SMPL - Simple Parallel Language

Final Report

Ajay S S Reddy Challa, Andrei Papancea, Devashi Tandon
{ac3647, alp2200, dt2450} @columbia.edu

December 20, 2013
Contents

1. Introduction
   1.1. Motivation
   1.2. Language Description

2. Tutorial
   2.1. A “Simple” Example
   2.2. Compiling and Running SMPL Programs
   2.3. “THE” Example

3. LRM
   3.1. Lexical Conventions
       3.1.1. Identifiers
       3.1.2. Keywords
       3.1.3. Comments
   3.2. Types
       3.2.1. Fundamental Types
       3.2.2. Derived Types
       3.2.3. Type casting
       3.2.4. Storage Classes
   3.3. Expressions
       3.3.1. Operators
       3.3.2. Precedence
   3.4. Syntax
       3.4.1. Statements
       3.4.2. Variable Declaration
       3.4.3. Function Declaration
       3.4.4. Parallel Constructs
           3.4.4.1. spawn
           3.4.4.2. barrier
           3.4.4.3. lock
           3.4.4.4. pfor

4. Project Plan
   4.1. Team Responsibilities
   4.2. Project Timeline
   4.3. Software Development Environment
   4.4. Processes followed
   4.5. Project Log

5. Architectural Design
5.1. Architecture
5.2. Error Recovery
5.3. Optimizations

6. Test Plan
6.1. Testing Phases
   6.1.1. Unit Testing
   6.1.2. Integration Testing
   6.1.3. System Testing
6.2. Automation
6.3. Test Suites

7. Lessons Learned
7.1. Ajay
7.2. Andrei
7.3. Devashi

8. References
1. Introduction

1.1 Motivation

Up until 2006, computer scientists benefited from what is referred to as the “free lunch”\(^1\). Programmers did not have to focus too much on writing very efficient code, since it was known that every two years processor speeds will increase and hence help their programs run faster. According to Moore’s Law\(^2\) the number of transistors on integrated circuits doubles approximately every two years. This law still holds today, but it is not practical to increase processor speed anymore because of financial and physical constraints such as the high cost of dealing with excessive heat dissipation. Subsequently, in a multicore world, the computer scientists all over are looking at parallel programming languages as the solution.

Two issues with parallel programming are that it is not trivial given the current toolsets and that not every problem can be parallelized. Furthermore, most popular programming languages such as C, C++, Java, or Python either do not support parallel programming or they have a very complex implementation of it, since they were not built with that idea in mind. Due to the non-trivial syntax of most parallel programming -capable languages, the concept of parallel programming is rarely taught at the undergraduate level — and it is crucial to teach this concept at the undergraduate level in order to stir interest in the field and push for advancement. Hence, there is a need for programming languages with simpler syntax, that can better illustrate parallel programming concepts and that makes them more suitable for undergraduate teaching\(^3\).

1.2 Language Description

This document describes Simple Parallel Language (SMPL) which is an approach to parallel programming that is based on a subset of the C programming language. Its major strength is that it has a clearer and simpler syntax for doing parallel programming than programs written using OpenMP or POSIX threads.

With the addition of only four keywords to the C language, SMPL gives the programmer the ability to parallelize his code with minimal effort.

---


2. Tutorial

2.1 A “Simple” Example

Here is a quick “Hello world!” program in SMPL:

```c
int main()
{
    printf("Hello world!\n");
}
```

Notice anything? Yes, SMPL uses the same syntax as C. Another thing that you should notice is that there is no need to include anything. All the includes, given the features of the language are added for you for convenience.

Let’s spice things up a bit, and write a more exciting “Hello world!” program:

```c
say(string str){
    printf("%s\n",str);
}

int main()
{
    spawn say(“Hello”);
    spawn say(“world”);
    spawn say(“user!”);
    barrier;
    printf("Done!\n");
}
```

The program above prints the strings “Hello”, “world”, “user!” in an undetermined order, followed by the string “Done!””. What spawn does is creates a new thread for the given function call. Hence, the function say gets called three times in three different threads, but it is not possible to predict which of the three threads will finish first. The barrier statement prevents “Done!” from beings printed before all the other threads have finished executing.

A few last things to notice before we move on are: void type is optional for function definitions, which is why the say function does not have a type, and string is a type that does not exist in C, so we implemented it in SMPL for simplicity — see what we did there? Here’s the generated C code for the SMPL code above:
/* Code Generated from SMPL */
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

void say(char * str);
void *thread_0(void *args);
void *thread_1(void *args);
void *thread_2(void *args);

pthread_t threads[3];

void *thread_0(void *args){
    void **arg_list = (void **)args;
    char * var0 = *(char **)arg_list[0];
    say(var0);
}

void *thread_1(void *args){
    void **arg_list = (void **)args;
    char * var0 = *(char **)arg_list[0];
    say(var0);
}

void *thread_2(void *args){
    void **arg_list = (void **)args;
    char * var0 = *(char **)arg_list[0];
    say(var0);
}

int main(){
    {void *args_thread_0[1];
     char * arg_lit_0_0 = "Hello";
     args_thread_0[0] = (void *)&arg_lit_0_0;
     pthread_create(&threads[0],NULL,thread_0,(void *)args_thread_0);}
    {void *args_thread_1[1];
     char * arg_lit_1_0 = "world";
     args_thread_1[0] = (void *)&arg_lit_1_0;
     pthread_create(&threads[1],NULL,thread_1,(void *)args_thread_1);}
    {void *args_thread_2[1];
     char * arg_lit_2_0 = "user!";
     args_thread_2[0] = (void *)&arg_lit_2_0;
2.2 Compiling and Running SMPL Programs

First, SMPL programs use .smpl as an extension, by convention. You have multiple options of interacting with the compiler (smplc) by using the following command line arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a [-t</td>
<td>filename]</td>
</tr>
<tr>
<td>-c [-t</td>
<td>filename]</td>
</tr>
<tr>
<td>-s [-t</td>
<td>filename]</td>
</tr>
</tbody>
</table>

In the table above -t tells the compiler to take input from the terminal and filename tells the compiler to compile the given file. The flags -a, -c, and -s must be followed by either -t or a filename. The SMPL compiler will output C code.

