Specification of Source §4 GPU—2021 edition

Martin Henz, Rahul Rajesh, Zhang Yuntong

National University of Singapore
School of Computing

April 30, 2020

The language Source is the official language of the textbook *Structure and Interpretation of Computer Programs*, JavaScript Adaptation. Source is a sublanguage of ECMAScript 2018 (9th Edition) and defined in the documents titled "Source §x", where x refers to the respective textbook chapter.

Changes

Source §4 GPU allows for Source programs to be accelerated on the GPU if certain conditions are met. The exact specifications for this is outlined on page 13. Source §4 GPU defines a formal specification to identify areas in the program that are embarrssingly parallel (e.g. for loops etc.) . These will then be run in parallel across GPU threads. Experimentation has shown that Source §4 GPU is orders of magnitude faster than Source §4 for heavy CPU bound tasks (matrix multiplication of large matrices)

Programs

A Source program is a *program*, defined using Backus-Naur Form[1] as follows:

[1] We adopt Henry Ledgard's BNF variant that he described in *A human engineered variant of BNF*, ACM SIGPLAN Notices, Volume 15 Issue 10, October 1980, Pages 57-62. In our grammars, we use **bold** font for keywords, *italics* for syntactic variables, ϵ for nothing, $x \mid y$ for x or y, and $x \ldots$ for zero or more repetitions of x.

```
        program  ::=  statement ...                                 statement sequence
      statement  ::=  const name = expression ;                     constant declaration
                   |  let ;                                         variable declaration
                   |  assignment ;                                  variable assignment
                   |  expression[expression] = expression ;         array assignment
                   |  function name ( parameters ) block            function declaration
                   |  return expression ;                           return statement
                   |  if-statement                                  conditional statement
                   |  while ( expression ) block                    while loop
                   |  for ( ( assignment | let );
                           expression ;
                           assignment ) block                       for loop
                   |  break ;                                       break statement
                   |  continue ;                                    continue statement
                   |  block                                         block statement
                   |  expression ;                                  expression statement
     parameters  ::=  ϵ | name ( , name ) ...                       function parameters
   if-statement  ::=  if ( expression ) block
                      else ( block | if-statement )                 conditional statement
          block  ::=  { program }                                   block statement
            let  ::=  let name = expression                         variable declaration
     assignment  ::=  name = expression                             variable assignment
     expression  ::=  number                                        primitive number expression
                   |  true | false                                  primitive boolean expression
                   |  null                                          primitive list expression
                   |  string                                        primitive string expression
                   |  name                                          name expression
                   |  expression binary-operator expression         binary operator combination
                   |  unary-operator expression                     unary operator combination
                   |  expression ( expressions )                    function application
                   |  ( name | ( parameters ) ) => expression       function definition (expr. body)
                   |  ( name | ( parameters ) ) => block            function definition (block body)
                   |  expression ? expression : expression          conditional expression
                   |  expression[expression]                        array access
                   |  [ expressions ]                               literal array expression
                   |  ( expression )                                parenthesised expression
binary-operator  ::=  + | - | * | / | % | === | !==
                   |  > | < | >= | <= | && | ||                     binary operator
 unary-operator  ::=  ! | -                                         unary operator
    expressions  ::=  ϵ | expression ( , expression ) ...           argument expressions
```

Binary boolean operators

Conjunction

expression$_1$ **&&** *expression*$_2$

stands for

expression$_1$ **?** *expression*$_2$ **: false**

Disjunction

expression$_1$ **||** *expression*$_2$

stands for

expression$_1$ **? true :** *expression*$_2$

Loops

while-loops

While loops are seen as abbreviations for function applications as follows:

while (*expression* **)** *block*

stands for

```
function _body() { block }
_while( () => expression , _body );
```

where `_while` is defined as follows:

```
function _while(test, body) {
    if (test()) {
        body();
        _while(test, body);
    } else {
        undefined;
    }
}
```

Simple for-loops

for (*assignment*$_1$ **;** *expression* **;** *assignment*$_2$ **)** *block*

stands for

```
assignment1
while (expression) {
    block
    assignment2
}
```

for-loops with loop control variable

for (let *name* = *expression*$_1$ **;** *expression*$_2$ **;** *assignment* **)** *block*

stands for

```
{
    let name = expression_1;
    for (name = name; expression_2; assignment) {
        const _copy_of_name = name;
        {
            const name = _copy_of_name;
            block
        }
    }
}
```

Restrictions

- Return statements are only allowed in bodies of functions.
- Return statements are not allowed in the bodies of while and for loops.
- There cannot be any newline character between **`return`** and *expression* in return statements.
- There cannot be any newline character between (*name* | **`(`** *parameters* **`)`**) and **`=>`** in function definition expressions.
- Local functions within an outer function must precede all other statements in body of the outer function.

Names

Names[2] start with _, `$` or a letter[3] and contain only _, `$`, letters or digits[4]. Reserved words[5] such as keywords are not allowed as names.
Valid names are `x`, `_45`, `$$` and π, but always keep in mind that programming is communicating and that the familiarity of the audience with the characters used in names is an important aspect of program readability.
In addition to names that are declared using **`const`**, **`function`**, **`=>`** (and **`let`** in Source §3 and 4), the following names refer to primitive functions and constants:

- `math_`*name*, where *name* is any name specified in the JavaScript `Math` library, see ECMAScript Specification, Section 20.2. Examples:
  - `math_PI`: Refers to the mathematical constant π,
  - `math_sqrt(n)`: Returns the square root of the *number* `n`.
- `runtime()`: Returns number of milliseconds elapsed since January 1, 1970 00:00:00 UTC
- `parse_int(s, i)`: interprets the *string* `s` as an integer, using the positive integer `i` as radix, and returns the respective value, see ECMAScript Specification, Section 18.2.5.
- `undefined`, `NaN`, `Infinity`: Refer to JavaScript's undefined, NaN ("Not a Number") and Infinity values, respectively.
- `is_boolean(x)`, `is_number(x)`, `is_string(x)`, `is_function(x)`: return `true` if the type of `x` matches the function name and `false` if it does not. Following JavaScript, we specify that `is_number` returns `true` for `NaN` and `Infinity`.

