Sheets

A High-Level Programming Language for writing GPU Accelerated Applications

Gabriel Blanco (gab2135)
Amelia Brunner (arb2196)
   Ben Barg (bbb2123)
   Ruchir Khaitan (rk2660)
Table of Contents

1. Introduction
2. Language Tutorial
   Simple Example
3. Language Manual
   Lexical Conventions
   Identifiers
   Comments
   Whitespace
   Types
   Fixed Point Type
   Floating Point Type
   Vector Types
   Operators
   Arithmetic Operators
   Relational Operators
   Assignment Operator
   Vector Access Operator
   Operator Precedence
   Standard Functions
   Declaration
   Function Calls
   Return Statements
   Gfunctions
   Declaration
   Block
   Gfunction Calls
   Restrictions
   Program Structure
   If Else Statements
   While Statements
   For Statements
   Scope
4. Project Plan
   Process
   Planning
   Testing
   Style Guide
   Timeline
5. Architectural Design
   Block Diagram
   Preprocessor
   Scanner
   Parser and AST
   Symbol Table
   Semantic Checking and Code Generation

6. Test Plan
   Source to Target
   Test Suites
   Test Automation

7. Lessons Learned
   Gabriel Blanco
   Amelia Brunner
   Ruchir Khaitan
   Ben Barg

8. Appendix
   Preprocessor
   Scanner
   Parser
   AST
   Generator
   Generator Utilities
   Environment Types
   Testing Script
1. Introduction

The Graphics Processing Unit (GPU) was invented in 1999 as a single-chip processor that allowed the CPU to offload graphics-intensive tasks to a separate processor. Unlike the CPU, which is built to contain only a handful of cores but a lot of cache memory, a GPU has limited memory but hundreds of cores, which, thanks to very efficient context switching, allows it to handle thousands of threads concurrently without significantly degrading the performance of the CPU. In the past, GPUs were seen as a luxury reserved for video processing and computer gaming. However, because of the advent of larger screen displays and a greater demand for video, image and signal processing applications, GPUs are quickly becoming more mainstream.

Although we are seeing more and more applications take advantage of the computational capabilities of the GPU, it is still very difficult to program for the GPU because of the many different types of GPU architectures and chip specific proprietary software. Sheets empowers application developers to take a high-level approach to programming on the GPU. With syntax, semantics and program structure to help programmers run parallelizable algorithms on the GPU, Sheets is a hardware-portable language that allows users to leverage the GPU's ability to handle large-vector data operations without needing to worry about hardware specifications. It does so by compiling down into OpenCL, an open-source programming language that can run on a wide variety of GPUs.

2. Language Tutorial

Sheets uses a very clean syntax that is syntactically similar to C, but with the use of white space delimiters and lack of semi-colons of a Pythonic program. The one requirement is every executable program must include a snuggle() function. The snuggle() is the first function that gets called by the CPU, and which determines the rest of the program execution. Confused about the name? Just remember: “What’s the first thing you do when you get into Sheets? You snuggle!”

2.1. Simple Example

Here is a very simple first program that simply adds to numbers, without any gfunc capabilities:

```plaintext
func int snuggle():
    int x = 1
    int y = 2
    int result = x * y
    return result
```

Now, let’s say that you want to perform this exact same operation, but over an array of 6 integers, and want to use GPU acceleration to speed up the calculation. This is how we would do it:

```plaintext
gfunc int[] gmultiply(int[] x, int[] y):
    for (int i=Block.start; i<Block.end; i=i+1;)
        Block.out[i] = x[i] * y[i]

func int[] snuggle():
    int[] x = [1.,2.,3.,4.,5.,6.]
```
```csharp
int[] y = [.5,.5,.5,.5,.5,.5]
int[] result[6]
return result = gmultiply(x,y)
```

_gfunctions_ (Functions that are executed on the GPU) have a unique syntax which abstracts away most of the complexities of performing operations on the GPU, while still enforcing conventions that make it possible for the functions to be called without violating the constraints of using a separate processing unit.

3. **Language Manual**

3.1. **Lexical Conventions**

3.1.1. **Identifiers**

An identifiers is a token given to a variable or function. They must begin with alphabetic character or underscore, followed by any number of alphanumeric characters or underscores. Identifiers are unique and cannot be double declared within the same scope, however you can mask a variable name within a higher level scope. Case matters, upper and lowercase letters are treated as distinct.

3.1.2. **Comments**

_Sheets_ supports both inline and block comments, although there is no support for nested block comments. The comments in _Sheets_ use the octothorpe, since they look a little bit like a threaded sheets:

```markdown
## Inline Comment

#~ A Block Comment ~#

#~
~ A Multiline Block Comment
~#
```

3.1.3. **Whitespace**

Any whitespace found after a new line is syntactic, and used to determine scope. Whitespace is any space character, but _Sheets_ does not allow the use of tab characters, since the size of a tab can vary in depending on the environment. Whitespace after the last non-whitespace, non-comment token is not syntactic and gets stripped away during compilation.

_Sheets_ allows for statements that are more than one line long to the use of the line-join character \ as shown below. Note that the whitespace indentation on the following line is not syntactically important, since it is simply a continuation of the first line.
3.2. Types

Sheets supports two kinds of primitives, both of which are numerical: Fixed Point and Floating Point.

3.2.1. Fixed Point Type

The fixed-point numerical primitive is the int. An int is a simply signed (two's complement) 32-bit integer type. Fixed-point literals can be declared as a sequence of numeric characters, although any number that is larger than the maximum integer size (\(\pm2147483647\)) will cause a compilation error.

3.2.2. Floating Point Type

The floating-point numerical primitive is the float. A float is a simply signed (two's complement) 32-bit single precision number, as defined by IEEE 754. Floating-point literals can be declared as a sequence of numeric characters followed by a decimal, and then an optional fractional component. If the decimal is omitted, the compiler will assume the literal to be a fixed-point type, which may lead to compilation errors.

3.2.3. Vector Types

Single Dimensional Arrays, or Vectors, represent multiple instances of either floating or fixed point numbers allocated in contiguous ranges of memory: either on the stack, the heap or in GPU memory (either globally or locally). Vectors have identical representations on both the CPU and the GPU, and Sheets automates data transfer between the CPU and GPU automatically. Vectors are zero indexed and can be accessed using square-bracket notation.

The syntax for declaring a new array is as follows; int[] is the type, SIZE is the integer number of elements allocated:

```
<vector_type> sampleVector [SIZE]
```

Vector literals can be defined within brackets, given that each element within is a literal of the singleton type of the vector. Declaring a vector literal with mixed types or larger than the allocated size will result in a compilation error

```
int[] sampleVector[4] = [1,2,3,4]
```

Two types of vectors are supported, fixed point int[] and floating point float[].

3.3. Operators

These are all the operators supported in Sheets:
3.3.1. Arithmetic Operators

*Sheets* provides the four basic arithmetic operators: *addition, subtraction, multiplication* and *division*. All arithmetic operators are left-associative binary operators, and multiplication and division have higher operator precedence than addition and subtraction.

3.3.2. Relational Operators

The relational operators compare the values of two expressions, the operators are *equivalence*, *non-equivalence, less-than, greater-than, less-than-or-equal-to* and *greater-than-or-equal-to*. They are used within conditional statements, such as *if, else, while, or for* blocks to control execution of code by testing a boolean condition.

3.3.3. Assignment Operator

The *assignment* operator = is a right-associative binary operator that takes the value on the right hand side of the operator and stores it in the left hand side. Only certain types of expressions, known as *assignment-expressions*, are allowed to be on the left hand side, the “receiving” side of the operator. *Assignment-expressions* are limited to *variables* and *array elements*.

3.3.4. Vector Access Operator

The double brackets [ ] are used to denote a right associative access of the array label immediately to the right, where the expression within the brackets has to be of type int. The value returned by the array access operator is the kth element in the array where k is the integer that the expression within the brackets evaluates to.

```plaintext
int[] sampleVector[4] = [1,2,3,4]
sampleVector[0] = 2
```

3.3.5. Operator Precedence

From highest to lowest precedence, where highest order operators bind tighter than lower order ones:

- Multiplication and Division
- Addition and Subtraction
- Relational Operators
3.4. Standard Functions

On the global scope, you are only allowed to declare variables, so any operations on a variable needs to be done from within a function. Program execution is determined by the `snuggle()` function, the first function called during execution that does not take any arguments. All subsequent functions, both standard and gfunctional can then be called either from `snuggle()`, or from another declared standard function.

3.4.1. Declaration

Standard functions have this declaration syntax:

```
func <type> <name> (arg1, arg2, ...):
...
```

Where `<name>` is a unique identifier, and `<type>` is the type for the returning value. The arguments are optional variable declarations for arguments that get passed in during the invocation of a function. When a function is called, the `function call statement` must be have the same number, ordering and type as the variables declared in the declaration.

3.4.2. Function Calls

Functions are called by using the unique identifier with parentheses to pass in arguments. The syntax for a `function-call` expression is:

```
<name>(arg1, arg2, ...)
```

Where the number of arguments (or lack thereof) must match what has previously been declared.

3.4.3. Return Statements

If a function contains a return statements, it can return a value of the type specified in the function declaration. If the value of a return statement does not match that of the function, the compiler will throw an error.

3.5. Gfunctions

`Gfunctions` are a special subset of functions that get executed on the GPU. `Gfunctions` have a special syntax in `Sheets` that allows programmers to write functions that work with the constraints of the GPU. When a `gfunction` is called, the body of a `gfunction` as well as a copy of the input parameters is sent to a group of threads on the GPU to be processed concurrently. This threaded processing allows for faster vector operations.