For a more concrete example, open your favorite text editor, paste any of the two “Hello world” programs above into the new file and save it as hello.smpl. Open the command line and run 

```
./smplc -c hello.smpl > hello.c
```

What “> hello.c” does is it redirects the output of the compiler to a file, rather than printing it in the terminal window.

Finally, to finish the compilation of your “Hello world” program run gcc -pthread hello.c. Note that trying to compile the program without the -pthread flag will fail, because the C output of SMPL uses the Posix Threads library to run programs in parallel.
2.3 “THE” Example

Here is a quick introduction to the four keywords that SMPL uses to perform parallelism:

1. **spawn**: tells the compiler that the function that spawn precedes should run in parallel with the caller thread.
2. **barrier**: causes all previously spawn-ed threads to wait for each other to complete before the program can continue
3. **lock**: synchronization mechanism that prevents multiple threads from changing a variable concurrently
4. **pfor**: a for loop whose work is divided amongst a specified number of threads

Now let’s take a look at one more example. The SMPL program below computes the sum of all the prime numbers up to one million.

```c
int sum = 0;
int isPrime(int n){
    int limit = n/2;
    int i = 0;
    for(i=2; i<=limit; i++)
        if(n%i == 0)
            return 0;
    return 1;
}
int main(){
    int i;
    int n = 1000000;
    pfor(8; i; 1; n){
        if(isPrime(i))
            sum = sum+i;
    }
    printf("The sum of the first 1M primes is %d.\n",sum);
}
```

Here’s what you need to know about the code above:

1. it has the same length as the equivalent C code (in terms of functionality, not performance)
2. it uses only one SMPL keyword
3. the SMPL program completed in 27.173 seconds.
4. the equivalent C program completed in 1 minute and 11.340 seconds.

Both programs ran on the CLIC machines using the same compiler and includes. In conclusion, with very limited effort you can turn a serial C program into a parallel one. In this case, the SMPL program ran more than twice as fast than its C counterpart. Here’s the C code generated for the SMPL code above is:

```c
/* Code Generated from SMPL */
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

int sum = 0;

int isPrime(int n);
void *thread_0(void *args);

pthread_t threads[1];

void *thread_0(void *args)
{
    int temp_sum = 0;
    pthread_mutex_t pfor_global_lock=PTHREAD_MUTEX_INITIALIZER;
    void **arguments = (void **)args;
    int lower = *((int *)arguments[0]);
    int upper = *((int *)arguments[1]);
    void **thread_0_local_vars = (void **)arguments[2];
    int i = *((int *)thread_0_local_vars[0]);
    int n = *((int *)thread_0_local_vars[1]);
    for(i=lower; i<upper; i++){
        if(isPrime(i)) temp_sum=temp_sum+i;
    }
    pthread_mutex_lock(&pfor_global_lock);
    sum=sum+temp_sum;
    pthread_mutex_unlock(&pfor_global_lock);
    pthread_exit(NULL);
}

int main()
{
    int i;
    int n=1000000;
```
int num_threads_0 = 8;
int pfor_init_0 = 1;
int pfor_limit_0 = n;

pthread_t pfor_threads_0[num_threads_0];
int pfor_uppers_0[num_threads_0];
int pfor_lowers_0[num_threads_0];
void *pfor_args_0[num_threads_0][3];
void *pfor_local_vars_0[2];
pfor_local_vars_0[0] = (void *)&i;
pfor_local_vars_0[1] = (void *)&n;
int pfor_i_0;

for (pfor_i_0=0; pfor_i_0<num_threads_0; pfor_i_0++){
    pfor_uppers_0[pfor_i_0] = pfor_init_0+((pfor_init_0+pfor_limit_0)/num_threads_0)*(pfor_i_0+1)+((pfor_init_0+pfor_limit_0)%num_threads_0);
    pfor_lowers_0[pfor_i_0] = pfor_init_0+((pfor_init_0+pfor_limit_0)/num_threads_0)*pfor_i_0;
    pfor_args_0[pfor_i_0][0] = (void *)&pfor_lowers_0[pfor_i_0];
    pfor_args_0[pfor_i_0][1] = (void *)&pfor_uppers_0[pfor_i_0];
    pfor_args_0[pfor_i_0][2] = (void *)&pfor_local_vars_0;
}

for (pfor_i_0=0; pfor_i_0<num_threads_0; pfor_i_0++){
    pthread_create(&pfor_threads_0[pfor_i_0],NULL,thread_0,(void *)&pfor_args_0[pfor_i_0]);
}

for (pfor_i_0=0; pfor_i_0<num_threads_0; pfor_i_0++){
    pthread_join(pfor_threads_0[pfor_i_0],NULL);
}
printf("The sum of the first 1M primes is %d.\n",sum);

int isPrime(int n){
    int limit=n/2;
    int i=0;
    for(i=2; i<=limit; i++) if(n%i==0) return 0;
    return 1;
3.1 Lexical Conventions

SMPL introduces four parallel keywords. SMPL supports four types of tokens: identifiers, keywords, functions, and separators. Tabs, and newlines are ignored, while multiple spaces are contrived to only one space, which serves as a token separator.

3.1.1 Identifiers

An identifier consists of a letter followed by one or more alphanumeric and/or underscore characters. The identifiers are case-sensitive.

3.1.2 Keywords

The following set of keywords are reserved names and cannot be used as identifiers:

- int
- float
- boolean
- char
- string
- void
- true
- false
- null
- if
- else
- for
- while
- return
- break
- main
- pfor
- spawn
- lock
- barrier

3.1.3 Comments

Comments are represented in a block as follows:

/* single line comment */

/* multi
   line
   comment */

SMPL does not support comment nesting.
3.2 Types

3.2.1 Fundamental Types

In SMPL the following fundamental data types are supported:

➔ int: integer variable
➔ float: floating-point variable
➔ boolean: boolean variable (true, false)
➔ char: single ASCII character variable
➔ string: multi ASCII character variable

In addition, SMPL uses void to symbolize that a function has not return value.