---

[2] In ECMAScript 2018 (9th Edition), these names are called *identifiers*.
[3] By *letter* we mean Unicode letters (L) or letter numbers (Nl).
[4] By *digit* we mean characters in the Unicode categories Nd (including the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9), Mn, Mc and Pc.
[5] By *Reserved word* we mean any of: **`break`**, **`case`**, **`catch`**, **`continue`**, **`debugger`**, **`default`**, **`delete`**, **`do`**, **`else`**, **`finally`**, **`for`**, **`function`**, **`if`**, **`in`**, **`instanceof`**, **`new`**, **`return`**, **`switch`**, **`this`**, **`throw`**, **`try`**, **`typeof`**, **`var`**, **`void`**, **`while`**, **`with`**, **`class`**, **`const`**, **`enum`**, **`export`**, **`extends`**, **`import`**, **`super`**, **`implements`**, **`interface`**, **`let`**, **`package`**, **`private`**, **`protected`**, **`public`**, **`static`**, **`yield`**, **`null`**, **`true`**, **`false`**.

- `prompt(s)`: Pops up a window that displays the *string* `s`, provides an input line for the user to enter a text, a "Cancel" button and an "OK" button. The call of `prompt` suspends execution of the program until one of the two buttons is pressed. If the "OK" button is pressed, `prompt` returns the entered text as a string. If the "Cancel" button is pressed, `prompt` returns a non-string value.
- `display(x)`: Displays the value `x` in the console[6]; returns the argument `a`.
- `display(x, s)`: Displays the string `s`, followed by a space character, followed by the value `x` in the console[6]; returns the argument `x`.
- `error(x)`: Displays the value `x` in the console[6] with error flag. The evaluation of any call of `error` aborts the running program immediately.
- `error(x, s)`: Displays the string `s`, followed by a space character, followed by the value `x` in the console[6] with error flag. The evaluation of any call of `error` aborts the running program immediately.
- `stringify(x)`: returns a string that represents[6] the value `x`.

All Source primitive functions, except `stringify`, can be assumed to run in $O(1)$ time, except `display`, `error` and `stringify`, which run in $O(n)$ time, where n is the size (number of components such as pairs) of their argument.

List Support

The following list processing functions are supported:

- `pair(x, y)`: *primitive*, makes a pair from `x` and `y`.
- `is_pair(x)`: *primitive*, returns `true` if `x` is a pair and `false` otherwise.
- `head(x)`: *primitive*, returns the head (first component) of the pair `x`.
- `tail(x)`: *primitive*, returns the tail (second component) of the pair `x`.
- `is_null(xs)`: *primitive*, returns `true` if `xs` is the empty list `null`, and `false` otherwise.
- `is_list(x)`: Returns `true` if `x` is a list as defined in the lectures, and `false` otherwise. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of the chain of `tail` operations that can be applied to `x`.
- `list(x1, x2,..., xn)`: *primitive*, returns a list with n elements. The first element is `x1`, the second `x2`, etc. Iterative process; time: $O(n)$, space: $O(n)$, since the constructed list data structure consists of n pairs, each of which takes up a constant amount of space.
- `draw_data(x)`: *primitive*, visualizes `x` in a separate drawing area in the Source Academy using a box-and-pointer diagram; time, space: $O(n)$, where n is the number of data structures such as pairs in `x`.
- `equal(x1, x2)`: Returns `true` if both have the same structure with respect to `pair`, and the same numbers, boolean values, functions or empty list at corresponding leave positions (places that are not themselves pairs), and `false` otherwise; time, space: $O(n)$, where n is the number of pairs in `x`.
- `length(xs)`: Returns the length of the list `xs`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `map(f, xs)`: Returns a list that results from list `xs` by element-wise application of `f`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `build_list(n, f)`: Makes a list with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`. Recursive process; time: $O(n)$, space: $O(n)$.

[6] The notation used for the display of values is consistent with JSON, but also displays `undefined` and function objects.

- `for_each(f, xs)`: Applies `f` to every element of the list `xs`, and then returns `true`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `list_to_string(xs)`: Returns a string that represents list `xs` using the text-based box-and-pointer notation `[...]`.
- `reverse(xs)`: Returns list `xs` in reverse order. Iterative process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`. The process is iterative, but consumes space $O(n)$ because of the result list.
- `append(xs, ys)`: Returns a list that results from appending the list `ys` to the list `xs`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `member(x, xs)`: Returns first postfix sublist whose head is identical to `x` (===); returns `[]` if the element does not occur in the list. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `remove(x, xs)`: Returns a list that results from `xs` by removing the first item from `xs` that is identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `remove_all(x, xs)`: Returns a list that results from `xs` by removing all items from `xs` that are identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `filter(pred, xs)`: Returns a list that contains only those elements for which the one-argument function `pred` returns `true`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `enum_list(start, end)`: Returns a list that enumerates numbers starting from `start` using a step size of 1, until the number exceeds (>) `end`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `list_ref(xs, n)`: Returns the element of list `xs` at position `n`, where the first element has index 0. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `accumulate(op, initial, xs)`: Applies binary function `op` to the elements of `xs` from right-to-left order, first applying `op` to the last element and the value `initial`, resulting in r_1, then to the second-last element and r_1, resulting in r_2, etc, and finally to the first element and r_{n-1}, where n is the length of the list. Thus, `accumulate(op,zero,list(1,2,3))` results in `op(1, op(2, op(3, zero)))`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`, assuming `op` takes constant time.

Pair Mutators

The following pair mutator functions are supported:

- `set_head(p, x)`: *primitive*, changes the pair `p` such that its head is `x`. Returns `undefined`.
- `set_tail(p, x)`: *primitive*, changes the pair `p` such that its tail is `x`. Returns `undefined`.

Array Support

The following array processing functions are supported:

- `array_length(x)`: *primitive*, returns the current length of array `x`, which is 1 plus the highest index `i` that has been used so far in an array assignment on `x`.
- `is_array(x)`: *primitive*, returns returns `true` if `x` is an array, and `false` if it is not.