3.5.1. Declaration
Gfunctions are declared in much the same way as a standard function, although with the option to define a **Block Size**: the range of indices within a vector that the each instance of a *gfunction* has access to during parallel execution. The syntax for declaring a *gfunction* is:

```
gfunc <vector_type> <name>(arg1, arg2, ...).[BLOCK_SIZE]:
...```

The *gfunction* type must be a vector, since Gfunctions are intended to be used for large vector operations. The block size declarator at the end of the *gfunction* declaration is optional. If it is omitted, the BLOCK_SIZE will be set to 1.

### 3.5.2. Block

In the body of the *gfunction*, you have access to a *gfunction-specific* language construct called **Block**. Block stores several variables that can be accessed by using the dot operator. These are the variables:

**Block.size**

The number of elements that an instance of a *gfunction* can access. This number is set by the programmer in the *gfunction declaration*. The purpose of defining a Block.size is that in order for a function to be parallelizable, the operations on a vector need to be able to be partitioned and performed in parallel. The Block.size defines the minimum number of vector elements in each group of that gets sent to a single thread on the GPU.

**Block.start**

The starting index for an instance of the a *gfunction*. Every instance of a *gfunction* handles a different, non-overlapping section of the array so that no write conflicts occur.

**Block.end**

The ending index for an instance of the a *gfunction*. The difference between the starting and ending index is always the block.size, so that no two instances of a *gfunction* handle overlapping sections of an array.

### 3.5.3. Gfunction Calls

```
<name> (arg1, arg2, ...)
```

*Gfunctions* are applied processed on the GPU, and program execution is sequential, so a *gfunction* call handles all of the GPU memory and thread management internally, and blocks until a value of vector type is returned from the *gfunction*

### 3.5.4. Restrictions
Since a GPU provides a tremendous boost in computational throughput by adhering to a Single Program Multiple Thread (SPMT) model wherein a single sequence of instructions are loaded onto many different cores and executed concurrently, any situation where a core has to halt execution to load other instructions, (for example, in a program with a lot of nested branching) leads to a significant increase in execution time. Therefore, GPU execution is primarily suited for a simple combinations of operations applied independently and concurrently to multiple disjoint parts of a data set, and to further reinforce best practices and maintain high performance, \textit{gfunctions} can only access primitive operations and control flow structures and can not call other functions from within the body of a \textit{gfunction}.

Also, \textit{gfunctions} cannot access variables declared in the program’s global scope, as \textit{gfunctions} operate within a separate GPU memory space that does not replicate variable’s declared globally within the CPU memory. However, global variables can be passed as arguments to \textit{gfunctions}.

### 3.6. Program Structure

The statements outside of any function definitions are in global scope. The only statements allowed in global scope are variable declarations for variables accessible by name from all functions, \textit{gfunction} definitions and function definitions. The starting execution point for a Sheets program is the required \texttt{snuggle()} function that is required to be the last function definition in a Sheets file.

#### 3.6.1. If Else Statements

Sheets supports \texttt{if / else} statements that allow for conditional branching of execution based on some boolean expression. An \texttt{if / else} statement introduces a new scope, and so the body of an \texttt{if} statement which executes if the boolean expression evaluates to true, has to be consistently indented. Conditional statements can nest freely, and finally, each \texttt{if} statement can optionally be paired with an \texttt{else} statement, which has to be at the same scope as the if statement and come immediately after the body of the matching \texttt{if} statement.

```latex
<func OR gfunc> <vector_type> <name>(arg1, arg2, ...):
  ...
  if(<boolean_expression>):
    ...
  else:
    ...
```

#### 3.6.2. While Statements

```latex
<func OR gfunc> <vector_type> <name>(arg1, arg2, ...):
  ...
  while (boolean_expression):
    ...
```

Sheets supports \texttt{while} statements that repeatedly execute the code in the scoped block directly under them as long as the boolean expression within the while statement evaluates to \texttt{true}. 

10
3.6.3. For Statements

Sheets supports for loop statement for repeated code execution as well, where the first assignment statement passed to the for statement is executed only once, and then the scoped block under the body of the for loop is executed as long as the boolean expression in the middle of the for loop evaluates to true. Each time, before the scoped block of statements executes, first, the boolean expression is reevaluated, and after each execution, the third assignment in the for statement is executed.

3.6.4. Scope

Variables declared in a function or gfunction are defined as being in function scope, and are visible or available to be referenced by name within the body of the function. Similarly, the body of a for, while, or if statement introduces a new subscope, where variables declared within that scope are only accessible to be referenced by name within that scope level.

Note that Sheets allows references to variables in a higher scope level (such as global scope) from within a lower sub-scope level and explicitly not the other way around. Also, variables declared in a lower scope level can mask variables declared in a higher scope level.

4. Project Plan

4.1. Process

4.1.1. Planning

Our team met to discuss the project twice a week, on Wednesdays with our T.A. Kuangya Zhai, and then again on Thursday as a team. With our weekly meetings with Kuangya, we would gage our progress and ask questions about any roadblocks we encountered during the week, and then in our weekly team meeting we would assign action items to try to have done by our next meeting. Of course, we did not always meet the goals set every single week, but by setting weekly goals we made sure that the project was always progressing.

There were often moments of “bottlenecking”, times where one member could not continue working on their part of the project until another member had finished their task. We found that the best way to avoid bottlenecking was to maintain good communication, so that if one member’s task was more difficult than previously thought, other members could offer support and help fix the issue.

4.1.2. Testing
We were only able to do full stack testing during the last week, since we only started generating code in the in the last two weeks of the project. For that reason, we developed test suites that tested individual components of the compiler. This meant that when writing a certain part of the compiler, one of the people working on that component was also expected to be writing tests that were expected to either pass or fail the specific module.

Even if the tests failed downstream in an unimplemented part of the compiler, analyzing the error message could informer the programmer whether the test was failing in the current module, upstream or downstream. We will cover the specifics of testing in more depth in Section 6.

4.2. Style Guide

The group used these conventions when writing code to ensure consistency and better readability:

- Lines of code should not be longer than 80 characters
- For consistency over different machines, use only spaces, no tabs, for indentation
- Each program uses either 2-space or 4-space indentation, do not mix within a program.
- Use under_scores for naming variables (as opposed to camelCase).

4.3. Timeline

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 23rd</td>
<td>First commit, creation of project repository</td>
</tr>
<tr>
<td>September 24th</td>
<td>Submitted Project Proposal</td>
</tr>
<tr>
<td>October 27th</td>
<td>Submitted Language Reference Manual</td>
</tr>
<tr>
<td>November 21st</td>
<td>Finished Preprocessor and Scanner</td>
</tr>
<tr>
<td>December 1st</td>
<td>Finished Parser and AST</td>
</tr>
<tr>
<td>December 8th</td>
<td>Successfully Generated Code</td>
</tr>
<tr>
<td>December 17th</td>
<td>Project presentation and submission of Final Project Report</td>
</tr>
</tbody>
</table>

4.4. Roles and Responsibilities

As the project progressed, the roles of each team members became a lot more fluid than the initially assigned roles of Tester, System Architect, Project Manager and Language Guru. Each member was heavily involved in the development of a least two of the major components of the Compiler, and worked to develop tests for of the system they were building and communicating with other members to make sure their component interfaced correctly with the other systems. The team frequently worked together in the same room, which means that a lot of the roles overlapped as we coded in together in pairs and exchanged roles.
### 4.5. Software Development Environment

Because we wanted to execute our language on a physical GPU, we targeted our build environment towards an Amazon Web Services EC2 instance with a GPU driver.

Our Stack:

- **Github-Hosted Git Repository** - for version control, which contains our compiler code, tests and also a small C library to encapsulate some of the OpenCL boilerplate
- **Python 2.7** - for a preprocessor that parses Sheets' whitespace scoping into a whitespace-independent intermediate representation that can be tokenized by an ocamllex-generated lexer.
- **OCaml 4.2.01** - for parsing and semantic checking of our preprocessed Sheets code and generation of C and accompanying OpenCL kernels.
- **GCC** - for building the C output of our compiler.
- **OpenCL 1.1** - headers and libraries for both the Nvidia GPU and Intel CPU, including the just-in-time compiler used by the C output of our compiler.

Our testing environment:

- **Amazon Web Services EC2 g2.2xLarge** Instance with the above packages installed
- **Nvidia Grid K520 GPU Driver Accessible by the AWS instance**
- **Intel Xeon E5-2670 CPU Accessible by the AWS instance**

### 4.6. Project Log

This project log shows a history of 463 commits starting from September 23rd and ending December 17th. As you can see, each member was heavily involved in the development of the project over a 3 month period.

```
commit 92345e70bb0b057a760b28e7f2a8a776ae185e2e87f6c
Author: Gabriel Blanco <blancgab@gmail.com>
Date: Wed Dec 19 11:36:38 2014 -0500
commit 693f4fd9ded8ae4d88e7ee3409135e865ee8934a
Author: arb2196 <arb2196@columbia.edu>
Date: Wed Dec 11 11:54:18 2014 -0500
commit f57b42a567082b9f718a64665c818d8753d0fa4
Author: Benjamin Barg <benbakerbarg@gmail.com>
```
5. Architectural Design

5.1. Block Diagram
5.2. Preprocessor

Filename: preprocessor.py

A preprocessor written in Python that takes the whitespace delimited *.sht source code and produces an output that can be tokenized by the scanner. The preprocessor goes line by line and calculates where new blocks of code are created based on the indentation level, as well as replacing newlines with semi-colons to mark the end of statements (unless a linejoin \ character is used). Inline comments and block comments are also removed in the preprocessor; this is required to correctly calculate correct indentation levels.

In addition to demarking scope from whitespace, the preprocessor also checks for certain invalid characters (tabs, right-brackets and left-brackets) and throws an error if they are encountered, since they could break the scanner.

The preprocessor is called by the execution script, and produces an intermediary output file called with the extension *.proc.sht

Source Code: example.sht

```python
#~ Code before being preprocessed ~#
int result

func int snuggle():
    int[] output[10]
    for (int i=0; i<10 ; i=i+1):
        if (x == 9):
            output[i] = 1  # only last elem=1
        else:
            result[i] = 0
    return result
```

Preprocessed Code: example.proc.sht