3.2.2 Derived Types

In SMPL the following derived data types are supported:

➔ functions: sequence of code having its own execution context

Refer to section 5.3 and 5.4 on how to use these types.

3.2.3 Type Casting

Type casting is not directly supported in SMPL. So if the function takes a particular type of arguments, the variables used to call the function should be of the same type. However, a larger size variable can be assigned value from a lower size variable but not the other way. So it is legal to assign char to int variable or int to float variable but int variable cannot be assigned to char. Also, strings can be assigned to only strings. The following example illustrates how to pass variables of different types as arguments to a function:

To do something like this:

funca(int,int)

int a; boolean b;
call funca(a, (int) b)
Do:

```c
int c = b;
call funca(a, c)
```

### 3.2.4 Storage Classes

In SMPL the scope of a fundamental type can be either of the following two storage classes:

- **local**: scope is limited to the function
- **global**: accessible in any part of the program

The scope is derived from where it is defined. If the variable is defined outside the function the scope is global, whereas if it is defined within a function the scope is local.
3.3. Expressions

3.3.1 Operators

The table below lists the operators supported by SMPL:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>++</td>
<td>unit incrementation</td>
</tr>
<tr>
<td>-=</td>
<td>unit decrementation</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than comparison</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than comparison</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to comparison</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to comparison</td>
</tr>
<tr>
<td>==</td>
<td>equality comparison</td>
</tr>
<tr>
<td>!=</td>
<td>inequality comparison</td>
</tr>
<tr>
<td>!</td>
<td>boolean NOT operator</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>boolean AND operator</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>,</td>
<td>argument separator</td>
</tr>
<tr>
<td>;</td>
<td>statement separator</td>
</tr>
</tbody>
</table>
### 3.3.2 Precedence

The table below illustrates the precedence for all the operators supported by SMPL:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Expressions/Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>highest</em></td>
<td>( f(arg, arg, \ldots) )</td>
</tr>
<tr>
<td></td>
<td>(!b)</td>
</tr>
<tr>
<td></td>
<td>( n++ \quad n-- \quad n \cdot o \quad n \div o \quad i % j )</td>
</tr>
<tr>
<td></td>
<td>( n + o \quad n - o )</td>
</tr>
<tr>
<td></td>
<td>( n &lt; o \quad n &gt; o \quad n \leq o \quad n \geq o )</td>
</tr>
<tr>
<td></td>
<td>( r == r \quad r != r )</td>
</tr>
<tr>
<td></td>
<td>( b &amp;&amp; c )</td>
</tr>
<tr>
<td></td>
<td>( b \mid\mid c )</td>
</tr>
<tr>
<td></td>
<td>( l = r )</td>
</tr>
<tr>
<td><em>lowest</em></td>
<td>( r1, r2 )</td>
</tr>
</tbody>
</table>

The table above illustrates how expressions are evaluated. Therefore these rules define implicit grouping, so the expression 1+2*3 is equivalent to 1+(2*3). If a different grouping is desired, such as \((1+2)*3\), then it needs to be explicitly specified using a pair of parentheses.
3.4. Syntax

3.4.1 Statements

A statement can have any of the following formats:

- expression;
- { statement-list }
- if ( expression ) statement else statement
- while ( expression ) statement
- for ( expression ; expression ; expression ) statement
- pfor ( literal ; expression ; expression ; expression ) statement
- lock statement
- spawn statement
- barrier;
- break;
- continue;
- return expression;

3.4.2 Variable Declaration

The syntax for variable declaration in SMPL has the following format:

```
type identifier; /* the identifier will be initialized to a default value */
type identifier = value; /* the identifier will be initialized to the specified value */
```

The default value for different data types is as follows:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>float</td>
<td>0.0</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>char</td>
<td>0 (null character)</td>
</tr>
<tr>
<td>string</td>
<td>&quot;&quot; (empty string)</td>
</tr>
</tbody>
</table>
3.4.3 Function Declaration

The syntax for function declaration in SMPL has the following format:

```
return-type identifier(arg-type arg1, arg-type arg2, ...){
  declaration-list
  statement-list
}
```

The return-type of a function can be any of the fundamental types, including void. The arg-type can be any of the fundamental types, excluding void.

3.4.4 Parallel Constructs

3.4.4.1 spawn

The spawn statement creates a thread for the given statement. Its syntax looks as follows:

```
spawn function_call;
```

spawn is transformed into C code by creating a new thread using pthread_create. The code generator creates an array of void pointers to store all the variables passed as arguments to spawn, if any. Furthermore, if the variables passed as arguments are literals, the code generator will place the literals in variables so their pointer can be placed in the array of arguments. A new wrapper function is created in which the actual function call is called. We did this because pthread_create requires a function whose return type is void. Last, the reference to this new thread is stored in an array, that will later be used to join threads.

3.4.4.2 barrier

The barrier statement prevents execution of code following it until all the threads spawned prior to it finish executing. Its syntax looks as follows:

```
barrier;
```

Here is a code example of when you would need barrier:

```
spawn thread1();
```
spawn thread2();

print(“Done!”);

Note that in the example above, thread1 and/or thread2 could finish after the printing is done, which is probably an undesired result. Subsequently, using a barrier above the print statement, would prevent the program execution to reach the latter before all the spawned threads have completed. The fix would look like this:

spawn thread1();
spawn thread2();
barrier;

print(“Done!”);

barrier is transformed into C code using a for loop that iterates through all the array of threads. Inside that loop, each thread is joined using pthread_join.

3.4.4.3 lock

The lock statement prevents other threads from accessing or modifying the contents of the statement that it precedes until the latter’s computation finishes. Its syntax looks as follows:

lock statement

Once the computation finishes, the lock will be automatically removed.