Stream Support

The following stream processing functions are supported:

- `stream_tail(x)`: *Built-in*, assumes that the tail (second component) of the pair `x` is a nullary function, and returns the result of applying that function.
  Laziness: Yes: `stream_tail` only forces the direct tail of a given stream, but not the rest of the stream, i.e. not the tail of the tail, etc.
- `stream(x1, x2,..., xn)`: *Built-in*, returns a stream with n elements. The first element is `x1`, the second `x2`, etc.
  Laziness: No: In this implementation, we generate first a complete list, and then a stream using `list_to_stream`.
- `is_stream(x)`: Returns `true` if `x` is a stream as defined in the lectures, and `false` otherwise.
  Laziness: No: `is_stream` needs to force the given stream.
- `list_to_stream(xs)`: transforms a given list to a stream.
  Laziness: Yes: `list_to_stream` goes down the list only when forced.
- `stream_to_list(s)`: transforms a given stream to a list.
  Laziness: No: `stream_to_list` needs to force the whole stream.
- `stream_length(s)`: Returns the length of the stream `s`.
  Laziness: No: The function needs to force the whole stream.
- `stream_map(f, s)`: Returns a stream that results from stream `s` by element-wise application of `f`.
  Laziness: Yes: The argument stream is only explored as forced by the result stream.
- `build_stream(n, f)`: Makes a stream with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`.
  Laziness: Yes: The result stream forces the applications of fun for the next element.
- `stream_for_each(f, s)`: Applies `f` to every element of the stream `s`, and then returns `true`.
  Laziness: No: `stream_for_each` forces the exploration of the entire stream.
- `stream_reverse(s)`: Returns finite stream `s` in reverse order. Does not terminate for infinite streams.
  Laziness: No: `stream_reverse` forces the exploration of the entire stream.
- `stream_append(xs, ys)`: Returns a stream that results from appending the stream `ys` to the stream `xs`.
  Laziness: Yes: Forcing the result stream activates the actual append operation.
- `stream_member(x, s)`: Returns first postfix substream whose head is equal to `x` (===); returns `null` if the element does not occur in the stream.
  Laziness: Sort-of: `stream_member` forces the stream only until the element is found.
- `stream_remove(x, s)`: Returns a stream that results from given stream `s` by removing the first item from `s` that is equal (===) to `x`. Returns the original list if there is no occurrence.
  Laziness: Yes: Forcing the result stream leads to construction of each next element.
- `stream_remove_all(x, s)`: Returns a stream that results from given stream `s` by removing all items from `s` that are equal (===) to `x`.
  Laziness: Yes: The result stream forces the construction of each next element.
- `stream_filter(pred, s)`: Returns a stream that contains only those elements for which the one-argument function `pred` returns `true`.
  Laziness: Yes: The result stream forces the construction of each next element. Of course, the construction of the next element needs to go down the stream until an element is found for which `pred` holds.

- `enum_stream(start, end)`: Returns a stream that enumerates numbers starting from `start` using a step size of 1, until the number exceeds (>) `end`.
  Laziness: Yes: Forcing the result stream leads to the construction of each next element.
- `integers_from(n)`: Constructs an infinite stream of integers starting at a given number `n`.
  Laziness: Yes: Forcing the result stream leads to the construction of each next element.
- `eval_stream(s, n)`: Constructs the list of the first `n` elements of a given stream `s`.
  Laziness: Sort-of: `eval_stream` only forces the computation of the first `n` elements, and leaves the rest of the stream untouched.
- `stream_ref(s, n)`: Returns the element of stream `s` at position `n`, where the first element has index 0.
  Laziness: Sort-of: `stream_ref` only forces the computation of the first `n` elements, and leaves the rest of the stream untouched.

Numbers

We use decimal notation for numbers, with an optional decimal dot. "Scientific notation" (multiplying the number with 10^x) is indicated with the letter `e`, followed by the exponent x. Examples for numbers are `5432`, `-5432.109`, and `-43.21e-45`.

Strings

Strings are of the form `"`*double-quote-characters*`"`, where *double-quote-characters* is a possibly empty sequence of characters without the character `"`, and of the form `'`*single-quote-characters*`'`, where *single-quote-characters* is a possibly empty sequence of characters without the character `'`,

Arrays

Arrays in Source are created using literal array expressions:

```
let my_array_1 = [];
let my_array_2 = [42, 71, 13];
```

Arrays in Source are limited to integers as keys. In statements like

```
a[i];
a[j] = v;
```

the values `i` and `j` must be integers if `a` is an array.
Note that pairs in Source are represented by arrays with two elements. Therefore,

```
is_pair([1, 2]);
```

and

```
equal(pair(1, 2), [1, 2]);
```

evaluate to `true`.
Access of an array with an integer index to which no prior assignment has been made on the array, returns `undefined`.

Typing

Expressions evaluate to numbers, boolean values, strings or function values.
Only function values can be applied using the syntax:

expression ::= *name* **(** *expressions* **)**

The following table specifies what arguments Source's operators take and what results they return.

operator	argument 1	argument 2	result
+	number	number	number
+	string	string	string
-	number	number	number
*****	number	number	number
/	number	number	number
%	number	number	number
===	any	any	bool
!==	any	any	bool
>	number	number	bool
>	string	string	bool
<	number	number	bool
<	string	string	bool
>=	number	number	bool
>=	string	string	bool
<=	number	number	bool
<=	string	string	bool
&&	bool	any	any
||	bool	any	any
!	bool		bool
-	number		number

Preceding **`?`** and following **`if`**, Source only allows boolean expressions.
Only numbers are allowed in the key of member expressions of arrays (**`arr[key]`**).

Interpreter Support

- `apply_in_underlying_javascript(f, xs)`: *primitive*, calls the function `f` with arguments `xs`. For example:

```
function times(x, y) {
   return x * y;
}
apply_in_underlying_javascript(times, list(2, 3)); // returns 6
```

- `parse(x)`: *primitive*, returns the parse tree that results from parsing the string `x` as a Source program. The following two pages describe the shape of the parse tree.

```
     program  ::= statement ...                          list("sequence", list of <statement>)
   statement  ::= const name = expression ;              list("constant_declaration", <name>, <expression>)
                | let ;                                  see below
                | assignment ;                           see below
                | expression[expression] = expression ;  list("array_assignment", <expression>, <expression>)
                | function name ( parameters ) block     treat as : const name = parameters => block;
                | return expression ;                    list("return_statement", <expression>)
                | if-statement                           see below
                | while ( expression ) block             list("while_loop", <expression>, <statement>)
                | for ( ( assignment | let );
                        expression ;
                        assignment ) block               list("for_loop", <statement>, <expression>, <statement>,
                                                              <statement>)
                | break ;                                list("break_statement")
                | continue ;                             list("continue_statement")
                | block                                  see below
                | expression ;                           see below
  parameters  ::= ϵ | name ( , name ) ...                list of <name>
if-statement  ::= if ( expression ) block
                  else ( block | if-statement )          list("conditional_statement", <expression>,
                                                             <statement>, <statement>)
       block  ::= { program }                            list("block", <statement>)
         let  ::= let name = expression                  list("variable_declaration", <name>, <expression>)
  assignment  ::= name = expression                      list("assignment", <name>, <expression>)
```