```python
int result;
func int snuggle():{
    int[] output[10];
    for (int i=0; i<10 ; i=i+1){
        if (x == 9){
            output[i] = 1;
        } else:
            result[i] = 0;
    }
    return result;
}
```

5.3. Scanner

Relevant Files: scanner.mll
The scanner, written in OCamlLex, takes the intermediary output from the preprocessor and tokenizes it into *keywords, identifiers* and *literals*. It also removes the rest of the block comments that were not removed in the preprocessor as well as *whitespace* (which is no longer syntactically useful). If there exist any other characters which cannot be lexed by the scanner, or if an *identifier* or *literal* is not syntactically valid, the scanner will throw an error.

The tokens created by the scanner are then used by the Parser to create an *Abstract Syntax Tree*.

### 5.4. Parser and AST

**Relevant Files:** parser.mly, ast.ml

The parser, written in OCamlYacc, takes a series of tokens and then, using the grammar declared in parser.mly and the datatypes defined in ast.ml, generates an *Abstract Syntax Tree*. In parser.mly we define the grammar using productions and rules. If the code can be successfully parsed, that means it is syntactically (although not necessarily semantically) correct.

### 5.5. Symbol Table

**Relevant Files:** environment.ml

From the *Abstract Syntax Tree* that is generated by the Parser, we can build a symbol table from all of the identifiers. For each identifier, the symbol table stores information about the type, scope and location of each identifier. By establishing a Symbol Table, the compiler makes certain that an identifier for either a variable or a function does not get rewritten within the same level of scope, and allows us to do type checking of identifier during the Semantic Checking step.

### 5.6. Semantic Checking and Code Generation

**Relevant Files:** generator.ml, generator_utility.ml

*Sheets* compilation is single-pass depth-first traversal of the *Abstract Syntax Tree*, which means that *semantic checking and code generation* happen simultaneously. If an expression or statement passes the semantic check, the translated code in OpenCL is appended to the target file, otherwise the compiler halts and throws a compilation error.

In *semantic checking*, the compiler verifies the validity *Abstract Syntax Tree*. For every binary operator, the compiler checks that the two elements of the operation have the same type since *Sheets* does not support type casting. The *Symbol Table* is used to verify that any identifier which gets called has a) previously been declared, b) is in scope and c) is of the correct type.

If a node within the AST passes semantic check, then the compiler generate the equivalent code in *OpenCL*. For statements in standard functions, translation is a fairly straightforward process; however when translating *gfuntions*, the generator needs to interpret them into *OpenCL* kernel files, which is significantly more complex.
6. Test Plan

6.1. Source to Target

Source Program:

gfunc float[] filter(float[] a).[1]:
    //= block size is 1, so Block.start is just our index //=
    int i = Block.start
    if (a[i] < 5.0):
        Block.out[i] = 0.0
    else:
        Block.out[i] = a[i]

func int snuggle():
    float[] a[5] = [10.0, 20.1, 5.2, 2.3, 90.4]
    float[] result[5] = filter(a)

Target Result:

#include <stdio.h>
#include "aws-g2.2xlarge.h"
#include "cl-helper.h"
#include "timing.h"
#include <CL/cl.h>

#define time_start() get_timestamp(&start)
#define time_end() get_timestamp(&end)
timestamp_type start;
timestamp_type end;
c1_context __sheets_context;
c1_command_queue __sheets_queue;
c1_int c1_err;
const char *filter_kernel_string = 
"__kernel void filter(__global const int __arr_len, __global double *__out, __global
cost double * a){const int _id = get_global_id(0);const int __block_start = _id * 
1;cost int __block_end = _id * 1 + 1;int i = __block_startif( < 5.)__out[i] = 0.;
} else __out[i] = a[i];}";
const char *filter_kernel_name = "filter";
c1_kernel filter_compiled_kernel;
double * filter(int __arr_len, double * a)
{
    c1_mem __arg1 = clCreateBuffer(__sheets_context,
CL_MEM_WRITE_ONLY,
sizeof(double) * __arr_len,
NULL,
& __cl_err);
CHECK_CL_ERROR(__cl_err, "clCreateBuffer");
    c1_mem __arg2 = clCreateBuffer(__sheets_context,
CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
sizeof(double) * __arr_len,
(void *) a,
& __cl_err);
CHECK_CL_ERROR(__cl_err, "clCreateBuffer");
CALL_CL_GUARDED(clEnqueueWriteBuffer,
(__sheets_queue,
6.2. Test Suites

As we developed the compiler, we wrote tests every time that we implemented a new feature to verify that it works. For each new feature, we wrote a minimum of two tests: One intended to pass, and one intended to fail. We built a lot of small tests to test individual elements, such as simple operators, function calls, for loops, variable declarations, etc.
Then, we also had “black box” tests, large programs with a lot of possible point of failures. The black box tests were the last ones that we managed to get to successfully pass.

Running all tests in current directory

Note: an 'x' next to a test indicates that the test is failing when it is expected to pass, or passing when expected to fail. This does not guarantee that a successful test is passing or failing for the expected reason, so make sure to verify that all outputs are correct.

[ ] Running Test: 001_p_funccall
[ ] Running Test: 002_n_funccall
[ ] Running Test: 003_p_funccall
[ ] Running Test: 004_n_funccall
[ ] Running Test: 005_p_globalsum
[ ] Running Test: 006_p_globalsum
[ ] Running Test: 007_p_globalsum
[ ] Running Test: 008_p_globalsum
[ ] Running Test: 009_p_globalproduct
[ ] Running Test: 010_p_globalproduct
[ ] Running Test: 011_p_globalproduct
[ ] Running Test: 012_p_assignment
[ ] Running Test: 013_n_assignment
[ ] Running Test: 014_p_initialization
[ ] Running Test: 015_p_initialization
[ ] Running Test: 016_p_return
[ ] Running Test: 017_p_return
[ ] Running Test: 018_n_return
[ ] Running Test: 019_n_return
[ ] Running Test: 020_n_return
[ ] Running Test: 021_p_return
[ ] Running Test: 022_p_ifelse
[ ] Running Test: 023_n_ifelse
[ ] Running Test: 024_n_ifelse
[ ] Running Test: 025_p_ifelse
[ ] Running Test: 026_n_ifelse
[ ] Running Test: 027_p_for
[ ] Running Test: 028_p_for
[ ] Running Test: 029_n_for
[ ] Running Test: 030_n_for
[ ] Running Test: 031_n_for
[ ] Running Test: 032_p_for
[ ] Running Test: 033_p_while
[ ] Running Test: 034_n_while
[ ] Running Test: 035_n_while
[ ] Running Test: 036_p_vdecl
[ ] Running Test: 037_n_vdecl
[ ] Running Test: 038_n_fdecl
[ ] Running Test: 039_p_fdecl
[ ] Running Test: 040_p_literals
6.3. Test Automation

With upwards up sixty unique tests in our suite by the end of the project, automation became a necessity very quickly. We wrote a test script, run_all.sh, that not only runs every test, but also checks the test name for the string “_n_”, which indicates a test that is intended to fail in the compiler. By redirecting the stderr, the testing script checks to see if a test fails and then prints a result (see above) of whether the test behaves as expected.

If a test fails when it was expected to pass, or vice versa, then the script marks that test with an “X” in the output. The script also writes the output, both stdin and stderr, to an output file which the programmer can then read to determine the exact cause of the error.

The testing script is included in the Appendix.

7. Lessons Learned

7.1. Gabriel Blanco

I learned a valuable skill in organizing a group project: coordinating sleep schedules with your group members for maximum efficiency. You need to fine tune it so that when one member of your team is starts to get sleepy, then another one can step in and pick up where the other left off, maximizing Peak Coding Efficiency Hours (PCEH). If one member was awake most of the previous night, they’ll need to get more sleep the following day. Caffeine adds a whole ‘nother wrench into the equation, because that’s just one more variable to account for. Further testing is required.
7.2. Amelia Brunner

I learned that setting up concrete timelines for large scale projects takes more effort than just setting deadlines onto the calendar; it requires actually UNDERSTANDING the gravity of tasks involved and planning for large tasks instead of letting them ambush you at the last minute. Also, functional programming is cool.

7.3. Ruchir Khaitan

Having a consistent coding style and naming convention is very important. Block comments above function definitions are critical. Wear sunscreen. Variable and function names should tell a story. Currying is good; curry is better.

7.4. Ben Barg

Don’t push code until it you’ve run it, and made sure it builds and passes all the tests! When you are working on a team project, you have to be constantly conscious of breaking other people’s code. Also, make sure to implement the simplest possible functionality of your project first, and only then start adding features; this is the best way to avoid overextending yourself.

Lastly, I learned that functional programming is indeed quite useful.

8. Appendix

8.1. Preprocessor

preprocessor.py

```python
#!/usr/bin/python
#
import os
import re
import sys

# Find the best implementation available on this platform
try:
    from cStringIO import StringIO
except:
    from StringIO import StringIO