Here is a code example of when you would need a lock:

int sum=0;

int add_to_sum(int val){
    sum = sum+val;
}

int main(){
    int i;

pfor(4;i;0;10000){
    add_to_sum(i);
}
printf("The sum is %d\n",sum);
}

Note that in the example above, each of the four threads that pfor creates will attempt to modify sum at the same time, creating concurrency problems and therefore outputting a different result every time the program runs. Subsequently, using a lock around the sum = sum+val statement, would prevent a thread from modifying the variable while another is already trying to change it. The fix would look like this:

int sum=0;
int add_to_sum(int val){
    lock sum = sum+val;
}

int main(){
    int i;
    pfor(4;i;0;10000){
        add_to_sum(i);
    }
    printf("The sum is %d\n",sum);
}

lock is transformed into C code by creating a new pthread_mutex and putting pthread_mutex_lock / unlock statements around the statement that is being locked.

3.4.4.4 pfor

The pfor statement defines a for loop that splits up the work in its body into multiple threads. Its syntax has the following format:

pfor(k; counter; init; limit)
    statement

In the example above, k represents the number of threads that pfor will break its body into. For instance, if k=4 then the work of the statements in the body of pfor will be broken into 4 separate
threads. `counter` represents the counter variable name that will be used within the `pfor` body (i.e. `i`). `init` represents the initialization of the loop counter, and `limit` represents the upper limit/stop condition.

The `pfor` construct uses an implicit `barrier` in order to ensure that all the threads have finished before proceeding to the next statement. The `pfor` construct also places an implicit `lock` on all the assignment operations. Nonetheless, the `pfor` construct assumes a unit-incrementing loop. Basically, the `pfor` loop defined above would be equivalent to the following pseudo-code:

```plaintext
pfor(k; counter=init; counter < limit; counter++)
 statement
```

Note that you should be cautious about spawning threads inside `pfor` since that would entail spawning threads inside other threads and create unexpected concurrency problems.

`pfor` is transformed into C code by using a fusion of how the other parallel constructs are generated. First, the `for` loop boundaries for each thread are created and saved in an array of void pointers. In this array of pointers we also save the pointers to all the local variables in the function in which `pfor` is defined. We do this in order to replicate the environment in the thread function that is generated and that will be spawned `k` times. Next, we generate a for loop that iterates `k` number of times and spawns the thread function with the specific upper and lower bound values for that thread. The thread function is generated similar to how the `spawn` thread function is generated, with the addition that global variable assignments inside the `pfor` body will have an implicit lock placed on them. Lastly, we generate the equivalent of a barrier, so that all the spawned threads are joined before the execution of the program continues.

For the cases that `pfor` cannot parallelize (i.e. computations that are reliant on previous steps), you can implement the `pfor` functionality using spawns, barriers, and locks.
4. Project Plan

4.1 Team Responsibilities

The team responsibilities were divided as follows:

<table>
<thead>
<tr>
<th>Team Member</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajay</td>
<td>syntax definition, test script setup, test case creation</td>
</tr>
<tr>
<td>Andrei</td>
<td>code generation, code generation optimization, documentation, presentation</td>
</tr>
<tr>
<td>Devashi</td>
<td>semantic analysis, optimization, compiler front end, building the “lego blocks aka “Hello World” Program</td>
</tr>
</tbody>
</table>

This is a rough outline, since there were improvements made by each of the members in multiple areas.

4.2 Project Timeline

The project timeline for SMPL is shown below:

<table>
<thead>
<tr>
<th>Date</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 18</td>
<td>Language Proposal complete</td>
</tr>
<tr>
<td>October 26</td>
<td>Language Reference Manual complete</td>
</tr>
<tr>
<td>November 11</td>
<td>scanner, parser, AST complete</td>
</tr>
<tr>
<td>November 13</td>
<td>“Hello World” program runs</td>
</tr>
<tr>
<td>November 15</td>
<td>automated testing script complete</td>
</tr>
<tr>
<td>November 23</td>
<td>spawn, lock, barrier complete</td>
</tr>
<tr>
<td>December 12</td>
<td>semantic analysis complete</td>
</tr>
<tr>
<td>December 14</td>
<td>optimization complete</td>
</tr>
<tr>
<td>December 16</td>
<td>pfor, code generation, debugging complete</td>
</tr>
<tr>
<td>December 17</td>
<td>final report complete</td>
</tr>
</tbody>
</table>
4.3 Software Development Environment

The programming and development environments for SMPL are:
   1. Development environment: vim and emacs on the CLIC machines
   2. Programming environment: OCaml, version 3.12.1

4.4 Processes followed

The following processes were followed for the development of the compiler:

   2. Coding style for Ocaml Code: General Vim editor indentation and formatting
   3. Coding style for output C code: Four space indentation for each code block
   4. Development Process: We broke down the code into two components: Section 1 was scanner/parser/AST and section 2 was the semantic verification and code generation. For section 2 we prepared the basic code blocks by building the “Hello World” program and then filling in the pieces
   5. Testing process: As the code was developed test cases were written to test the code. Some test cases were borrowed from microC and migrated to SMPL syntax, to test the basic C constructs.

4.5 Project Log

<table>
<thead>
<tr>
<th>Date</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 10</td>
<td>language idea defined</td>
</tr>
<tr>
<td>September 18</td>
<td>Language Proposal complete</td>
</tr>
<tr>
<td>September 22</td>
<td>changed syntax to look like C</td>
</tr>
<tr>
<td>October 20</td>
<td>first version of scanner and parser complete</td>
</tr>
<tr>
<td>October 26</td>
<td>Language Reference Manual complete</td>
</tr>
<tr>
<td>November 11</td>
<td>scanner, parser, AST complete</td>
</tr>
<tr>
<td>November 13</td>
<td>“Hello World” program runs</td>
</tr>
<tr>
<td>November 15</td>
<td>automated testing script complete</td>
</tr>
<tr>
<td>Date</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 23</td>
<td>spawn, lock, barrier complete</td>
</tr>
<tr>
<td>December 7</td>
<td>added an env variable that gets computed during semantic analysis</td>
</tr>
<tr>
<td>December 8</td>
<td>added reference count in the env, for functions, local, and global variables</td>
</tr>
<tr>
<td>December 11</td>
<td>added ++ and -- operators</td>
</tr>
<tr>
<td>December 12</td>
<td>semantic analysis complete</td>
</tr>
<tr>
<td>December 13</td>
<td>started optimization for code generation by removing unused variables and func</td>
</tr>
<tr>
<td>December 14</td>
<td>optimization complete</td>
</tr>
<tr>
<td>December 16</td>
<td>pfor, code generation, debugging complete</td>
</tr>
<tr>
<td>December 17</td>
<td>final report complete</td>
</tr>
</tbody>
</table>
5. Architectural Design