<table>
<tr><td>expression ::=</td><td>number</td><td>self-evaluating</td></tr>
<tr><td>|</td><td>true | false</td><td>self-evaluating</td></tr>
<tr><td>|</td><td>null</td><td>self-evaluating</td></tr>
<tr><td>|</td><td>string</td><td>self-evaluating</td></tr>
<tr><td>|</td><td>name</td><td>list("name", string) or list("name", string, location)</td></tr>
<tr><td>|</td><td>expression binary-operator expression</td><td>list("application", <name>, list of <expression>)</td></tr>
<tr><td>|</td><td>unary-operator expression</td><td>list("application", <name>, list of <expression>)</td></tr>
<tr><td>|</td><td>expression (expressions)</td><td>list("application", <expression>, list of <expression>)</td></tr>
<tr><td>|</td><td>(name | (parameters)) => expression</td><td>list("function_definition", <parameters>,
list("return_statement", <expression>))</td></tr>
<tr><td>|</td><td>(name | (parameters)) => block</td><td>list("function_definition", <parameters>, <statement>)</td></tr>
<tr><td>|</td><td>expression ? expression : expression</td><td>list("conditional_expression", <expression>,
<expression>, <expression>)</td></tr>
<tr><td>|</td><td>expression[expression]</td><td>list("array_access", <expression>, <expression>)</td></tr>
<tr><td>|</td><td>[expressions]</td><td>list("array_expression", list of <expression>)</td></tr>
<tr><td>|</td><td>(expression)</td><td>treat as: expression</td></tr>
<tr><td>binary-operator ::=</td><td>+ | - | * | / | % | === | !==</td><td></td></tr>
<tr><td>|</td><td>> | < | >= | <= | && | ||</td><td>list("name", string)</td></tr>
<tr><td>unary-operator ::=</td><td>! | -</td><td>list("name", string)</td></tr>
<tr><td>expressions ::=</td><td>ϵ | expression (, expression) ...</td><td>list of <expression></td></tr>
</table>

Comments

In Source, any sequence of characters between "`/*`" and the next "`*/`" is ignored.
After "`//`" any characters until the next newline character is ignored.

Deviations from JavaScript

We intend the Source language to be a conservative extension of JavaScript: Every correct Source program should behave *exactly* the same using a Source implementation, as it does using a JavaScript implementation. We assume, of course, that suitable libraries are used by the JavaScript implementation, to account for the predefined names of each Source language.
This section lists some exceptions where we think a Source implementation should be allowed to deviate from the JavaScript specification, for the sake of internal consistency and esthetics.

Empty block as last statement of toplevel sequence: In JavaScript, empty blocks as last statement of a sequence are apparently ignored. Thus the result of evaluating such a sequence is the result of evaluating the previous statement. Implementations of Source might stick to the more intuitive result: `undefined`. Example:

```
1;
{
  // empty block
}
```

The result of evaluating this program can be `undefined` for implementations of Source. Note that this issue only arises at the toplevel—outside of functions.

GPU Acceleration

This section outlines the specifications for programs to be accelerated using the GPU.

```
          gpu_statement ::= for ( gpu_for_let;
                                  gpu_condition;
                                  gpu_for_assignment ) gpu_block;
            gpu_for_let ::= let name = 0 ;
      gpu_for_condition ::= name < (number | name) ;
                          | name <= (number | name) ;
     gpu_for_assignment ::= name = name + 1 ;
              gpu_block ::= { gpu_statement } | { core_statements }
        core_statements ::= core_statement... gpu_result_assignment
  gpu_result_assignment ::= gpu_access[gpu_name] = gpu_result ;
             gpu_access ::= name | gpu_access[gpu_name]
             gpu_result ::= number | [ gpu_result... ]
         core_statement ::= const name = gpu_expression ;
                          | gpu_let ;
                          | gpu_assignment ;
                          | gpu_expression[gpu_expression] = gpu_expression ;
                          | while( gpu_expression ) gpu_block
                          | for ( ( gpu_assignment | gpu_let );
                                  gpu_expression;
                                  gpu_assignment ) gpu_block
         gpu_assignment ::= name = gpu_expression
                gpu_let ::= let name = gpu_expression
         gpu_expression ::= number
                          | true | false
                          | null
                          | name
                          | string
                          | gpu_expression binary_operator gpu_expression
                          | unary_operator gpu_expression
                          | gpu_function( gpu_expressions )
                          | gpu_expression ? gpu_expression : gpu_expression
                          | gpu_expression [ gpu_expression ]
                          | [ gpu_expressions ]
                          | ( gpu_expression )
        gpu_expressions ::= ϵ | gpu_expression ( , gpu_expression ) ...
```

Restrictions

Even if the BNF syntax is met, GPU acceleration can only take place if all the restrictions below are satisfied. If all criteria are met, the *gpu_statement* loops are embarrassingly parallel.

Special For Loops

In the BNF, we have special loops that take on this form:

`for (` *gpu_for_let*`;`
gpu_condition`;`
gpu_for_assignment **`)`**

These are the loops that will be taken into consideration for parallelization. However, on top of the BNF syntax, the below requirement must also be statisfied:

- the names declared in each *gpu_for_let* have to be different across the loops
- in each loop, the *gpu_condition* and the *gpu_for_assignment* must use the name declared in the respective *gpu_for_let* statement

GPU Function

A *gpu_function* has to be a *math_* $*$ function

Core Statement

Within *core_statement*, there are some constraints:

- no assignment to any global variables (all assignments can only be done to variables defined in the *gpu_block*)
- no use of the variable in *gpu_result_assignment* at an offset from the current index e.g. cannot be i - 1

GPU Result Statement

The *gpu_result_assignment* is the statement that stores a value calculated in core statements into a result array. It access an array at a certain coordinate e.g. $array[i_1][i_2][i_3]$. For this:

- This result array has to be defined outside the *gpu_block*.

- The sequence of coordinates which we access in the result array $i_1, i_2, i_3 ... i_k$ must be a prefix of the special for loop counters $[c_1, c_2 ... c_n]$.
- If you have n special for loops, the array expression can take on k coordinates where $0 < k \leq n$. The order matters as well, it has to follow the same order as the special for loops: you cannot have $name[c_2][c_1]$.