def process(input_file):
    invalidchar = (',', ')', '
')
    blockcomment = ['#~', '~#']
    stack = []
```
output = StringIO()
newindent = False
commented = False
linejoin = False

for i, line in enumerate(input_file):
    lineout = remove_inline(line)

    if lineout:
        for x in invalidchar:
            if x in lineout:
                error("SyntaxError: Invalid character {} found on line {}".format(x,i))

        # Check if first statement is a block comment
        lstripline = lineout.lstrip()
        if len(lstripline) > 1 and blockcomment[0] == lstripline[0:2]:
            commented = True

        # Checks if line gets uncommented
        if commented:
            if len(lineout) > 1 and blockcomment[1] == lineout[-2:]:
                commented = False
        else:
            if not linejoin:
                wcount = len(lineout) - len(lineout.lstrip(' '))

                # If the previous line began an indentation, add the new
                # indentation level to the block (so long as the new indentation
                # level is greater than the previous one)
                if newindent:
                    if wcount > stack[-1]:
                        stack.append(wcount)
                        newindent = False
                    else:
                        error("IndentationError on line {}".format(i))

                # If the indentation level is greater than expected, throw an error
                if wcount > stack[-1]:
                    error("IndentationError on line {}".format(i))

            else:
                # If the indentation level is less than the current level, return
                # to a previous indentation block. Throw an error if you return to
                # an indentation level that doesn't exist
                while(wcount < stack[-1]):
                    lineout = "") + lineout
                    stack.pop()

                if wcount != stack[-1]:
                    error("IndentationError on line {}".format(i))

                # Given that the indentation level is correct, check for the start
                # of a new code block (where a line ends with a ':' and insert a
                # '{'). At the end of a line, add a semicolon ';' unless if there is
                # a linejoin character '\'.
                if lineout[-1] == ':':
                    lineout = lineout + '{
                    newindent = True

                elif lineout[-1] == '\\':
                    linejoin = True
                    lineout = lineout[:-1]

            else:
lineout = lineout + ';
' linejoin = False
output.write(lineout)

while 0 < stack[-1]:
    output.write('}"
    stack.pop()

return output
def error(msg):
sys.stderr.write(msg+'\n')
sys.exit(2)
def remove_inline(line):
    if "##" in line:
        regex = re.compile("^\(.\?\)#\.*\|\.*")
        m = regex.match(line)
        comments_removed = m.group(1)
    else:
        comments_removed = line
    return comments_removed.rstrip()
def usage():
    print"
python preprocessor.py [input.sht]
"

if __name__ == '__main__':
    if len(sys.argv) != 2:
        usage()
        sys.exit(2)
try:
    f_in = open(sys.argv[1],"r")
except IOError:
    error("IOError: Cannot read input file %s\n" % sys.argv[1])
name_ext = os.path.basename(f_in.name)
dir_ext = os.path.dirname(f_in.name)+"/

if name_ext.lower().endswith(('.sht','.sheet')):
    fname = os.path.splitext(name_ext)[0]
else:
    error('NameError: Input must have Sheets file extension')
out_str = process(f_in)
f_out = open(dir_ext+fname+'.proc.sht', 'w')
f_out.write(out_str.getvalue())

8.2.  Scanner

scanner.mll

(*
 * Sheets scanner
 *)
/* Authors: Gabriel Blanco, Ruchir Khaitan  
* Copyright 2014, Symposium Software  
* */

{ open Parser;; }

let num = ['0'..'9']
let flt = '-'?num+ '.' num* | '.' num

rule token = parse
(* Whitespace *)
| [' ' '
' ''] { token lexbuf }

(* Comments *)
| "#~" { comment lexbuf }

(* Punctuation *)
| '(' { LPAREN } | ')') { RPAREN }
| '{' { LBRACE } | ')') { RBRACE }
| '[' { LBRACK } | ']' { RBRACK }
| ';' { SEMI } | ';' { COMMA }
| '.' { PERIOD } | ':' { COLON }

(* Arithmetic Operators *)
| '+' { PLUS } | '-' { MINUS }
| '*' { TIMES } | '/' { DIVIDE }

(* Relational Operators *)
| '=' { EQ } | '!=' { NEQ }
| '<' { LT } | '<=' { LEQ }
| '>' { GT } | '>=' { GEQ }

(* Assignment Operator *)
| '=' { ASSIGN }

(* Conditional Keywords *)
| "if" { IF } | "else" { ELSE }

(* Loop Keywords*)
| "while" { WHILE } | "for" { FOR }
| "break" { BREAK } | "continue" { CONTINUE }

(* Function Keywords *)
| "func" { FUNC } | "gfunc" { GFUNC }
| "return" { RETURN }

(* Type Keywords*)
| "int" { INT } | "float" { FLOAT }
| "Block" { BLOCK }

(* End-of-File *)
| eof { EOF }

(* Identifiers *)
| ['a'..'z' 'A'..'Z' '_'] ['a'..'z' 'A'..'Z' '0'..'9' '_']* as lxm { ID(lxm) }

(* Literals *)
| '-'?num+ as intlit { INT_LITERAL(int_of_string intlit) }
| flt as fltit { FLOAT_LITERAL(float_of_string fltit) }

(* Throw Error for Invalid Token *)
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
| "#~" { token lexbuf } (* End-of-comment *)
| _ { comment lexbuf } (* Eat everything else *)
8.3. Parser

parser.mly

/*
 * Sheets parser
 * Authors: Amella Brunner, Gabriel Blanco, Ben Barg
 * Copyright 2014, Symposium Software
 */
%
{ open Ast; }
%

///////////////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////////////

/* Punctuation Tokens */
%token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK SEMI COMMA PERIOD COLON EOF

/* Loop Keywords */
%token WHILE FOR BREAK CONTINUE

/* Conditional Keywords */
%token IF ELSE

/* Function Keywords */
%token FUNC GFUNC BLOCK RETURN

/* Type Keywords */
%token INT FLOAT

/* Operator Tokens */
%token PLUS MINUS TIMES DIVIDE
%token EQ NEQ LT LEQ GT GEQ

/* Assignment Operator */
%token ASSIGN

%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <int list> INT_ARRAY_LITERAL
%token <float list> FLOAT_ARRAY_LITERAL
%token <string> ID

/* Precedence Definition */
%nonassoc NOELSE ELSE
%right ASSIGN G_ASSIGN
%left EQ NEQ
%left LT LEQ GT GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left PERIOD

%start program  /* start symbol */
%type <Ast.program> program /* type returned by program */
%

///////////////////////////////////////////////////////////////////////////////
program:  /* [vdecls], [fdecls] */
  | /* Empty Program */
  | { [], [] }  
  | program vdecl SEMI  { ($2 :: fst $1), snd $1 }  
  | program fdecl  { fst $1, ( $2 :: snd $1 ) }  
  | program gfdecl  { fst $1, ( $2 :: snd $1 ) }  

FUNCTION type_name ID LPAREN forms_opt RPAREN COLON  
LBRACE stmt_list_opt RBRACE  
{}
  r_type = $2; (* return type *)  
  fname = $3; (* function name *)  
  formals = $5; (* list of arguments *)  
  body = $9; (* statement list *)  
  isGfunc = false; (* false b/c not a gfunc *)  
  blocksize = -1 (* block size unused *)  
}

FUNCTION type_name ID LPAREN forms_opt RPAREN blocksize COLON  
LBRACE gffunc_stmt_list_opt RBRACE  
{}
  r_type = $2; (* return type *)  
  fname = $3; (* gfunction name *)  
  formals = $5; (* list of arguments *)  
  body = $10; (* gfunction statement list *)  
  isGfunc = true; (* true b/c a gfunc *)  
  blocksize = $7 (* block size *)  
}

Optional Formal Arguments */  
formals_opt:  
  | /* Nothing */
  | { [] }  
  | formal_list  

formal_list:  
  | vdecl  
  | formal_list COMMA vdecl  

blocksize:  
  | /* Nothing */
  | PERIOD LBRACK INT_LITERAL RBRACK  

VARIABLES */  

Type Declaration */  
type_name:  
  | INT LBRACK RBRACK { "int[]" }  
  | FLOAT LBRACK RBRACK { "float[]" }  
  | INT { "int" }
FLOAT { "float" }

/* Optional Array Size Declaration */
array_opt:
    | /*Nothing*/ { -1 }
    | LBRAKC INT_LITERAL RBRACK { $2 }

/* <type> name [<array_size>] */
vdecl:
    type_name ID array_opt
    {
        v_type = $1; (* variable type *)
        v_name = $2; (* variable name *)
        a_size = $3; (* array size *)
    }

vdecl_list:
    | vdecl SEMI { [$1] }
    | vdecl_list vdecl SEMI { $2 :: $1 }

////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
_STAETMENTS //////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////

/* Optional List of Statements */
stmt_list_opt:
    | /* Nothing */ { [] }
    | stmt_list { list.rev $1 }

stmt_list:
    | stmt { [$1] }
    | stmt_list stmt { $2 :: $1 }

/* Optional List of Gfunction Statements */
gfunc_stmt_list_opt:
    | /* Nothing */ { [] }
    | gfunc_stmt_list { list.rev $1 }

gfunc_stmt_list:
    | gstmt { [$1] }
    | gfunc_stmt_list gstmt { $2 :: $1 }

/* Optional List of Arguments */
args_opt:
    | /* Nothing */ { [] }
    | args_list { list.rev $1 }

args_list:
    | expr { [$1] }
    | args_list COMMA expr { $3 :: $1 }

/* Statements are found in the body of a functions, Gstatements *
* are found in Gfunctions. Gstatements contain all statements *
* as well as access to the special Block construct. */
stmt:
    | vdecl SEMI { Vdecl($1) }
    | expr SEMI { Expr($1) }
    | RETURN expr SEMI { Return($2) }
    | assign_expr ASSIGN expr SEMI { Assign($1, $3) }
    | vdecl ASSIGN expr SEMI { Init($1, $3) }
    | LBRAKC stmt_list RBRACE { Block(list.rev $2) }
    | IF bool_block COLON block_body %prec NOELSE { If($2, $4, Block[] ) }
    | IF bool_block COLON block_body ELSE COLON block_body { If($2, $4, $7) }
    | FOR for_pt1 for_pt2 for_pt3 COLON block_body { For($2, $3, $4, $6) }
    | WHILE bool_block COLON block_body { While($2, $4) }
stmt:
| vdecl SEMI { Vdecl($1) }
| expr SEMI { Expr($1) }
| blockexpr SEMI { Expr($1) }
| g_assign_expr ASSIGN expr SEMI { Assign($1, $3) }
| g_assign_expr ASSIGN blockexpr SEMI { Assign($1, $3) }
| vdecl ASSIGN expr SEMI { Init($1, $3) }
| vdecl ASSIGN blockexpr SEMI { Init($1, $3) }
| IF bool_block COLON gblock_body %prec NOELSE { If($2, $4, Block([])) }
| IF bool_block COLON gblock_body ELSE COLON gblock_body { If($2, $4, $7) }
| FOR gfor_pt1 gfor_pt2 gfor_pt3 COLON gblock_body { For($2, $3, $4, $6) }
| WHILE bool_block COLON gblock_body { While($2, $4) }

/* Conditional and Loop Statements*/
bool_block: LPAREN bool_expr RPAREN { $2 }

block_body: LBRACE loop_stmt_list RBRACE { Block(List.rev $2) }
block_body: LBRACE loop_stmt_list RBRACE { Block(List.rev $2) }

for_pt1: LPAREN stmt { $2 }
for_pt2: bool_expr SEMI { $1 }
for_pt3: stmt RPAREN { $1 }

gfor_pt1: LPAREN gstmt { $2 }
gfor_pt2: bool_expr SEMI { $1 }
gfor_pt3: gstmt RPAREN { $1 }

/* Loops can contain all normal expressions, and also Break and Continues */
loop_stmt_list:
  | /* Nothing */ []
  | stmt_list { $1 }
  | loop_stmt_list looplevel { $2 :: $1 }

gloop_stmt_list:
  | /* Nothing */ []
  | gfunc_stmt_list { $1 }
  | gloop_stmt_list looplevel { $2 :: $1 }

/////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////////////// EXPRESSIONS /////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////

looplevel:
| CONTINUE SEMI { Continue }
| BREAK SEMI { Break }

blockexpr:
| BLOCK PERIOD ID { BlockAcc($3, Literal_int(-1)) }
| BLOCK PERIOD ID LBRACK INT_LITERAL RBRACK { BlockAcc($3, Literal_int($5)) }
| BLOCK PERIOD ID LBRACK ID RBRACK { BlockAcc($3, Id($5)) }

bool_expr:
| expr EQ expr { Binop($1, Equal, $3) }
| expr NEQ expr { Binop($1, Neq, $3) }
| expr LT expr { Binop($1, Less, $3) }
| expr LEQ expr { Binop($1, Leq, $3) }
| expr GT expr { Binop($1, Greater, $3) }
| expr GEQ expr { Binop($1, Geq, $3) }

array_expr:
| ID { Id($1) }
| array literal { $1 }

assign_expr:
8.4. AST

ast.ml

(*
 * Sheets Abstract Syntax Tree types
 *
 * Authors: Amelia Brunner, Gabriel Blanco
 * Copyright 2014, Symposium Software
 *)

type op = Plus | Minus | Times | Divide | Equal | Neq | Less | Leq | Greater | Geq

type expr =
    | Literal_int of int
    | Literal_int_a of int list
    | Literal_float of float
    | Literal_float_a of float list
    | Id of string
    | Binop of expr * op * expr
    | Call of string * expr list
    | ArrayAcc of expr * expr
    (* ArrayAcc(expr1,expr2)*)
    | expr1 : evaluates to an array,
8.5. Generator

(*
 * Sheets Code Generator
 *
 * Authors: Ruchir Khaitan, Ben Barg, Amelia Brunner
 * Copyright 2014, Symposium Software
 *)

open Ast;;
open Environment;;
open Printf;;
open String;;

exception SyntaxError of int * int * string;;
exception NotImplementedWarning of string;;
exception UndefinedTypeError;;
exception BadExpressionError of string;;
let rec generate_type datatype env =
  match datatype with
  | Int -> Environment.append env [Text("int")]
  | Float -> Environment.append env [Text("double")]
  | Array(t) -> Environment.append env [Generator(generate_type t);
      Text("*")) (* Handling array types differently *)

let generate_checked_id check_id id env =
  if (check_id id env) then
    Environment.append env [Text(id)]
  else raise (VariableNotFound id)

let op_to_txt op =
  match op with
  | Plus -> "+
  | Minus -> "-
  | Times -> "*
  | Divide -> "/"
  | Equal -> "==
  | Greater -> ">"
  | Less -> "<"
  | Geq -> ">=
  | Leq -> "=<
  | Neq -> "!="
  | _ -> ""

let rec exp_to_txt exp =
  match exp with
  | Literal_int(i) -> string_of_int(i)
  | Literal_float(f) -> string_of_float(f)
  | Id(s) -> s
  | Binop(e1, op, e2) -> (exp_to_txt e1) ^ " " ^ (op_to_txt op) ^ " " ^ (exp_to_txt e2)
  | Literal_string(s) -> "\"" ^ s ^ "\"
  | _ -> ""

let rec args_to_txt arg_list str=
  match arg_list with
  | [] ->
    if(String.contains str ',') then
      String.sub str 0 (String.length str - 2)
    else
      str
    | arg :: arg_tail -> args_to_txt arg_tail (str ^ (exp_to_txt arg) ^ ",")

let generate_checked_binop check_binop binop env =
  check_binop binop env;
  match binop with