5.1 Architecture

The architecture of our compiler is a standard one. The input code in SMPL syntax is submitted to the compiler which then generates an output C code. The C code can then be compiled using a C compiler and executed. The advantage of writing the program in SMPL instead of directly writing in C is that SMPL provides simple constructs to write the code. The complexity of creating thread functions and parallelizing the code by creating multiple threads is handled by our compiler and is transparent to the programmer. The flow diagram is as follows:
5.2 Error Recovery

Error reporting and recovery is an important part of the compiler. For parsing errors, the compiler simply reports that there is an error in parsing the SMPL code. In order to figure out the issue, code comments can be used to narrow down the line which led to the parsing issue. This is a limitation in the compiler since Ocaml doesn’t provide a convenient way to report line numbers and specific reasons for parsing errors.

The second type of errors is the semantic check failures. These include passing wrong argument types to functions, use of undefined variables/functions, using break/continue outside loop and so on. These errors are caught by the semantic checker. The semantic checker accurately reports the error with a sufficiently detailed reason and the function in which the error was seen. However, it doesn’t report the line number of the code in which the error occurred, since the compiler doesn’t maintain the line number information of the file. While parsing the code, the code is not necessarily parsed in the order in which it appears in the file.

Runtime errors are not caught by the compiler. This is generally true for any compiler as by definition runtime errors are ones that escaped the compiler and occur only when the code is executed. The SMPL compiler generates a C code but doesn’t validate it in any way. The code is stable enough to give a working C code in most cases, but could possibly spit out a code that doesn’t compile in GCC as a result of some bug in the SMPL compiler. These errors are not caught by the compiler and would be seen only when the C code is compiled/executed.

5.3 Optimizations

This is one of the cool features of our compiler. The original design of SMPL or the compiler functionality did not include this, but after understanding how a compiler works, we decided to implement perform one optimization.

The compiler eliminates dead code from the SMPL code and does not include it in the output C code. Dead code optimized by our compiler may include unused global variables, unused local variables, or unused functions. All of these are removed by the compiler while generating the output code.

Dead code is identified by the semantic checker while generating the symbol table. It parses the code and increases the reference count of variables and functions once a usage of any of them is seen. However, it also keeps track of usage of functions from main. So if a global variable is used only in an unused function, the reference count for the global will not be updated. Similarly, if function $a$ calls function $b$, but function $a$ is not called from any path from main function, then both function $a$ and
function $b$ are treated as dead code.

The implementation of reference count is done using a two stack method. Two lists (stacks) one of visited functions and another one of unvisited functions is maintained. The parsing is done starting from main function which is first put in the unvisited function stack. As function calls are seen in main, the functions are added to the unvisited stack. The functions are popped from the top of unvisited stack and then pushed to the visited stack and after that their body is processed. If the function is found in the visited stack and a call for it is seen, the reference count is not updated. This takes care of both direct and indirect recursive calls. Hence if function $a$ calls itself but no other function calls function $a$, it is still a dead function and will not get executed. This is handled by our algorithm and the function is removed from the output C code.

Once all the reference counts are generated, the C code generation logic checks for the reference count and if it is 0, does not output the function-variable in the resultant C code.

However, there is one downside of this optimization. The semantic checker ignores semantic analysis of the dead code in most cases. However, since the code is anyways going to be removed from the output C code, this should be an acceptable limitation of the compiler.

Example SMPL code (with dead code):

```
1 int unused1 = 0;
2 char unused2 = 'c';
3 float used1 = 2.0;
4 void main()
5 {
6   int test = 0;
7   used1 = 4.0;
8   printf("%f \n", used1);
9   func1_used();
10 }
11
12 func1_unused()
13 {
14   unused2 = 'd';
15   func2_unused(unused2);
16 }
```
func2_unused(char var)
{
    printf("%c", var);
    /* Indirect recursion */
    func1_unused();
}

func1_used()
{
    int unused_local = 100;
    printf("This function had an unused variable\n");
}

Output C code (dead code eliminated):

/* Code Generated from SMPL */
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

float used1 = 2.;
void func1_used();

void main();

void func1_used(){
    printf("This function had an unused variable\n");
}

void main(){
    used1=4.;
    printf("%f \n",used1);
    func1_used();
}
6. Test Plan

6.1 Testing Phases

6.1.1 Unit Testing

Unit testing was done at very frequent intervals, where each unit developed was tested rigorously using multiple cases. The scanner, parse and the ast were tested in phase 1 and in the later phase the semantic checker and the code generator were tested.

6.1.2 Integration Testing

In this phase, the various modules were put together and tested as a whole. Initially the syntax of the code was tested using multiple test cases and it was also ensured that only the correct syntax is accepted by running multiple fail test cases. Later on, the semantic checker was integrated and the semantics of the code were checked to be accurate without violating any of the rules of the language. In the final phase the code generator was added to the system and was tested to generate the corresponding C code.

6.1.3 System Testing

In order to ensure the proper working of the system, we began with a simple ‘Hello-World’ program and went on to test various constructs, function and also the parallel aspects of the language using a variety of test cases and also cross-checking against the expected output.

6.2 Automation

A shell script was written in order to automate the test cases at each level, syntax, semantic, code generation and accurate execution.

6.3 Test Suites

Tests were classified as Syntax, Semantic and Execution tests and were run at different phases during the construction of the compiler. The following aspects of the program were tested in order:

As a method of syntax verification, accurate parsing was ensured by running various pass and fail test cases, in semantic checking the semantic checker was verified to accept syntax that complies with various rules such of the language. In the case of code generation, the generated code was run on the C compiler and the output verified against the expected results.
7. Lessons Learned

7.1 Ajay

Regular team meetings helped us plan and develop the language with a clear outlook. Defining the syntax is a crucial part and should be done realistically without being too fancy. Keeping the syntax simple helped us focus on the core functionality of the compiler and we were able to deliver the promised model on time. Integrating and testing the code from time to time is important and is much effective as compared to bulk testing.