Examples

Below are some examples of valid and invalid source gpu programs:

Valid - Using first loop counter. (meaning the loop will be run across N threads; the first loop is parallelized away):

```
for (let i = 0; i < N; i = i + 1) {
    for (let k = 0; k < M; k = k + 1) {
        res[i] = arr[k % 2] + 1;
    }
}
```

Invalid - Counter used is not a prefix of for loop counters:

```
for (let i = 0; i < N; i = i + 1) {
    for (let k = 0; k < M; k = k + 1) {
        res[k] = arr[i % 2] + 1;
    }
}
```

Valid - Using first three loop counters (meaning the loop will be run across N*M*C threads, if available):

```
for (let i = 0; i < N; i = i + 1) {
    for (let j = 0; j < M; j = j + 1) {
        for (let k = 0; k < C; k = k + 1) {
            let x = math_pow(2, 10);
            let y = x * (1000);
            arr[i][j][k] = (x + y * 2);
        }
    }
}
```

Invalid - Indices are in wrong order (must respect for loop counter orders):

```
for (let i = 0; i < N; i = i + 1) {
    for (let j = 0; j < M; j = j + 1) {
        for (let k = 0; k < C; k = k + 1) {
            let x = math_pow(2, 10);
```

```
            let y = x * (1000);
            res[k][j][i] = (x + y * 2);
        }
    }
}
```

Invalid - Using an index that is not part of a special for loop (see above):

```
for (let i = 0; i < N; i = i + 1) {
    for (let j = 0; j < M; j = j + 1) {
        for (let k = 1; k < C; k = k + 2) {
            res[k] = arr1[i] + arr2[j];
        }
    }
}
```