  | Binop(e1, op, e2) -> Environment.append env [
    Text((exp_to_txt e1) ^ " " ^
      (op_to_txt op) ^ " " ^
      (exp_to_txt e2))]
  | _ -> raise (BadExpressionError("binop"))

let generate_checked_array_access check_array_access array_expr env =
check_array_access array_expr env;
match array_expr with
| ArrayAcc(e1, e2) -> Environment.append env [Text((exp_to_txt e1) ^ "[" ^
(exp_to_txt e2) ^ "]")]
| _ -> raise (BadExpressionError("Array Access"))

let generate_checked_f_call check_f_call f_call env =
check_f_call f_call env;
match f_call with
| Call(id, expressions) ->
  if Environment.id_is_gfunc id env then
    Environment.append env [Text(id ^ "(100000, " ^
    (args_to_txt expressions "")
    ^ ")");]
  else
    Environment.append env [Text(id ^ "(" ^
    (args_to_txt expressions "")
    ^ ")");]
| _ -> raise (BadExpressionError("Function Call"))

let rec print_int_array array_list str =
match array_list with
| [] ->
  if (String.contains str ',' ') then
    String.sub str 0 (String.length str - 2)
  else
    str
| head :: array_tail ->
  print_int_array array_tail (str ^ (string_of_int head) ^ ", ", )

let rec print_float_array array_list str =
match array_list with
| [] ->
  if (String.contains str ',' ') then
    String.sub str 0 (String.length str - 2)
  else
    str
| head :: array_tail ->
  print_float_array array_tail (str ^
    (string_of_float head) ^ ", ", )

let rec generate_exp exp env =
match exp with
| Literal_int(i) -> Environment.append env [Text(string_of_int(i))]
| Literal_float(f) -> Environment.append env [Text(string_of_float(f))]
| Literal_int_a(list_i) -> Environment.append env [Text("{ ");
  Text(print_int_array list_i ")");
Text("}"");]
| Literal_float_a(list_f) -> Environment.append env [Text("{ ");
  Text(print_float_array list_f ")");
Text("}"");]
| Id(s) -> Environment.append env [Generator(generate_checked_id
is_var_in_scope s )]
| Binop(., ., .) -> Environment.append env [Generator(generate_checked_binop
Generator_utilities.expr_typeof exp)]
| Call(func_id, formals_list) -> Environment.append env [Generator(generate_checked_f_call
Generator_utilities.expr_typeof exp)]
| ArrayAcc(., _) -> Environment.append env [Generator(generate_checked_array_access
Generator_utilities.expr_typeof exp)]
| BlockAcc(id, exp) -> Environment.append env [Generator(generate_checked_block id exp)]
| _ -> raise (NotImplementedError("unsupported expression"))
and generate_checked_block id exp env =
match id with
| "start" -> Environment.append env [Text("__block_start")]

52
let generate_init vdecl exp env =
if((GeneratorUtilities.vdecl_type vdecl) = (GeneratorUtilities.expr_typeof exp env)) then
  let v_type = GeneratorUtilities.vdecl_type vdecl in
  match v_type with
    | Array(Int) -> Environment.append env [Env(add_var vdecl.v_name (v_type));
      Text("int" ^ " " ^ vdecl.v_name ^ "]" = "");
        Generator(generate_exp exp);]
    | Array(Float) -> Environment.append env [Env(add_var vdecl.v_name (v_type);
      Text("float" ^ " " ^ vdecl.v_name ^ "]" = "");
        Generator(generate_exp exp);]
    | _ -> Environment.append env [Env(add_var vdecl.v_name
      (GeneratorUtilities.vdecl_type vdecl));
      Text((GeneratorUtilities.c_type_from_str vdecl.v_type)
        ^ " " ^ vdecl.v_name ^ " = ");
        Generator(generate_exp exp)]
  else
    raise(BadExpressionError("Assignment of incompatible types"))

let generate_return exp env =
let func_info = Environment.get_func_info env.current_function env in
let return_type = func_info.return in
let exp_type = GeneratorUtilities.expr_typeof exp env in
if(exp_type = return_type) then
  Environment.append env [
    Text("return ");
    Generator(generate_exp exp)]
else
  raise(BadExpressionError("Bad return type"))

let generate_assign id exp env =
match id with
  | Id(a) -> if (is_var_in_scope a env) then
    if(GeneratorUtilities.expr_typeof id env =
      GeneratorUtilities.expr_typeof exp env) then
      Environment.append env [
        Text(a ^ " = " );
        Generator(generate_exp exp)]
    else
      raise(BadExpressionError("Assignment of incompatible types"))
  else
    raise (BadExpressionError("assignment to undefined id"))
  | BlockAcc(s, expr) -> Environment.append env [Text("_out[");
    Generator(generate_exp expr);
    Text("] = ");
    Generator(generate_exp exp);
    Text(";")]
  | _ -> raise (BadExpressionError("Invalid Assignment"))

let rec process_stmt_list stmt_list env =
match stmt_list with
  stmt :: other_stmts -> Environment.append env [Generator(process_stmt

stmt); Generator(process_stmt_list other_stmts)]
| []  -> Environment.append env [ Text(""") ]
and process_stmt stmt env =
match stmt with
Vdecl(vdecl) ->
    Environment.append env [ Generator(process_vdecl vdecl);
    Text(";\n") ]
| Block(stmt_list) ->
    Environment.append env [ Generator(process_stmt_list stmt_list) ]
| Expr(expr) ->
    Environment.append env [ Generator(generate_exp expr);
    Text(";\n") ]
| Assign(name, expr) ->
    Environment.append env [ Generator(generate_assign name expr);
    Text(";\n") ]
| Return(expr) ->
    Environment.append env [ Generator(generate_return expr);
    Text(";\n") ]
| Init(vdecl, expr) ->
    Environment.append env [ Generator(generate_init vdecl expr);
    Text(";\n") ]
| If(boolexpr, ifstmt, elsestmt) ->
    Environment.append env [ Generator(generate_if boolexpr ifstmt elsestmt)]
| While(expr, body) ->
    Environment.append env [ NewScope(generate_while expr body)]
| For(s1, e2, s3, body) ->
    Environment.append env [ NewScope(generate_for s1 e2 s3 body)]
| Continue ->
    Environment.append env [ Text("continue;\n") ]
| Break ->
    Environment.append env [ Text("break;\n") ]
| _  -> raise (NotImplementedError("Undefined type of expression"))
and process_vdecl vdecl env =
let v_datatype = Generator_utilities.str_to_type vdecl.v_type in
Environment.append env [ Env(add_var vdecl.v_name v_datatype);
    Text((Generator_utilities.c_type_from_str vdecl.v_type)
    ^ " " ^ vdecl.v_name)]
and generate_while bool_expr body env =
match bool_expr with
| Binop(e1, o, e2) ->
    match o with
    | Equal -> append_while bool_expr body env
    | Neq -> append_while bool_expr body env
    | Greater -> append_while bool_expr body env
    | Less -> append_while bool_expr body env
    | Geq -> append_while bool_expr body env
    | Leq -> append_while bool_expr body env
    | _ -> raise (BadExpressionError("Binop is not boolean"))
| _  -> raise(BadExpressionError("Conditional expression is not binop"))
and generate_for stmt1 bool_expr stmt2 body env =
match bool_expr with
| Binop(e1, o, e2) ->
    match o with
    | Equal -> append_for stmt1 bool_expr stmt2 body env
    | Neq -> append_for stmt1 bool_expr stmt2 body env
    | Greater -> append_for stmt1 bool_expr stmt2 body env
    | Less -> append_for stmt1 bool_expr stmt2 body env
    | Geq -> append_for stmt1 bool_expr stmt2 body env
    | Leq -> append_for stmt1 bool_expr stmt2 body env
    | _  -> raise (BadExpressionError("Binop is not boolean"))
| _  -> raise(BadExpressionError("Conditional expression is not binop"))
and generate_if bool_expr ifbody elsebody env =
match bool_expr with
| Binop(e1, o, e2) ->
    match o with
    | Equal -> append_if stmt1 bool_expr stmt2 body env
    | Neq -> append_if stmt1 bool_expr stmt2 body env
    | Greater -> append_if stmt1 bool_expr stmt2 body env
    | Less -> append_if stmt1 bool_expr stmt2 body env
    | Geq -> append_if stmt1 bool_expr stmt2 body env
    | Leq -> append_if stmt1 bool_expr stmt2 body env
    | _  -> raise (BadExpressionError("Binop is not boolean"))
| _  -> raise(BadExpressionError("Conditional expression is not binop"))
| Equal -> append_if_else bool_expr ifbody elsebody env |
| Neq -> append_if_else bool_expr ifbody elsebody env |
| Greater -> append_if_else bool_expr ifbody elsebody env |
| Less -> append_if Else bool_expr ifbody elsebody env |
| Geq -> append_if_else bool_expr ifbody elsebody env |
| Leq -> append_if_else bool_expr ifbody elsebody env |
| __ -> raise (BadExpressionError("Binop is not boolean")) |

and append_while bool_expr body env =
Generator_utilities.expr_typeof bool_expr env;
Environment.append env [Text("While("); Generator(generate_exp bool_expr);
Text(")"); NewScope(process_stmt ifbody); Text("\n")]

and append_if_else bool_expr ifbody elsebody env =
Generator_utilities.expr_typeof bool_expr env;
Environment.append env [Text("if("); Generator(generate_exp bool_expr);
Text(")"); NewScope(process_stmt ifbody); Text("\n")]

(* For loops have to have assignment, boolean expression, assignment *)

and print_in_for_loop stmt first env =
match stmt with
Assign(name, expr) -> if first then Environment.append env [
Generator(generate_assign name expr);
Text(";")]
else Environment.append env [
Generator(generate_assign name expr);
]
| __ -> raise (BadExpressionError("Argument in for loop invalid")) |

and append_for stmt1 bool_expr stmt2 body env =
Generator_utilities.expr_typeof bool_expr env;
Environment.append env [Text("For("); Generator(print_in_for_loop stmt1 true );
Generator(generate_exp bool_expr); Text(";");
Generator(print_in_for_loop stmt2 false); Text("\n"); Generator(process_stmt body);
Text("\n")]

(* Global Variable Declarations *)

(* Global Constant Declarations *)

let rec generate_global_vdecl_list vdecls env =
let generate_global_vdecl vdecl env =
let v_datatype = Generator_utilities.str_to_type vdecl.v_type in
Environment.append env [Env((add_var vdecl.v_name v.datatype));
Generator(generate_type v_datatype);
Text(" ^ vdecl.v_name ^ ";\n")]

match vdecls with
[ ] -> ",", env
| [vdecl] -> generate_global_vdecl vdecl env
| vdecl :: other_vdecls ->
Environment.append env [Generator(generate_global_vdecl vdecl);
Generator(generate_global_vdecl_list other_vdecls)]

let rec generate_formals_vdecl_list vdecl_list env =
let generate_formals_vdecl vdecl env =
let v_datatype = Generator_utilities.str_to_type vdecl.v_type in
Environment.append env [Env((add_var vdecl.v_name v.datatype));
Generator(generate_type v_datatype);
Text(" ^ vdecl.v_name ^ ", ")]

match vdecl_list with
[ ] -> ",", env
| [vdecl] -> Environment.append env [Env((add_var vdecl.v_name
(Generator_utilities.vdecl_type vdecl)));
Text((Generator_utilities.c_type_from_str
vdecl.v_type)
^ " ^ vdecl.v_name)]
vdecl :: other_vdecls ->
   Environment.append env [Generator(generate_formals_vdecl vdecl);
    Generator(generate_formals_vdecl_list other_vdecls)]

(* - --------------------------------- * *)
(* Kernel invocation declarations   * *)
(* - --------------------------------- * *)
(* If we have a gfunc declared as

> gfunc float[] my_gfunc(float[] arg1, float[] arg2, int arg3)

then our generated C code will call this function in the same
way that CPU functions are called:

> result = my_gfunc(arg1, arg2, arg3, arg4)

However, when my_gfunc is actually defined in the C file, its
contents will be the boilerplate OpenCL code that invokes the
kernel on the GPU.

At this point, all semantic checking has been completed, so we
don't need to worry about checking the function map or anything
like that.

NOTES:
- Because each kernel invocation function has its own C scope, we
don't have to worry about variable name collision
- The definitions of these functions will appear interspersed with
CPU func definitions, but this will not interfere with namespace
conventions. Essentially, the CPU code thinks it's calling
another CPU function, but internally that CPU function is
implemented as a GPU function

- This method has the side-benefit that we don't have to process
  literals passed to functions differently

INVARIANTS:
- the output array and input arrays of an individual gfunc MUST be
  the same size
- if there are non-array arguments, we rename them to __argn
- the actual args list starts at arg2 (first 2 are reserved for
  size and output array*)

let generate_kernel_invocation_function fdecl env =
  let base_r_type = Generator_utilities.arr_type_str_to_base_type fdecl.r_type in
  let generate_cl_arg_list fdecl env =
    let rec generate_cl_args arg_n formals env =
      let generate_cl_arg arg_n formal env =
        if Generator_utilities.is_array_type formal.v_type then
          (* create a CL memory buffer for array args *)
          Environment.append env [Text(sprintf "cl_mem __arg%d = clCreateBuffer(" arg_n);
            Text("__sheets_context, \n");
            Text("CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, \n");
            Text(sprintf "sizeof(%s) * __arr_len, \n" base_r_type);
            Text(sprintf "(void *) %s, \n" formal.v_name);
            Text("&_cl_err);\n")
          else
            (* pass primitives directly *)
            Environment.append env [Text(sprintf "%s __arg%d = %s;\n" base_r_type arg_n formal.v_name)]
        in
        match formals with
        [] -> ", env
        | formal :: other_formals ->
          Environment.append env [Generator(generate_cl_arg arg_n formal);
          Generator(generate_cl_args (arg_n + 1) other_formals)]
in
(* we make a cl_mem buffer for the return array first *)
Environment.append env [Text("cl_mem __arg1 = clCreateBuffer();
    Text("__sheets_context,\n");
    Text("CL_MEM_WRITE_ONLY,\n");
    Text(sprintf "%s\n", base_r_type);
    Text("\n\n);
    Text("\n\n"));
    Text("CHECK_CL_ERROR(\n"));
    Text("\n\n")];
(* user-defined args start at 2 *)
Generator(generate_cl_args 2 fdecl.formals)]

in
let generate_cl_enqueue_write_buffer_list fdecl env =
let rec generate_cl_enqueue_writeBuffers arg_n formals env =
    let generate_cl_enqueue_write_buffer arg_n formal env =