7.2 Andrei

OCaml is an extremely powerful language. Extremely annoying at first, yet powerful. That quote from the first class about the student “complaining” that “never have I written so little that does so much” is entirely valid. I can imagine how implementing a language in Java, Python, or C can be a tremendous pain.

Furthermore, it was interesting to see how we progressed as a team from “let’s use this syntax so that it is easier to parse” to understanding that what we might perceive difficult is completely trivial for the computer to do, as far as the language has a grammar that can be parsed.

Lessons learned? Start early. Start early. Start early. That includes programming in OCaml and writing your language. Also, do not attempt to implement 10,000 features because it is a) infeasible and b) out of the scope of this class.

7.3 Devashi

The project was the most interesting part of this course. I learnt more about how a compiler works and what goes on behind it by implementing one myself. Ocaml can be tedious and yet very convenient to code in. It is tedious because it takes time to catch up with the programming style of a functional language after coming from a C programmer background. Documents are sparse but the course slides helped. In my opinion we took a lot of time in identifying how to start writing the compiler. The assignments helped build the understanding but the project framework and how to write a Hello World program should be understood early into the course. This will help in delivering more functionality in the project. Working in a team and delivering the project in the given timeframe was a good learning experience.


8. References

We have modeled our Language Reference Manual and final report after the following four documents:


Appendix

Makefile

```bash
# Author: Devashi Tandon
TARFILES = Makefile scanner.mll parser.mly ast.mli printast.ml semantic_checker.ml compile.ml smpl.ml

OBJS = ast.cmo parser.cmo scanner.cmo printast.cmo semantic_checker.cmo compile.cmo smplc.cmo

smplc : $(OBJS)
    ocamlc str.cma -o smplc $(OBJS)

scanner.ml : scanner.mll
    ocamlle scanner.mll

parser.ml parser.mli : parser.mly
    ocamlyacc parser.mly

%.cmi : %.ml
    ocamlc -c $<

%.cmo : %.ml
    ocamlc -c $<

smplcompiler.tar.gz : $(TARFILES)
    cd .. && tar zcf smplcompiler/smplcompiler.tar.gz $(TARFILES:%=smplcompiler/%)

.PHONY : clean
clean :
    rm -f smplc parser.ml parser.mli scanner.ml *.cmo *.cmi
```


type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater | Geq | And | Or

type data_type =
  IntType of string
| FloatType of string
| BoolType of string
| CharType of string
| StrType of string
| VoidType of string

type literal =
  Integer of int
| Float of float
Boolean of bool
| Char of char
| String of string

type expr =
  Id of string
| Literal of literal
| Binop of expr * op * expr
| Not of expr
| Pp of string
| Mm of string
| Assign of expr * expr
| Call of string * expr list
| Paren of expr
| Noexpr

type stmt =
  Block of stmt list
| Expr of expr
| Return of expr
| Break of expr
| Continue of expr
| Declare of data_type
| DeclareAssign of data_type * expr
| If of expr * stmt * stmt
| For of expr * expr * expr * stmt
| While of expr * stmt
| Pfor of expr * expr * expr * expr * stmt
| Spawn of expr
| Lock of stmt
| Barrier of expr

type func_decl = {
  fname : data_type;
  formals : data_type list;
  body : stmt list;
}
type program = (data_type * literal) list * func_decl list

printast.ml

#Author: Devashi Tandon
open Ast

let string_of_data_type = function
    IntType(s) -> "Integer" ^ s
| FloatType(s) -> "Float" ^ s
| BoolType(s) -> "Boolean" ^ s
| CharType(s) -> "Char" ^ s
| StrType(s) -> "Str" ^ s
| VoidType(s) -> "Void" ^ s

let string_of_literal = function
    Integer(i) -> "Integer Lit: " ^ string_of_int i
| Float(f) -> "Float Lit: " ^ string_of_float f
| Boolean(b) -> "Boolean Lit: " ^ string_of_bool b
| Char(c) -> "Char Lit: " ^ Char.escaped c
| String(s) -> "String Lit: " ^ ""^ s ^ ""

let rec string_of_expr = function
    Literal(l) -> string_of_literal l
| Id(s) -> "ID:" ^ s
| Paren(s) -> "(" ^ string_of_expr s ^ ")"
| Binop(e1, o, e2) -> "BINOP:" ^ string_of_expr e1 ^ " " ^
    (match o with
    Add -> "PLUS"
| Sub -> "MINUS"
| Mult -> "TIMES"
| Div -> "DIV"
| Mod -> "MOD"
| Equal -> "EQUAL"
| Neq -> "NOTEQUAL"
| Greater -> "GT"
| Geq -> "GTE"
| Less -> "LT" |
| Leq -> "LTE" |
| And -> "AND" |
| Or -> "OR" | ^ " " | ^ string_of_expr e2 |
| Not(e1) -> "BINOP:" | ^ "NOT" | ^ string_of_expr e1 |
| Pp(id) -> "INCREMENT " | ^ id |
| Mm(id) -> "DECREMENT " | ^ id |
| Assign(id, idx) -> "ASSIGN- " | ^ |
| (match id with |
| Id(id) -> id |
| _ -> "ERROR") | ^ "to: [" | ^ string_of_expr idx | ^ "]" |
| Call(s1, al) -> "Call: " | ^ s1 | ^ ": (" | ^ String.concat | ";"
| (List.map (fun e -> string_of_expr e) al) | ^ ")"
| Noexpr -> "NoExpr"

let rec string_of_stmt = function

Block(stmts) ->
"{" | ^ String.concat | "" | (List.map string_of_stmt stmts) | ^ "}\n"

| Expr(e) -> "EXPR: [" | ^ string_of_expr e | ^ "]\n"
| Return(e) -> "RETURN " | ^ string_of_expr e | ^ ";\n"
| Break(e) -> "BREAK " | ^ string_of_expr e | ^ ";\n"
| Continue(e) -> "CONTINUE " | ^ string_of_expr e | ^ ";\n"
| Declare(d) -> "Declare " | ^ string_of_data_type d | ^ ";\n"
| DeclareAssign(d, e) -> "DeclareAssign " | ^ string_of_data_type d | ^ ": AND Assign: " | ^ string_of_expr e | ^ ";\n"