Appendix: List library

Those list library functions that are not primitive functions are pre-declared as follows:

```
// list.js START


/**
 * makes a pair whose head (first component) is <CODE>x</CODE>
 * and whose tail (second component) is <CODE>y</CODE>.
 * @param {value} x - given head
 * @param {value} y - given tail
 * @returns {pair} pair with <CODE>x</CODE> as head and <CODE>y</CODE> as tail.
 */
function pair(x, y) {}

/**
 * returns <CODE>true</CODE> if <CODE>x</CODE> is a
 * pair and false otherwise.
 * @param {value} x - given value
 * @returns {boolean} whether <CODE>x</CODE> is a pair
 */
function is_pair(x) {}

/**
 * returns head (first component) of given pair <CODE>p</CODE>
 * @param {pair} p - given pair
 * @returns {value} head of <CODE>p</CODE>
 */
function head(p) {}

/**
 * returns tail (second component of given pair <CODE>p</CODE>
 * @param {pair} p - given pair
 * @returns {value} tail of <CODE>p</CODE>
 */
function tail(p) {}

/**
 * returns <CODE>true</CODE> if <CODE>x</CODE> is the
 * empty list <CODE>null</CODE>, and <CODE>false</CODE> otherwise.
 * @param {value} x - given value
 * @returns {boolean} whether <CODE>x</CODE> is <CODE>null</CODE>
 */
function is_null(x) {}
```

```
/**
 * Returns <CODE>true</CODE> if
 * <CODE>xs</CODE> is a list as defined in the textbook, and
 * <CODE>false</CODE> otherwise. Iterative process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(1)</CODE>, where <CODE>n</CODE>
 * is the length of the
 * chain of <CODE>tail</CODE> operations that can be applied to <CODE>xs</CODE>.
 * recurses down the list and checks that it ends with the empty list null
 * @param {value} xs - given candidate
 * @returns whether {xs} is a list
 */
function is_list(xs) {
    return is_null(xs) || (is_pair(xs) && is_list(tail(xs)));
}

/**
 * Given <CODE>n</CODE> values, returns a list of length <CODE>n</CODE>.
 * The elements of the list are the given values in the given order.
 * @param {value} value1,value2,...,value_n - given values
 * @returns {list} list containing all values
 */
function list(value1, value2, ...values ) {}

/**
 * visualizes <CODE>x</CODE> in a separate drawing
 * area in the Source Academy using a box-and-pointer diagram; time, space:
 * O(n), where n is the number of data structures such as
 * pairs in <CODE>x</CODE>.
 * @param {value} x - given value
 * @returns {value} given <CODE>x</CODE>
 */
function draw_data(x) {}

/**
 * Returns <CODE>true</CODE> if both
 * have the same structure with respect to <CODE>pair</CODE>,
 * and the same numbers, boolean values, functions or empty list
 * at corresponding leave positions (places that are not themselves pairs),
 * and <CODE>false</CODE> otherwise; time, space:
 * <CODE>O(n)</CODE>, where <CODE>n</CODE> is the number of pairs in
 * <CODE>x</CODE>.
 * @param {value} x - given value
 * @param {value} y - given value
 * @returns {boolean} whether <CODE>x</CODE> is structurally equal to <CODE>y</CO
```

```
 */
function equal(x, y) {
    return (is_pair(x) && is_pair(y))
        ? (equal(head(x), head(y)) &&
            equal(tail(x), tail(y)))
        : x === y;
}

/**
 * Returns the length of the list
 * <CODE>xs</CODE>.
 * Iterative process; time: <CODE>O(n)</CODE>, space:
 * <CODE>O(1)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * @param {list} xs - given list
 * @returns {number} length of <CODE>xs</CODE>
 */
function length(xs) {
    function iter(ys, acc) {
       return is_null(ys)
           ? acc
           : iter(tail(ys), acc + 1);
    }
    return iter(xs, 0);
}

/**
 * Returns a list that results from list
 * <CODE>xs</CODE> by element-wise application of unary function <CODE>f</CODE>.
 * Recursive process; time: <CODE>O(n)</CODE>,
 * space: <CODE>O(n)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE
 * <CODE>f</CODE> is applied element-by-element:
 * <CODE>map(f, list(1, 2))</CODE> results in <CODE>list(f(1), f(2))</CODE>.
 * @param {function} f - unary
 * @param {list} xs - given list
 * @returns {list} result of mapping
 */
function map(f, xs) {
    return is_null(xs)
        ? null
        : pair(f(head(xs)), map(f, tail(xs)));
}

/**
 * Makes a list with <CODE>n</CODE>
 * elements by applying the unary function <CODE>f</CODE>
```

```
 * to the numbers 0 to <CODE>n - 1</CODE>, assumed to be a non-negative integer.
 * Recursive process; time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>.
 * @param {number} n - given non-negative integer
 * @param {function} f - unary function
 * @returns {list} resulting list
 */
function build_list(n, f) {
    function build(i, f, already_built) {
        return i < 0
            ? already_built
            : build(i - 1, f, pair(f(i),
                already_built));
    }
    return build(n - 1, f, null);
}

/**
 * Applies unary function <CODE>f</CODE> to every
 * element of the list <CODE>xs</CODE>.
 * Iterative process; time: <CODE>O(n)</CODE>, space: <CODE>O(1)</CODE>,
 * Where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * <CODE>f</CODE> is applied element-by-element:
 * <CODE>for_each(fun, list(1, 2))</CODE> results in the calls
 * <CODE>fun(1)</CODE> and <CODE>fun(2)</CODE>.
 * @param {function} f - unary
 * @param {list} xs - given list
 * @returns {boolean} true
 */

function for_each(f, xs) {
    if (is_null(xs)) {
        return true;
    } else {
        f(head(xs));
        return for_each(f, tail(xs));
    }
}

/**
 * Returns a string that represents
 * list <CODE>xs</CODE> using the text-based box-and-pointer notation
 * <CODE>[...]</CODE>.
 * @param {list} xs - given list
 * @returns {string} <CODE>xs</CODE> converted to string
 */
```

```
function list_to_string(xs) {
    return is_null(xs)
        ? "null"
        : is_pair(xs)
            ? "[" + list_to_string(head(xs)) + "," +
                list_to_string(tail(xs)) + "]"
            : stringify(xs);
}

/**
 * Returns list <CODE>xs</CODE> in reverse
 * order. Iterative process; time: <CODE>O(n)</CODE>,
 * space: <CODE>O(n)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE
 * The process is iterative, but consumes space <CODE>O(n)</CODE>
 * because of the result list.
 * @param {list} xs - given list
 * @returns {list} <CODE>xs</CODE> in reverse
 */
function reverse(xs) {
    function rev(original, reversed) {
        return is_null(original)
            ? reversed
            : rev(tail(original),
                pair(head(original), reversed));
    }
    return rev(xs, null);
}

/**
 * Returns a list that results from
 * appending the list <CODE>ys</CODE> to the list <CODE>xs</CODE>.
 * Recursive process; time: <CODE>O(n)</CODE>, space:
 * <CODE>O(n)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * In the result, null at the end of the first argument list
 * is replaced by the second argument, regardless what the second
 * argument consists of.
 * @param {list} xs - given first list
 * @param {list} ys - given second list
 * @returns {list} result of appending <CODE>xs</CODE> and <CODE>ys</CODE>
 */
function append(xs, ys) {
    return is_null(xs)
        ? ys
        : pair(head(xs),
            append(tail(xs), ys));
```

```
}

/**
 * Returns first postfix sublist
 * whose head is identical to
 * <CODE>v</CODE> (using <CODE>===</CODE>); returns <CODE>null</CODE> if the
 * element does not occur in the list.
 * Iterative process; time: <CODE>O(n)</CODE>,
 * space: <CODE>O(1)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE
 * @param {value} v - given value
 * @param {list} xs - given list
 * @returns {list} postfix sublist that starts with <CODE>v</CODE>
 */
function member(v, xs) {
    return is_null(xs)
        ? null
        : (v === head(xs))
            ? xs
            : member(v, tail(xs));
}

/** Returns a list that results from
 * <CODE>xs</CODE> by removing the first item from <CODE>xs</CODE> that
 * is identical (<CODE>===</CODE>) to <CODE>v</CODE>.
 * Returns the original
 * list if there is no occurrence. Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>, where <CODE>n</CODE>
 * is the length of <CODE>xs</CODE>.
 * @param {value} v - given value
 * @param {list} xs - given list
 * @returns {list} <CODE>xs</CODE> with first occurrence of <CODE>v</CODE> remove
 */
function remove(v, xs) {
    return is_null(xs)
        ? null
        : v === head(xs)
            ? tail(xs)
            : pair(head(xs),
                remove(v, tail(xs)));
}

/**
 * Returns a list that results from
 * <CODE>xs</CODE> by removing all items from <CODE>xs</CODE> that
 * are identical (<CODE>===</CODE>) to <CODE>v</CODE>.
```

```
 * Returns the original
 * list if there is no occurrence.
 * Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>, where <CODE>n</CODE>
 * is the length of <CODE>xs</CODE>.
 * @param {value} v - given value
 * @param {list} xs - given list
 * @returns {list} <CODE>xs</CODE> with all occurrences of <CODE>v</CODE> removed
 */
function remove_all(v, xs) {
    return is_null(xs)
        ? null
        : v === head(xs)
            ? remove_all(v, tail(xs))
            : pair(head(xs),
                remove_all(v, tail(xs)));
}

/**
 * Returns a list that contains
 * only those elements for which the one-argument function
 * <CODE>pred</CODE>
 * returns <CODE>true</CODE>.
 * Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>,
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * @param {function} pred - unary function returning boolean value
 * @param {list} xs - given list
 * @returns {list} list with those elements of <CODE>xs</CODE> for which <CODE>pr
 */
function filter(pred, xs) {
    return is_null(xs)
        ? xs
        : pred(head(xs))
            ? pair(head(xs),
                filter(pred, tail(xs)))
            : filter(pred, tail(xs));
}

/**
 * Returns a list that enumerates
 * numbers starting from <CODE>start</CODE> using a step size of 1, until
 * the number exceeds (<CODE>&gt;</CODE>) <CODE>end</CODE>.
 * Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>,
```

```
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * @param {number} start - starting number
 * @param {number} end - ending number
 * @returns {list} list from <CODE>start</CODE> to <CODE>end</CODE>
 */
function enum_list(start, end) {
    return start > end
        ? null
        : pair(start,
            enum_list(start + 1, end));
}

/**
 * Returns the element
 * of list <CODE>xs</CODE> at position <CODE>n</CODE>,
 * where the first element has index 0.
 * Iterative process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(1)</CODE>,
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * @param {list} xs - given list
 * @param {number} n - given position
 * @returns {value} item in <CODE>xs</CODE> at position <CODE>n</CODE>
 */
function list_ref(xs, n) {
    return n === 0
        ? head(xs)
        : list_ref(tail(xs), n - 1);
}

/** Applies binary
 * function <CODE>f</CODE> to the elements of <CODE>xs</CODE> from
 * right-to-left order, first applying <CODE>f</CODE> to the last element
 * and the value <CODE>initial</CODE>, resulting in <CODE>r</CODE><SUB>1</SUB>,
 * then to the
 * second-last element and <CODE>r</CODE><SUB>1</SUB>, resulting in
 * <CODE>r</CODE><SUB>2</SUB>,
 * etc, and finally
 * to the first element and <CODE>r</CODE><SUB>n-1</SUB>, where
 * <CODE>n</CODE> is the length of the
 * list. Thus, <CODE>accumulate(f,zero,list(1,2,3))</CODE> results in
 * <CODE>f(1, f(2, f(3, zero)))</CODE>.
 * Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>,
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>
 * assuming <CODE>f</CODE> takes constant time.
```

```
 * @param {function} f - binary function
 * @param {value} initial - initial value
 * @param {list} xs - given list
 * @returns {value} result of accumulating <CODE>xs</CODE> using <CODE>f</CODE> s
 */
function accumulate(f, initial, xs) {
    return is_null(xs)
        ? initial
        : f(head(xs),
            accumulate(f, initial, tail(xs)));
}

//

// list.js END
```

