
   body
}
```

Functions are defined only by identifying the block of code with `func`, giving the function a name, optionally supplying parameters, and defining a function body. Function return types are any data type, or `void` for no value.

**Example:**

```plaintext
func void Stuff(Object person){
   line.add(person);
   #10 line.remove(person);
}
```

**Statements**

Statements are used to get the program to do something. Statements are used for variable declarations and assignment, delays, returns, control flow, loops, and expressions. All statements end with a semicolon `;`. Statements are used within blocks. All of these are examples of statements:

```plaintext
{
   4+4;
   #5 return 0;
   if (5<6) {
      string name = "Pete";
   }
   Terminate;
}
```

**Blocks and Control Flow**

**Blocks**

A block is defined inside curly braces, which can include a possibly-empty list of variable declarations and statements. A block is structured as:

```plaintext
{
   statements
}
```

**If-else**
Slang accepts:

- if expression block

```
if (3 == 3){
    block
}
```

- if expression block else block where expression evaluates to a boolean value. The else keyword is tied to the nearest previous if. Example:

```
if (5>10) { /*if statement one*/
    if (4>5) { /*if statement two*/
        block
    } else { /*tied to if statement two*/
    }
}
```

**Iteration**

Slang has for-loops and while loops. A for loop can be:

- starting assignment; boolean loop condition expression; assignment for advancing to next iteration { block }

```
for (int i = 0; i < 10; i++) {
    block
}
```

- or an empty for;; { block }, which is infinite

```
for;; {
    block
}
```

A while loop is:

```
while (condition) {
    block
}
```

**Typecasting**

Typecasting can be used to change the type of value to another type.

**Casting to numbers**

Casting a float to an integer truncates the fraction part, while casting an integer to an integer. Casting a bool to an int or a float would result in either 1 or 0. Casting a string to a float or integer is allowed if the string contains numerical values that would be found in an integer or float.

```
float a = 42.3;
int b = int(a); /*b now holds value 42*/
a = float(b); /*a now holds value 42.0*/
a = float(true); /*a now holds value 1.0*/
b = int(false); /*b now holds value 0*/
```
Casting to Strings and Booleans

Casting an integer or float to a string converts the number into a string, while casting a bool into a string creates either "true" or "false." Converting any number above 0 or non-empty string to a bool leads to true, while 0 and a string whose value is "false" or is empty leads to false.

```plaintext
float c = 20.3;
bool d = false;

string number = string(c); /*string now holds value "20.3"*/
string boolean = string(d); /*string now holds value "false"*/

d = bool(20); /*d is now true*/
d = bool("false"); /*d is now false, same as if d = bool(boolean)*/
d = bool("Mike"); /*d is now true*/
d = bool(""); /*d is now false*/
```

Objects cannot be cast to or from other data types.

**Terminate**

When the Terminate keyword statement is found within the program, the program ends.

```plaintext
main() {
    init{
        int a = 5;
    }
    always{
        Terminate;
        a = 2; /*program ends before this statement is executed*/
    }
}
```

**Delays**

In Slang, you can delay a block of code for a designated number of simulation time units by doing #time, where time is a float or a variable. This will delay only the current init or always block, and the other blocks will execute as they would have before.

```plaintext
main() {
    init{
        #12 Terminate;
    }
}
```

**Return**

The return keyword accepts an expression, function call, or nothing, and exists out of the smallest containing function or calling block.

**Pass by Value and Pass by Reference**

Slang passes arguments by value. The argument sent to a function is in fact a copy of the original argument sent to the function. In this way the function can not modify the argument originally sent to it. The only exceptions to this are arrays, objects, and functions, which are passed by reference.

**Overloading**

Slang does not allow overloading of functions, or of identifiers (there can be no int x at the same time as object x, for example).

**Keywords**
Our keywords are:

```
if else for while return int float bool string object void func main init always true false Ter
minute
```

These words have special meanings and are reserved, so the user may not use them as an identifier.

**Threads**

An Aslang program consists of threads, specified by the `init` and `always` keywords. An `init` thread is:

```
init{
  body
}
```

And an `always` thread is:

```
always{
  body
}
```

An `init` block of code is executed a single time at the beginning of the program, setting up any conditions necessary for execution. The body of an `init` thread can be empty. An `always` thread executes once per time unit, looping consistently until the program terminates. `Always` blocks run as separate threads, and therefore it is possible to run multiple `always` threads concurrently.

**Program Structure**

The program consists of function declarations followed by a `main`, which contains 'always' and/or 'init' threads. All threads execute concurrently and have access to the data structures and variables defined in other threads. Example:

```
1         func foo(){
2                #2 2+2;
3         }
4
5         func bar(){
6                #5 1+1;
7         }
8
9         main(){
10                init {
11                        #20 Terminate;
12                }
13                always { //always_1
14                        #3 foo(); //because of the delay of 2 within foo(), this thre
ad sleeps for another 2 time units
15                }
16                always { //always_2
17                        #10 bar(); //because of the delay of 5 within bar(), this thr
ead sleeps for another 5 time units
18                }
19        }
20        }
```

On an absolute time scale from the beginning of the execution of main:

<table>
<thead>
<tr>
<th>Time</th>
<th>Lines executed</th>
<th>Details on actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:</td>
<td>11, 14, 18</td>
<td>sleep each thread for the time units specified</td>
</tr>
<tr>
<td>03:</td>
<td>14, 2</td>
<td>foo() is called in always_1, which sleeps the the thread for another 2 time units</td>
</tr>
</tbody>
</table>
Because threads can wake and access data structures at the same time, there are race condition concerns. In the case of race conditions, Slang does not guarantee a particular behavior.

*Because the order in which threads execute events are undefined, line 14 terminating the program might occur before or after any of the other lines in other threads.

**Grammar**

In the grammar below, words in all caps are tokens passed in from the lexer.

```plaintext
program:      main | fdecl program
main:        MAIN() { vdecl_list timeblock_list }

timeblock_list: /* nothing */ | timeblock_list timeblock

timeblock:    INIT { stmt_list } | ALWAYS { stmt_list }

fdecl:        FUNC TYPE ID ( formals_opt ) { stmt_list }

formals_opt:  /* nothing */ | formal_list

formal_list:  param | formal_list, param

param:        TYPE ID | OBJECT ID | TYPE ID []

stmt_list:    /* nothing */ | stmt_list stmt
```
delay:
  | INT_LITERAL
  | FLOAT_LITERAL
  | ID

donode_list:
  /* nothing */
  | donode; donode_list

donode:
  TYPE ID
  | TYPE ID = expr
  | TYPE ID [ ]
  | TYPE ID [ ] = [ expr_list ]
  | OBJECT ID
  | OBJECT ID = OBJECT(property_list)

property_list:
  /* nothing */
  | property, property_list

property:
  TYPE ID
  | TYPE ID [ ]
  | TYPE ID COLON expr
  | TYPE ID [ ] = [ expr_list ]

expr_list:
  /* nothing */
  | expr, expr_list

doten:
  expr;
  | #delay doten
  | Terminate;
  | return expr;
  | { doten_list }
  | If ( expr ) doten
  | If ( expr ) doten else doten
  | For ( expr_opt; expr_opt; expr_opt ) doten
  | While ( expr ) doten
  | vdecl;
  | ID.ID = expr;

eexpr_list:
  /* nothing */
  | expr, expr_list

expr:
  INT_LITERAL
  | FLOAT_LITERAL
  | STRING_LITERAL
  | BOOL_LITERAL
  | TYPE(expr)
  | ID
  | expr + expr
  | expr - expr
  | expr * expr
  | expr / expr
  | expr == expr
  | expr != expr
  | expr < expr
  | expr <= expr
  | expr > expr
  | expr >= expr
  | expr % expr
  | ID = expr
  | ID[INT_LITERAL] = expr
  | ID[INT_LITERAL]
  | (expr)
  | -expr
  | expr++
| expr-- |
| expr & expr |
| expr | expr |