| If (p, t, f) -> |
| "IF " | ^ string_of_expr p |
| ^ " THEN DO " |
| ^ string_of_stmt t |
| ^ " ELSE DO " | ^ string_of_stmt f | ^ ";\n"
| For (s, e, se, b) -> |
| "DECLARE: " | ^ string_of_expr s |
| ^ " AND DO " | ^ string_of_stmt b |
| ^ " WHILE " | ^ string_of_expr e |
| ^ " PERFORMING " | ^ string_of_expr se | ^ ";\n"
| While (e, b) -> " WHILE " | ^ string_of_expr e | ^ ": DO " | ^ string_of_stmt b | ^ ";\n"
| Pfor(t, c, s, e, b) -> "PFOR: Threads: " | ^ string_of_expr t | ^ "; Variable: " |
| ^ string_of_expr c | ^ ": " | ^ string_of_expr s
let string_of_vdecl id = "long " ^ id ^ ";\n"

let string_of_fdecl fdecl =
"FUNCTION " ^ string_of_data_type fdecl.fname ^ "(" ^
String.concat "," (List.map string_of_data_type fdecl.formals) ^
")\n\n" ^
String.concat "" (List.map string_of_stmt fdecl.body) ^
")\n"

let string_of_globals globals =
"GLOBALS " ^ string_of_data_type (fst globals) ^ " = " ^ string_of_literal (snd globals)

let parse_program (vars, funcs) =
"Vars: \n" ^ String.concat ";\n" (List.map string_of_globals vars) ^ "\n" ^
"Funcs: \n" ^ String.concat "" (List.map string_of_fdecl funcs)

scanner.mll

# Primary author: Devashi Tandon
# Secondary author: Andrei Pancea
{ open Parser
  open Str
}

rule token = parse
[ " ", 't' , 'r' , "\n"] { token lexbuf }
| "/*" { comment lexbuf } (* Comments *)
| '(' { LPAREN }
| ')" { RPAREN }
| "null" | { NULL } |
| "true" as lxm | { BOOL_LIT(bool_of_string lxm) } |
| "false" as lxm | { BOOL_LIT(bool_of_string lxm) } |
| [0-9] as lit | { INTEGER_LIT(int_of_string lit) } |
| [0-9][.]?[0-9]+? as lit | { FLOAT_LIT(float_of_string lit) } |
| [0-9]+[e][][0-9]+? as lit | { FLOAT_LIT(float_of_string lit) } |
| ["\]^ as lxm | { STRING_LIT(lxm) } |
| [a-zA-Z_0-9] as lxm | { ID(lxm) } |
| eof | { EOF } |
| _ as char | { raise (Failure("illegal character " ^ Char.escaped char)) } |

and comment = parse
  "*/" { token lexbuf } |
  _ { comment lexbuf }

### parser.mly

```
#/ Authors: Devashi Tandon, Andrei Papancea
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS PP MM TIMES DIVIDE MODULUS ASSIGN
%token AND OR NOT
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE INT
%token INT FLOAT BOOLEAN CHAR STRING
%token SPAWN PFOR LOCK BARRIER
%token BREAK CONTINUE
%token VOID NULL
%token <int> INTEGER_LIT
%token <float> FLOAT_LIT
%token <bool> BOOL_LIT
%token <char> CHAR_LIT
%token <string> STRING_LIT
%token <string> ID
```
%token GLOBAL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULUS
%left NOT

%start program
%type < Ast.program> program

%%

program: /* nothing */ { [], [] } |
| program fdecl { fst $1, (snd $1)$2 :: snd $1}) /* devashi: What the heck is this? */ |
| program vdecl ASSIGN type_literal SEMI { ( (snd $4) :: fst $1), snd $1 }) /* devashi: What the heck is this? */ |

fdecl:
   ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
   {{ fname = VoidType($1); formals = $3; body = List.rev $6 }}
| VOID ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
   {{ fname = VoidType($2); formals = $4; body = List.rev $7 }}
| vdecl LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
   {{ fname = $1; formals = $3; body = List.rev $6 }}

formals_opt:
   /* nothing */ { [] } |
| formal_list { List.rev $1 }

formal_list:

vdecl { [$1] }
| formal_list COMMA vdecl { $3 :: $1 } |

vdecl:
  INT ID { IntType($2) }
| FLOAT ID { FloatType($2) }
| BOOLEAN ID { BoolType($2) }
| CHAR ID { CharType($2) }
| STRING ID { StrType($2) }

stmt_list:
  /* nothing */ { [] }
| stmt_list stmt { $2 :: $1 }

stmt:
  expr SEMI { Expr($1) }
| vdecl SEMI { Declare($1) }
| vdecl ASSIGN expr SEMI { DeclareAssign($1, $3) }
| RETURN expr_opt SEMI { Return($2) }
| BREAK empty_opt SEMI { Break($2) }
| CONTINUE empty_opt SEMI { Continue($2) }
| LBRACE stmt_list RBRACE { Block(List.rev $2) }
| IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
| IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
| FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt { For($3, $5, $7, $9) }
| WHILE LPAREN expr RPAREN stmt { While($3, $5) }
| PFOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt { Pfor($3, $5, $7, $9, $11) }
| SPAWN call_opt SEMI { Spawn($2) }
| LOCK stmt { Lock($2) }
| BARRIER empty_opt SEMI { Barrier($2) }

call_opt:
  ID LPAREN actuals_opt RPAREN { Call($1, $3) }

empty_opt:
  /* nothing */ { Noexpr }
expr_opt:
/* nothing */ { Noexpr }
| expr { $1 }

type_literal:
INTEGER_LIT { Integer($1) }
| FLOAT_LIT { Float($1) }
| BOOL_LIT { Boolean($1) }
| CHAR_LIT { Char($1) }
| STRING_LIT { String($1) }