Appendix: Stream library

Those stream library functions that are not primitive functions are pre-declared as follows:

```
// stream.js START


// Supporting streams in the Scheme style, following
// "stream discipline"

/**
 * assumes that the tail (second component) of the
 * pair {x} is a nullary function, and returns the result of
 * applying that function. Throws an exception if the argument
 * is not a pair, or if the tail is not a function.
 * Laziness: Yes: {stream_tail} only forces the direct tail
 * stream, but not the rest of the stream, i.e. not the tail
 * of the tail, etc.
 * @param {Stream} xs - given stream
 * @returns {Stream} result stream (if stream discipline is used)
 */

function stream_tail(xs) {
    if (is_pair(xs)) {
        const tail = head(xs);
        if (is_function(tail)) {
            return tail();
        } else {
            error(tail,
                  'stream_tail(xs) expects a function as ' +
                  'the tail of the argument pair xs, ' +
                  'but encountered ');
        }
    } else {
        error(xs, 'stream_tail(xs) expects a pair as ' +
              'argument xs, but encountered ');
    }
}

/**
 * Returns <CODE>true</CODE> if
 * <CODE>xs</CODE> is a stream as defined in the textbook, and
 * <CODE>false</CODE> otherwise. Iterative process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(1)</CODE>, where <CODE>n</CODE>
 * is the length of the
```

```
 * chain of <CODE>stream_tail</CODE> operations that can be applied to <CODE>xs</
 * recurses down the stream and checks that it ends with the empty stream null.
 * Laziness:  No: <CODE>is_stream</CODE> needs to force the given stream.
 * @param {value} xs - given candidate
 * @returns {boolean} whether <CODE>xs</CODE> is a stream
 */

function is_stream(xs) {
    return is_null(xs) || (is_pair(xs) && is_list(stream_tail(xs)));
}

/**
 * Given list <CODE>xs</CODE>, returns a stream of same length with
 * the same elements as <CODE>xs</CODE> in the same order.
 * Laziness:  Yes: <CODE>list_to_stream</CODE>
 * goes down the list only when forced.
 * @param {list} xs - given list
 * @returns {stream} stream containing all elements of <CODE>xs</CODE>
 */

function list_to_stream(xs) {
    return is_null(xs)
        ? null
        : pair(head(xs),
            () => list_to_stream(tail(xs)));
}

/**
 * Given stream <CODE>xs</CODE>, returns a list of same length with
 * the same elements as <CODE>xs</CODE> in the same order.
 * Laziness:  No: <CODE>stream_to_list</CODE> needs to force the whole
 * stream.
 * @param {stream} xs - stream
 * @returns {list} containing all elements of <CODE>xs</CODE>
 */

function stream_to_list(xs) {
    return is_null(xs)
        ? null
        : pair(head(xs), stream_to_list(stream_tail(xs)));
}



/**
```

```
 * Given <CODE>n</CODE> values, returns a stream of length <CODE>n</CODE>.
 * The elements of the stream are the given values in the given order.
 * Lazy? No: A
 * complete list is generated,
 * and then a stream using <CODE>list_to_stream</CODE> is generated from it.
 * @param {value} value1,value2,...,value_n - given values
 * @returns {stream} stream containing all values
 */

function stream() {
  var the_list = null
  for (var i = arguments.length - 1; i >= 0; i--) {
    the_list = pair(arguments[i], the_list)
  }
  return list_to_stream(the_list)
}

/**
 * Returns the length of the stream
 * <CODE>xs</CODE>.
 * Iterative process; time: <CODE>O(n)</CODE>, space:
 * <CODE>O(1)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * Lazy? No: The function needs to explore the whole stream
 * @param {stream} xs - given stream
 * @returns {number} length of <CODE>xs</CODE>
 */

function stream_length(xs) {
    return is_null(xs)
        ? 0
        : 1 + stream_length(stream_tail(xs));
}

/**
 * Returns a stream that results from stream
 * <CODE>xs</CODE> by element-wise application
 * of unary function <CODE>f</CODE>.
 * <CODE>f</CODE> is applied element-by-element:
 * <CODE>stream_map(f, stream(1,2))</CODE> results in
 * the same as <CODE>stream(f(1),f(2))</CODE>.
 * Lazy? Yes: The argument stream is only explored as forced by
 *            the result stream.
 * @param {function} f - unary
 * @param {stream} xs - given stream
 * @returns {stream} result of mapping
```

```
 */
function stream_map(f, s) {
    return is_null(s)
        ? null
        : pair(f(head(s)),
            () => stream_map(f, stream_tail(s)));
}

/**
 * Makes a stream with <CODE>n</CODE>
 * elements by applying the unary function <CODE>f</CODE>
 * to the numbers 0 to <CODE>n - 1</CODE>, assumed to be a non-negative integer.
 * Lazy? Yes: The result stream forces the application of <CODE>f</CODE>
 *           for the next element
 * @param {number} n - given non-negative integer
 * @param {function} f - unary function
 * @returns {stream} resulting stream
 */

function build_stream(n, fun) {
    function build(i) {
        return i >= n
            ? null
            : pair(fun(i),
                () => build(i + 1));
    }
    return build(0);
}


/**
 * Applies unary function <CODE>f</CODE> to every
 * element of the stream <CODE>xs</CODE>.
 * Iterative process; time: <CODE>O(n)</CODE>, space: <CODE>O(1)</CODE>,
 * Where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * <CODE>f</CODE> is applied element-by-element:
 * <CODE>stream_for_each(f, stream(1, 2))</CODE> results in the calls
 * <CODE>f(1)</CODE> and <CODE>f(2)</CODE>.
 * Lazy? No: <CODE>stream_for_each</CODE>
 * forces the exploration of the entire stream
 * @param {function} f - unary
 * @param {stream} xs - given stream
 * @returns {boolean} true
 */
```

```
function stream_for_each(fun, xs) {
    if (is_null(xs)) {
        return true;
    } else {
        fun(head(xs));
        return stream_for_each(fun, stream_tail(xs));
    }
}

/**
 * Returns stream <CODE>xs</CODE> in reverse
 * order. Iterative process; time: <CODE>O(n)</CODE>,
 * space: <CODE>O(n)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE
 * The process is iterative, but consumes space <CODE>O(n)</CODE>
 * because of the result stream.
 * Lazy? No: <CODE>stream_reverse</CODE>
 * forces the exploration of the entire stream
 * @param {stream} xs - given stream