        if Generator_utilities.is_array_type formal.v_type then
            Environment.append env [Text("CALL_CL_GUARDED(clEnqueueWriteBuffer,\n");
                Text("\n\n");
                Text(sprintf "\n", arg_n);
                Text("CL_TRUE,\n"); (* ensure blocking write *)
                Text("\n"); (* no offset *)
                Text(sprintf "%s\n", base_r_type);
                Text(sprintf "\n", formal.v_name);
                Text("\n");
                Text("\n"); (* no wait list *)
                Text("\n");
                Text("\n")];
        else "", env (* no need to alloc buffer for primitives *)
        in
        match formals with
        [ ] -> "", env |
        | formal :: other_formals ->
            Environment.append env [Generator(generate_cl_enqueue_write_buffer arg_n
            formal)];
        Generator(generate_cl_enqueue_writeBuffers
            (arg_n + 1) other_formals)]

in
(* user-defined args start at 2 *)
Environment.append env [Generator(generate_cl_enqueue_writeBuffers 2
fdecl.formals)]

in
let generate_cl_set_kernel_args fdecl env =
    (* list of __argn vars *)
let generate_arg ns num_user_args env =
    let rec _helper num_arg ns_left arg_n env =
        match num_arg ns_left with
        0 -> "", env |
        | 1 -> Environment.append env [Text(sprintf "\n", arg_n)]
        | _ -> Environment.append env [Text(sprintf "\n", arg_n);
            Generator(_helper (num_arg ns_left - 1) (arg_n +
                        1))]
        in
        Environment.append env [Generator(_helper num_user_args 2)]

in
(* only need to add 1 because __arr_len is already in formals list *)
Environment.append env [Text(sprintf "%s_KERNEL_ARGS(" ((List.length
fdecl.formals) + 2));
    Text(sprintf "%s\n", fdecl.fname);
    Text("\n");
    Text("\n");
    Generator(generate_arg ns (List.length fdecl.formals));
    Text("\n")]

in
let global_work_items = function
(* provide a buffer for when block_size doesn't divide array
size*)
let generate_cl_enqueue_nd_range_kernel fdecl env =
  Environment.append env [Text(sprintf "size_t gdims[] = { %s };\n"
    (global_work_items fdecl.blocksize));
  Text("CALL_CL_GUARDED(clEnqueueNDRangeKernel,\n")
  Text("%sheets_queue,\n");
  Text(sprintf "%s_compiled_kernel,\n" fdecl.fname));
  Text("1,\n"); (* only 1 dimensional array support *)
  Text("0,\n"); (* 0 offset *)
  Text("gdims,\n");
  Text("NULL,\n");
  Text("0,\n");
  Text("NULL,\n");
  Text("\n");]

let generate_cl_enqueue_read_buffer fdecl env =
  (* only one buffer to read, since there's only one output arg *)
  Environment.append env [Text(sprintf "%s *__out = (%s*) malloc(\_arr_len *
    sizeof(%s));\n" base_r_type base_r_type);
  Text("CALL_CL_GUARDED(clEnqueueBuffer,\n")
  Text("\_sheets_queue,\n");
  Text("\_out,\n");
  Text("CL_TRUE,\n"); (* blocking read *)
  Text("0,\n"); (* 0 offset *)
  Text(sprintf "sizeof(%s) * \_arr_len,\n" base_r_type);
  Text("\void * \_out,\n");
  Text("0,\n"); (* empty wait queue *)
  Text("NULL,\n");
  Text("\n");]

let generate_cl_release_list fdecl env =
  let rec generate_cl_releases arg_n formals env =
    let generate_cl_release arg_n formal env =
      (* only release array args (those alloc-ed with
       clCreateBuffer) *)
      if Generator_utilities.is_array_type formal.v_type then
        Environment.append env [Text("CALL_CL_GUARDED(\n");
          Text("clReleaseMemObject,\n");
          Text(sprintf "\_arg%d\n" arg_n);
          Text("\n");]
      else "", env
    in
    match formals with
    [] -> "", env
    | formal :: other_formals ->
      Environment.append env [Text("CALL_CL_GUARDED(\n"); (* always free output arr *)
        Text("clReleaseMemObject,\n");
        Text("\_arg1\n");
        Text("\n");
        Generator(generate_cl_release arg_n formal);
        Generator(generate_cl_releases (arg_n + 1) other_formals)]
    in
    Environment.append env [Generator(generate_cl_releases 2 fdecl.formals)]
  in
  let __arr_len = {
    v_type = "int";
    v_name = "\_arr_len";
    a_size = -1;
  } in
  Environment.append env [Text(sprintf "%s %s"
    (Generator_utilities.c_type_from_str fdecl.r_type)
    fdecl.fname);
Generator(generate_formals_vdecl_list (__arr_len :: fdecl.formals));
Text("\n\n"); Generator(generate_cl_arg_list fdecl);
Generator(generate_cl_enqueue_write_buffer_list fdecl);
Generator(generate_cl_set_kernel_args fdecl);
Generator(generate_cl_enqueue ND_range kernel fdecl);
Generator(generate_cl_enqueue read buffer fdecl);
Generator(generate_cl_release list fdecl);
Text("return __out;\n"); Text("\n")

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - * )
(* CPU functions * )
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - * )

let generate_func_formals and body stmt_list vdecl_list env =
Environment.append env [Generator(generate_formals vdecl_list vdecl_list);
Text("\n\n"); Generator(process_stmt_list (stmt_list));
Text("\n")]

let rec generate_cpu_funcs fdecls env =
let generate_cpu_func fdecl env =
match fdecl.isGfunc with
false ->
let main_checked_name = function
"main" -> "snuggle"
| other_name -> other_name
in
Environment.append env [Env(add_func
fdecl.fname (Generator_utilities.fdecl_to_func_info
fdecl));
Env(update_curr_func fdecl.fname);
Text(sprintf "%s %s("
  fdecl.r_type (main_checked_name fdecl.fname));
NewScope(generate_func_formals and body
  fdecl.body fdecl.formals)])
| true ->
Environment.append env [Env(add_Gfunc fdecl);
Env(add_func fdecl.fname
  (Generator_utilities.fdecl_to_func_info fdecl));
NewScope(generate_kernel_invocation_function fdecl)]
in
match fdecls with
[] -> "", env
| [fdecl] -> generate_cpu_func fdecl env
| fdecl :: other_fdecls ->
Environment.append env [Generator(generate_cpu_func fdecl);
  Generator(generate_cpu_funcs other_fdecls);]

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - * )
(* OpenCL Kernels * )
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - * )

(* Each gfunc has a requires a set of variables to access its
associated OpenCL kernel representation. For a gfunc called
"mygfunc", these variables are:
  - `mygfunc_kernel_string` : a string of the opencl kernel code
  - `mygfunc_kernel_name` : a string of the function name;
circuitously, this will be "mygfunc"
  - `mygfunc_compiled_kernel` : the compiled cl_kernel object

We have to declare all of these variable globally (and at the top
of our generated c program) so they will be accessible from any
cpu function.

ASSUMPTIONS:
- the incoming func_info struct is typechecked
- the first argument of the __kernel is the size for the whole function
- the second argument of the __kernel is the output array *

(* add the blocksize variables and then process the statment list *)

let rec generate_cl_kernel_body stmt_list fdecl env =
    Environment.append env
    [Env(update_scope_add_var "__block_start" Int);
     Env(update_scope_add_var "__block_end" Int);
     Env(update_scope_add_var "_id" Int);
     Text("const int _id = get_global_id(0);");
     Text(sprintf "const int __block_start = _id * %d;"
             fdecl.blocksize);
     Text(sprintf "const int __block_end = _id * %d + %d;"
             fdecl.blocksize
             fdecl.blocksize);
     Generator(process_stmt_list stmt_list);
    ]

(* return a comma separated list of kernel formal declarations and
 adds the variables to the current scope *)

let rec generate_cl_kernel_vdecl_list vdecl_list env =
    let c_type = Generator_utilities.c_type_from_str vdecl.v_type in
    let sheets_type = Generator_utilities.vdecl_type vdecl in
    Environment.append env [Env(update_scope_add_var vdecl.v_name sheets_type);
                            Text(sprintf "%d global const %s %s c_type %d vdecl.v_name)]]
    in
    match vdecl_list with
      [ ] -> "", env
    | [vdecl] -> generate_cl_kernel_vdecl vdecl env
    | vdecl :: other_vdecls ->
        Environment.append env [Generator(generate_cl_kernel_vdecl vdecl);
                                Text(",");
                                Generator(generate_cl_kernel_vdecl_list other_vdecls)]

let gfunc_to_cl_kernel_string gdecl env =
    (* we have to reject all references to variables that aren't
     immediately in scope *)
    (* we're going to have to escape double-quotes when we write
     these string literals *)
    (* here we use KernelText instead of Text, which surrounds the
     string it takes in with quote marks and adds a newline at the end
     so that the text shows up in the generated c code as a multi-line
     string literal *)
    let base_r_type = Generator_utilities.c_type_from_str gdecl.r_type in
    let sheets_r_type = Generator_utilities.str_to_type gdecl.r_type in
    Environment.append env [
        (* we have to manually modify scope because we're processing
         gfunc bodies separately from their declarations*)
        Env(update_curr_func gdecl.fname);
        Text("\"");
        Env(update_on_gpu true);
        Env(update_scope_add_var "__arr_len" Int);
        Env(update_scope_add_var "__out" sheets_r_type);
        Text("__kernel ");
        Text(sprintf "void %s(__global const int __arr_len, __global
                  %s__out,"
                  gdecl.fname base_r_type);
        Generator(generate_cl_kernel_vdecl_list gdecl.formals);
        Text("\"");
        Text("\"");
        Generator(generate_cl_kernel_body gdecl.body gdecl);
        Text("\"");
        Env(update_on_gpu false);
        Text("\"");
let gfunc_to_cl_kernel gfdecl env =
   Environment.append env [Text(sprintf "const char *%s_kernel_string =\n
gfdecl.fname);
   (* we aren't ever changing the environment
above the gfunc's scope, but we need to
generate a new scope to parse the gfunc's
contents *)
   NewScope(gfunc_to_cl_kernel_string gfdecl);
   Text(";\n");
   Text(sprintf "const char *%s_kernel_name = \"%s\";\n
gfdecl.fname gfdecl.fname);
   Text(sprintf "cl_kernel %s_compiled_kernel;\n" gfdecl.fname]]

let rec gfunc_list_to_cl_kernels gfdecl_list env =
   match gfdecl_list with
   [] -> "", env
   | gfdecl :: other_gfdecls ->
      Environment.append env [Generator(gfunc_to_cl_kernel gfdecl);
                           Generator(gfunc_list_to_cl_kernels other_gfdecls)]]

let generate_cl_kernels env =
   let cl_globals = "cl_context __sheets_context;\n^ "cl_command_queue __sheets_queue;\n^ "cl_int __cl_err;\n"
in
   Environment.append env [Text(cl_globals);
                           Generator(gfunc_list_to_cl_kernels env.gfunc_list)]

   (* ----------------------------------------------------------- *)
   (* Main: opencl context creation and frees                    *)
   (* ----------------------------------------------------------- *)

let rec generate_compile_kernels gfdecl_list env =
   let generate_compile_kernel gfdecl =
      sprintf "%s_compiled_kernel = kernel_from_string(__sheets_context,
%#kernel_string, %#kernel_name, SHEETS_KERNEL_COMPILE_OPTS);\n" gfdecl.fname
   in
   match gfdecl_list with
   [] -> "", env
   | gfdecl :: other_gfdecls ->
      Environment.append env [Text(generate_compile_kernel gfdecl);
                           Generator(generate_compile_kernels other_gfdecls)]]

let rec generate_release_kernels gfdecl_list env =
   let generate_release_kernel gfdecl =
      sprintf "CALL_CL_GUARDED(clReleaseKernel, (%s_compiled_kernel));\n" gfdecl.fname
   in
   match gfdecl_list with
   [] -> "", env
   | gfdecl :: other_gfdecls ->
      Environment.append env [Text(generate_release_kernel gfdecl);
                           Generator(generate_release_kernels other_gfdecls)]]

let generate_main env =
   Environment.append env [Text("int main()\n");
                      Text("\n");
                      Text("create_context_on(SHEETS_PLAT_NAME, SHEETS_DEV_NAME, 0,
&__sheets_context, &__sheets_queue, 0);\n"));
   Generator(generate_compile_kernels env.gfunc_list);
   Text("snuggle();\n");
   Generator(generate_release_kernels env.gfunc_list);
   Text("CALL_CL_GUARDED(clReleaseCommandQueue,
(__sheets_queue));\n");
   Text("CALL_CL_GUARDED(clReleaseContext,
8.6. Generator Utilities

(*
 * Sheets Code Generator Utilities
 * Authors: Ruchir Khaitan, Ben Barg, Amelia Brunner
 * Copyright 2014, Symposium Software
 *)

open Ast;;
open Environment;;
open Printf;;

exception Type_Error of string;;
exception NotImplemented_Error of string;;
exception Unsupported_Array_Type_Error;;

let eval_basic_binop type1 type2 =
  if (type1 = type2) then
    match type1 with
    | Int -> type1
    | Float -> type1
    | Array (Int) -> type1
    | Array (Float) -> type2
    | _ -> raise (Type_Error("Types not ID, int, or float"))
else
    raise (TypeError("Incompatible types"))

let eval_binop type1 type2 op =
match op with
  | Plus -> (eval_basic_binop type1 type2)
  | Minus -> (eval_basic_binop type1 type2)
  | Times -> (eval_basic_binop type1 type2)
  | Divide -> (eval_basic_binop type1 type2)
  | Equal -> (eval_basic_binop type1 type2)
  | Greater -> (eval_basic_binop type1 type2)
  | Less -> (eval_basic_binop type1 type2)
  | Geq -> (eval_basic_binop type1 type2)
  | Leq -> (eval_basic_binop type1 type2)
  | _ -> raise (TypeError("Incompatible types"))

let eval_array_acc array_ int_expr =
match array_ with
  | Array ( Int ) ->
    (match int_expr with
      | Int -> Int
      | _ -> raise (TypeError("Cannot access element in array with non-int datatype")))
  | Array ( Float ) ->
    (match int_expr with
      | Int -> Float
      | _ -> raise (TypeError("Cannot access element in array with non-int datatype")))
  | _ -> raise (TypeError("Cannot access element in non-array type"))

(* Returns typeof block access *)
let typeof_block_acc id env =
match id with
  | "start" -> Int
  | "end" -> Int
  | "out" -> (Environment.return_typeof_func_env.current_function_env env)
  | _ -> raise (TypeError("Invalid block access"))

let rec expr_typeof expr env =
match expr with
  | Literal_int(i) -> Int
  | Literal_float(f) -> Float
  | Literal_int_a(i_a) -> Array( Int )
  | Literal_float_a(i_a) -> Array(Float)
  | Id(s) -> Environment.typeof s env
  | Binop(exp1, op, exp2) -> (eval_binop (expr_typeof exp1 env) (expr_typeof exp2 env) op)
  | ArrayAcc(exp1, exp2) -> (eval_array_acc (expr_typeof exp1 env) (expr_typeof exp2 env))
  | Call(func_id, expr_list ) -> (typeof_func_call func_id expr_list
                                 (Environment.get_func_args func_id env env))
  | BlockAcc(id, _) -> (typeof_block_acc id env)
  | _ -> raise (NotImplementedError("Undefined type of expression"))

and typeof_func_call func_id expr_list arg_list env =
(* First make sure that all of the arguments are valid, then check
 all ids, then return the type of the function *)
let rec check_expr_list expr_list arg_list =
match expr_list, arg_list with
  | [],[] -> Environment.return_typeof_func func_id env
  | expr1::other_exps, arg1::other_args ->
    if((expr_typeof expr1 env) != arg1 ) then
      raise (TypeError("Function arguments of incorrect type"))
    else
      check_expr_list other_exps other_args
  | _ -> raise (TypeError("Incorrect number of function arguments"))

in
match func_id with
  "printf" -> Int
| _   -> check_expr_list expr_list arg_list

let str_to_type str =
  match str with
  "int" -> Int
| "float" -> Float
| "int[]" -> Array(Int)
| "float[]" -> Array(Float)
| _   -> raise (NotImplementedError("Unrecognized type " ^ str))

let rec typecheck_stmt stmt env =
  true

let rec typecheck_stmt_list stmt_list env =
  [] -> true
| stmt :: rest_of_stmts -> typecheck_stmt stmt env;
    typecheck_stmt_list rest_of_stmts env

let vdecl_type vdecl =
  str_to_type vdecl.v_type

let rec vdecl_list_to_type_list vdecl_list =
  match vdecl_list with
  []       -> []
| vdecl::rest_of_vdecls -> (vdecl_type vdecl)::(vdecl_list_to_type_list rest_of_vdecls)

let rec vdecl_list_to_string_list vdecl_list =
  match vdecl_list with
  []       -> []
| vdecl::rest_of_vdecls -> (vdecl.v_name)::(vdecl_list_to_string_list rest_of_vdecls)

let fdecl_to_func_info fdecl =
  {
    id = fdecl.fname;
    gpu = fdecl.isGfunc;
    return = str_to_type fdecl.r_type;
    args = vdecl_list_to_type_list fdecl.formals;
    arg_names = vdecl_list_to_string_list fdecl.formals;
    _blocksize = fdecl.blocksize;
  }

let arr_type_str_to_base_type = function
  "float[]" -> "double"
| "int[]"   -> "int"
| "float[][]" -> "double"
| "int[][]"   -> "int"
| _   -> raise UnsupportedArrayType

let c_type_from_str = function
  "int"   -> "int"
| "float" -> "double"
| "float[]" -> "double **"
| "int[]"   -> "int *"
| "float[][]" -> "double ***"
| "int[][]"   -> "int **"
| _   -> raise UnsupportedArrayType

let is_array_type = function
  "int" | "float" -> false
| _   -> true
8.7. Environment Types

(*
 * Sheets Environment Types
 *
 * Authors: Ruchir Khaitan, Ben Barg, Amelia Brunner
 * Copyright 2014, Symposium Software
 *)

open Ast;;

module VariableMap = Map.Make(String);;
module FunctionMap = Map.Make(String);;

exception EmptyEnvironmentError;;
exception NameAlreadyBoundError of string;;
exception VariableNotFound of string;;
exception VariableAlreadyDeclared;;
exception AlreadyDeclaredError;;
exception FunctionNotFoundDefinitionError;;
exception ReservedWordError of string;;

type func_info = {
  id : string;
  gpu : bool;
  return : datatype;
  args : datatype list;
  arg_names: string list;
  _blocksize : int;
}

(* Record indicating what the current environment keeps track of *)
type env = {
  var_stack: datatype VariableMap.t list;
  func_return_type_map: func_info FunctionMap.t;
  current_function: string;
  on_gpu: bool;
  gfunc_list: fdecl list; (* we need to save the body of the
                            gfunc so we can do kernel string
                            generation *)

  num_array_returns : int;
  var_array_sizes : int VariableMap.t;
}

(* Types that can be returned by the generator as it modifies either
the text of the generated code changes the environment as it is
parsing the file or passes functions to edit both code and
environment either in the existing scope or in a new scope *)
type source =
  | Text of string
  | Env of (env -> env)
  | Generator of (env -> (string * env))
  | NewScope of (env -> (string * env))

(* Create initializes an empty record for environment *)
let add_default_func f_id fmap =
  let f_info = {
    id = f_id;
    gpu = false;
    return = Int;
    args = [];
    arg_names = [];
    _blocksize = -1;
  }
in
(FunctionMap.add f_id f_info fmap)

let init_func_map =
  let time_start_fmap =
    let printf_fmap =
      add_default_func "printf" FunctionMap.empty
    in
      add_default_func "time_start" printf_fmap
    in
      add_default_func "time_end" time_start_fmap
  in
    create =
      { var_stack = VariableMap.empty::[];
        func_return_type_map = init_func_map;
        current_function = ""; (* TODO maybe this needs a better convention *)
        on_gpu = false;
        gfunc_list = [];
        num_array_returns = 0;
        var_array_sizes = VariableMap.empty;
    }

  (* Update gives a new env record with updated values of the record *)
  let update v_stack f_map curr_f gpu g_list num_arr_ret v_a_s =
    { var_stack = v_stack;
      func_return_type_map = f_map;
      current_function = curr_f;
      on_gpu = gpu;
      gfunc_list = g_list;
      num_array_returns = num_arr_ret;
      var_array_sizes = v_a_s;
    }

  (* Functions that let us modify only one variable in environment at a time *)
  let update_only_scope new_scope env =
    update new_scope env.func_return_type_map env.current_function env.on_gpu
    env.gfunc_list env.num_array_returns env.var_array_sizes

  let update_only_func new_func env =
    update env var_stack new_func env.current_function env.on_gpu env.gfunc_list
    env.num_array_returns env.var_array_sizes

  let update_curr_func new_curr_func env =
    update env var_stack env.func_return_type_map new_curr_func env.on_gpu env.gfunc_list
    env.num_array_returns env.var_array_sizes

  let update_on_gpu gpu env =
    update env var_stack env.func_return_type_map env.current_function gpu env.gfunc_list
    env.num_array_returns env.var_array_sizes

  let update_gfunc_list g_list env =
    update env var_stack env.func_return_type_map env.current_function env.on_gpu g_list
    env.num_array_returns env.var_array_sizes

  let increment_num_array_rets env =
    update env var_stack env.func_return_type_map env.current_function env.on_gpu
    env.gfunc_list (env.num_array_returns + 1) env.var_array_sizes

  let add_array_size id size env =
    update env var_stack env.func_return_type_map env.current_function env.on_gpu
    env.gfunc_list env.num_array_returns (VariableMap.add id size env.var_array_sizes)

  (* Checks all scopes to see if variable has been declared *)
  let is_var_in_scope id env =
let rec check_scopes scope_stack =
  let check_level scope_level =
    VariableMap.mem id scope_level
  in
  match scope_stack with
  | [] -> false
  | [scope_level] ->
    if env.on_gpu then false
    else check_level scope_level
  | scope_level :: other_scope_levels ->
    if check_level scope_level then true
    else check_scopes other_scope_levels
  in check_scopes env.var_stack

(* Checks all scopes to find variable and returns type if found
 * Raises exception if not found *)
let typeof id env =
  let rec check_scopes scope_stack =
    match scope_stack with
    | [] -> raise (VariableNotFoundException id)
    | scope_level :: other_scope_levels ->
      if VariableMap.mem id scope_level then
        VariableMap.find id scope_level
      else
        check_scopes other_scope_levels
    in check_scopes env.var_stack

(* Adds a variable to the topmost level of the variable stack. Does
 * not check if variable is already in stack do elsewhere raises error
 * if stack is an empty list returns an updated environment with an
 * updated scope stack *)
let update_scope_add_var id datatype env =
  let old_scope, scope_tail =
    (match env.var_stack with
     | old_scope :: scope_tail -> old_scope, scope_tail
     | [] -> raise EmptyEnvironmentException ) in
  let new_scope = VariableMap.add id datatype old_scope in
  update_only_scope (new_scope::scope_tail) env

(* Handles adding a variable to the current environment.

  Takes a tuple of
  - id : string
  - datatype
  - (text: string, env : env)

  and either raises
  - EmptyError
  - AlreadyDeclaredError

  or it adds the variable to the current top scope with
  update_scope_add_var and returns a updated env *)
let add_var id datatype env =
  match env.var_stack with
  | [] -> raise EmptyEnvironmentException
  | scope_level :: scope_tail ->
    if VariableMap.mem id scope_level then
      raise VariableAlreadyDeclared
    else
      update_scope_add_var id datatype env

let get_arr_size id env =
  if VariableMap.mem id env.var_array_sizes then
    (VariableMap.find id env.var_array_sizes)
  else
    raise VariableAlreadyDeclared
(* adds a new empty variableMap to the top of var stack used to enter
  a subscope takes an env, returns an updated env *)
let push_scope env =
    update_only_scope (VariableMap.empty::env.var_stack) env

(* removes a variableMap from the top of var stack used to enter a
  subscope takes an env, returns an updated env *)
let pop_scope env =
    match env.var_stack with
    | popped_scope::otherscopes ->
      update_only_scope otherscopes env
    | [] -> raise EmptyEnvironmentError

(* --------------------------------------------------------------------- *)
(* The following methods deal with handling the function map *)
(* --------------------------------------------------------------------- *)

(* Takes a function id and checks if the function is defined
  returns bool *)
let is_func_declared id env =
    FunctionMap.mem id env.func_return_type_map

(* Takes a function id and either returns the mapped function info
  or raises an undefined function error *)
let get_func_info id env =
    if is_func_declared id env then
      FunctionMap.find id env.func_return_type_map
    else
      raise FunctionNotFoundException

let get_func_args id env =
  (get_func_info id env).args

(* Inserts a new function to the function map and updates environment
  or raises a AlreadyDeclaredError *)
let add_func id finfo env =
    if is_func_declared id env then
      raise VariableAlreadyDeclared
    else
      update_only_func (FunctionMap.add id finfo env.func_return_type_map) env

(* Returns the datatype of a function or
  raises a undefined error if the function is not defined *)
let return_typeof_func id env =
    let f_info =
      get_func_info id env in
    f_info.return

let id_is_gfunc id env =
    let rec name_in_gfunc_list gfunc_list =
      match gfunc_list with
      | [] -> false
      | gfunc :: other_gfuncs ->
        if id = gfunc.fname then true
        else name_in_gfunc_list other_gfuncs
    in
    name_in_gfunc_list env.gfunc_list

let rec check_gfunc_name_in_list glist gfunc_fdecl =
    match glist with
    | [] -> false
    | gfunc_fdecl :: rest_of_gfuncs -> if gfunc_fdecl.fname = gfunc_fdecl.fname then true
    else (check_gfunc_name_in_list rest_of_gfuncs gfunc_fdecl)

let is_gfunc_declared gfunc_fdecl env =
    if check_gfunc_name_in_list env.gfunc_list gfunc_fdecl then
raise (AlreadyDeclaredError)
else if is_func_declared gfunc_fdecl.fname env then
    raise (AlreadyDeclaredError)
else false

let add_gfunc gfunc_fdecl env =
    if (is_gfunc_declared gfunc_fdecl env) then
        raise (AlreadyDeclaredError)
    else if (gfunc_fdecl.fname = "main") then
        raise (ReservedWordError("a gfunc cannot be the main method"))
    else
        update_gfunc_list (gfunc_fdecl :: env.gfunc_list) env

let quote_and_strip_newline str =
    let len = String.length str in
    let last_char s = s.[(len - 1)] in
    if str = "" then ""
    else
        match last_char str with
        | '\n' -> (String.sub str 0 (len - 2))
        | _ -> str

(* This is the running loop of the codegen step. First, f is a
function that takes a (text, environment) tuple and matches it with
a component which is either

* some new text - gets appended to existing text
* some modified environment - replaces existing environment
* some function named gen applied to the same scope

gen is applied to the existing environment and gen returns a tuple
of (string, env). The text gets appended to the existing text, and
the new env replaces the old env.

Note that gen can be a function also with arguments given to it
then this function f is applied to component in the list of
components always returning an updated (text, env) tuple that is
passed to the next component in the list *)

let append init_env components =
    let f (text, env) component =
        match component with
        | Text(str) -> if env.on_gpu then
            text ^ (quote_and_strip_newline str), env
        else
            text ^ str, env
        | Env (env_gen) -> let new_env = env_gen env in
            text, new_env
        | Generator(gen) -> let new_str, new_env = gen env in
            text ^ new_str, new_env
        | NewScope(gen) -> let new_str, new_env = gen (push_scope env) in
            text ^ new_str, pop_scope new_env in
            List.fold_left f("", init_env) components
    ;;

### 8.8. Testing Script