 * @returns {stream} <CODE>xs</CODE> in reverse
 */

function stream_reverse(xs) {
    function rev(original, reversed) {
        return is_null(original)
            ? reversed
            : rev(stream_tail(original),
                pair(head(original), () => reversed));
    }
    return rev(xs, null);
}

/**
 * Returns a stream that results from
 * appending the stream <CODE>ys</CODE> to the stream <CODE>xs</CODE>.
 * In the result, null at the end of the first argument stream
 * is replaced by the second argument, regardless what the second
 * argument consists of.
 * Lazy? Yes: the result stream forces the actual append operation
 * @param {stream} xs - given first stream
 * @param {stream} ys - given second stream
 * @returns {stream} result of appending <CODE>xs</CODE> and <CODE>ys</CODE>
 */

function stream_append(xs, ys) {
    return is_null(xs)
```

```
        ? ys
        : pair(head(xs),
            () => stream_append(stream_tail(xs), ys));
}

/**
 * Returns first postfix substream
 * whose head is identical to
 * <CODE>v</CODE> (using <CODE>===</CODE>); returns <CODE>null</CODE> if the
 * element does not occur in the stream.
 * Iterative process; time: <CODE>O(n)</CODE>,
 * space: <CODE>O(1)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE
 * Lazy? Sort-of: <CODE>stream_member</CODE>
 * forces the stream only until the element
 * is found.
 * @param {value} v - given value
 * @param {stream} xs - given stream
 * @returns {stream} postfix substream that starts with <CODE>v</CODE>
 */

function stream_member(x, s) {
    return is_null(s)
        ? null
        : head(s) === x
            ? s
            : stream_member(x, stream_tail(s));
}

/** Returns a stream that results from
 * <CODE>xs</CODE> by removing the first item from <CODE>xs</CODE> that
 * is identical (<CODE>===</CODE>) to <CODE>v</CODE>.
 * Returns the original
 * stream if there is no occurrence.
 * Lazy? Yes: the result stream forces the construction of each next element
 * @param {value} v - given value
 * @param {stream} xs - given stream
 * @returns {stream} <CODE>xs</CODE> with first occurrence of <CODE>v</CODE> remo
 */

function stream_remove(v, xs) {
    return is_null(xs)
        ? null
        : v === head(xs)
            ? stream_tail(xs)
            : pair(head(xs),
```

```
                () => stream_remove(v, stream_tail(xs)));
}

/**
 * Returns a stream that results from
 * <CODE>xs</CODE> by removing all items from <CODE>xs</CODE> that
 * are identical (<CODE>===</CODE>) to <CODE>v</CODE>.
 * Returns the original
 * stream if there is no occurrence.
 * Recursive process.
 * Lazy? Yes: the result stream forces the construction of each next
 * element
 * @param {value} v - given value
 * @param {stream} xs - given stream
 * @returns {stream} <CODE>xs</CODE> with all occurrences of <CODE>v</CODE> remov
 */

function stream_remove_all(v, xs) {
    return is_null(xs)
        ? null
        : v === head(xs)
            ? stream_remove_all(v, stream_tail(xs))
            : pair(head(xs), () => stream_remove_all(v, stream_tail(xs)));
}

/**
 * Returns a stream that contains
 * only those elements of given stream <CODE>xs</CODE>
 * for which the one-argument function
 * <CODE>pred</CODE>
 * returns <CODE>true</CODE>.
 * Lazy? Yes: The result stream forces the construction of
 *            each next element. Of course, the construction
 *            of the next element needs to go down the stream
 *            until an element is found for which <CODE>pred</CODE> holds.
 * @param {function} pred - unary function returning boolean value
 * @param {stream} xs - given stream
 * @returns {stream} stream with those elements of <CODE>xs</CODE> for which <COD
 */

function stream_filter(p, s) {
    return is_null(s)
        ? null
        : p(head(s))
            ? pair(head(s),
```

```
                () => stream_filter(p, stream_tail(s)))
            : stream_filter(p, stream_tail(s));
}

/**
 * Returns a stream that enumerates
 * numbers starting from <CODE>start</CODE> using a step size of 1, until
 * the number exceeds (<CODE>&gt;</CODE>) <CODE>end</CODE>.
 * Lazy? Yes: The result stream forces the construction of
 *            each next element
 * @param {number} start - starting number
 * @param {number} end - ending number
 * @returns {stream} stream from <CODE>start</CODE> to <CODE>end</CODE>
 */

function enum_stream(start, end) {
    return start > end
        ? null
        : pair(start,
            () => enum_stream(start + 1, end));
}

/**
 * Returns infinite stream if integers starting
 * at given number <CODE>n</CODE> using a step size of 1.
 * Lazy? Yes: The result stream forces the construction of
 *            each next element
 * @param {number} start - starting number
 * @returns {stream} infinite stream from <CODE>n</CODE>
 */

function integers_from(n) {
    return pair(n,
        () => integers_from(n + 1));
}

/**
 * Constructs the list of the first <CODE>n</CODE> elements
 * of a given stream <CODE>s</CODE>
 * Lazy? Sort-of: <CODE>eval_stream</CODE> only forces the computation of
 * the first <CODE>n</CODE> elements, and leaves the rest of
 * the stream untouched.
 * @param {stream} s - starting number
 * @param {number} n - number of elements to place in result list
 * @returns {list} result list
```

```
 */

function eval_stream(s, n) {
    return n === 0
        ? null
        : pair(head(s),
            eval_stream(stream_tail(s),
                n - 1));
}

/**
 * Returns the element
 * of stream <CODE>xs</CODE> at position <CODE>n</CODE>,
 * where the first element has index 0.
 * Iterative process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(1)</CODE>,
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * Lazy? Sort-of: <CODE>stream_ref</CODE> only forces the computation of
 *                the first <CODE>n</CODE> elements, and leaves the rest of
 *                the stream untouched.
 * @param {stream} xs - given stream
 * @param {number} n - given position
 * @returns {value} item in <CODE>xs</CODE> at position <CODE>n</CODE>
 */

function stream_ref(s, n) {
    return n === 0
        ? head(s)
        : stream_ref(stream_tail(s), n - 1);
}

//

// stream.js END
```