```
#!/bin/bash
# Sheets Automated Testing Script
# Author: Gabriel Blanco
# Copyright 2014, Symposium Software
```
executable="/test_env.sh"
output_file="envtests.output"
rm -f *.proc.sh
rm -f envtests.output
environment_tests=$(find . -name "*.sht")

path_to_name()
{
    local fullpath=$1
    test_path="${fullpath%.*}"  # Strip Extension
    test_name="${test_path##*/}"  # Strip Preceding Path
}

done

echo "Running all environment tests in current directory"

echo ""
echo "  Note: an 'x' next to a test indicates that the test"
echo "  is failing when it is expected to pass, or passing"
echo "  when expected to fail. This does not guarantee that"
echo "  a successful test is passing or failing for the"
echo "  expected reason, so make sure to verify that all"
echo "  outputs are correct"
echo ""

echo "Environment Testing Suite" >> $output_file

for file in $environment_tests ; do
    path_to_name $file
    check_if_fail $test_name

    echo "" >> $output_file
    echo "================================" >> $output_file

    VAR=$( .=./executable $test_name 2>&1 )
    echo "$VAR" >> $output_file

    fatal_error=$(echo $VAR | grep 'Fatal error:')
    if [[ $fatal_error == *' ]]] ; then
        if [[ $fatal_error == *'' ]]] ; then
            echo "[x] Running Test: $test_name"
        else
            echo "[ ] Running Test: $test_name"
        fi
    else
        if [[ $fatal_error == *'' ]]] ; then
            echo "[ ] Running Test: $test_name"
        else
            echo "[x] Running Test: $test_name"
        fi
    fi

    echo "================================" >> $output_file
    echo "" >> $output_file
done

echo "done"
echo "OUTPUT FILE: '$output_file'"
rm -f *.proc.sht
exit