expr:
ID { Id($1) }
| ID PP { Pp($1) }
| ID MM { Mm($1) }
| type_literal { Literal($1) }
| expr PLUS expr { Binop($1, Add, $3) }
| expr MINUS expr { Binop($1, Sub, $3) }
| expr TIMES expr { Binop($1, Mult, $3) }
| expr DIVIDE expr { Binop($1, Div, $3) }
| expr MODULUS expr { Binop($1, Mod, $3) }
| expr AND expr { Binop($1, And, $3) }
| expr OR expr { Binop($1, Or, $3) }
| NOT expr { Not($2) }
| 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) }
| ID ASSIGN expr { Assign(Id($1), $3) }
| LPAREN expr RPAREN { Paren($2) }
| ID LPAREN actuals_opt RPAREN { Call($1, $3) }

actuals_opt:
/* nothing */ { [] }
| actuals_list { List.rev $1 }
actuals_list:
  expr { [$1] }
| actuals_list COMMA expr { $3 :: $1 }

semantic_checker.ml

# Primary author: Devashi Tandon
open Ast
open Str

module VarMap = Map.Make(struct
  type t = string * string
  let compare x y = Pervasives.compare x y
end)
module StringMap = Map.Make(String)

(* Symbol table: Information about all the symbols *)
type env = {
  globals_map : (string * int) StringMap.t; (* global variable name-> type, ref_count *)
  functions_map : (string * int) StringMap.t; (* function name -> return type, ref_count *)
  formals_map : (string) VarMap.t; (* function:argument name -> types *)
  locals_map : (string * int) VarMap.t; (*function name:local variable name -> type, ref_count *)
  unvisited_funcs : string list; (* maintains function call stack for getting ref_count *)
  visited_funcs : string list; (* maintains list of visited functions *)
  formals_list : (string * data_type) list; (* needed as map stores in alphabetical order, not in order of
 insertion*)
  visited_all : bool;
  valid_syntax : bool
}

let print_map = false
let print_process_funcs = false

let validate_program (globalvars, funcs) =
  let print_string str_to_print =
    print_string ("/* " ^ str_to_print ^ " */\n")
in let id_of_data_type = function
  IntType(s) -> s
  FloatType(s) -> s
  BoolType(s) -> s
  CharType(s) -> s
  StrType(s) -> s
  VoidType(s) -> s

in let get_data_type = function
  IntType(s) -> "integer"
  FloatType(s) -> "float"
  BoolType(s) -> "boolean"
  CharType(s) -> "char"
  StrType(s) -> "string"
  VoidType(s) -> "void"

in let string_of_data_type = function
  IntType(s) -> "integer " ^ s
  FloatType(s) -> "float " ^ s
  BoolType(s) -> "boolean " ^ s
  CharType(s) -> "char " ^ s
  StrType(s) -> "string " ^ s
  VoidType(s) -> "void " ^ s

in let string_of_literal = function
  Integer(i) -> "integer value: " ^ string_of_int i
  Float(f) -> "float value: " ^ string_of_float f
  Boolean(b) -> "boolean value: " ^ string_of_bool b
  Char(c) -> "char value: " ^ Char.escaped c
  String(s) -> "string value: \"" ^ s ^ "\"

in let get_literal_type = function
  Integer(s) -> "integer"
  Float(s) -> "float"
  Boolean(s) -> "boolean"
  Char(s) -> "char"
  String(s) -> "string"
in let global_map =
  List.fold_left (fun m global ->
    let dt_global = id_of_data_type(fst global) in
    let found = StringMap.mem dt_global m in
    if(found = false) then
      (StringMap.add dt_global (get_data_type(fst global), 0) m)
    else
      raise (Failure ("global variable " ^ dt_global ^
                  " is redefined"))
  ) StringMap.empty globalvars
in let function_map =
  List.fold_left (fun m func ->
    let id_func = id_of_data_type(func.fname) in
    let found = StringMap.mem id_func m in
    if(found = true) then
      raise (Failure ("Function " ^ id_func ^ " is redefined"))
    else (let found = StringMap.mem id_func global_map in
       if(found = false) then
         (StringMap.add id_func (get_data_type(func.fname), 0) m)
       else
         raise (Failure ("Function " ^ id_func ^
                         " conflicts with a global variable"))
    )
  ) StringMap.empty funcs
in let func_decl_map =
  List.fold_left (fun m func -> StringMap.add
    (id_of_data_type(func.fname)) func m) StringMap.empty funcs
in let global_env = {
  globals_map = global_map;
  functions_map = function_map;
  formals_map = VarMap.empty;
  locals_map = VarMap.empty;
  unvisited_funcs = ["main"];
  visited_functs = [];
  formals_list = [];
  visited_all = false;
valid_syntax = false;

in let reserved_keyword keyword =
  if((keyword = "printf")||(keyword = "threads")||
  (keyword = "pthread_create")||(keyword = "pthread_join")
  ||(keyword = "pthread_mutex_t")||(keyword = "pthread_mutex_lock")
  ||(keyword = "pthread_mutex_unlock")||(keyword = "NULL")) then
    true
  else (let regex = Str.regexp ("\(thread_.*\)\|\(pfor_uppers_.*\)\|\(pfor_lowers_.*\)\|\(pfor_args_.*\)\|\(pfor_limit_.*\)\|\(pfor_i_.*\)\|\(pfor_threads_.*\)\|\(pfor_init_.*\)\|\(num_threads_.*\)\|\(lock_.*\)")
in if(Str.string_match regex keyword 0) then
    true
  else false)

  in let supported_typecasting type_left type_right =
    if(type_left = "float" && (type_right =
      "integer"||type_right = "boolean"||type_right = "char")) then
      true
    else if(type_left = "integer" && (type_right =
      "boolean"||type_right = "char")) then
      true
    else if(type_left = "char" && type_right = "integer") then
      true
    else
      false
  (* For debugging purposes *)
in let print_global_map env =
  StringMap.iter (fun dt_key (vartype, ref) -> print_string ("Global var: id: " ^
  dt_key ^ " type=" ^ vartype ^ " ref=" ^ (string_of_int ref)))
  env.globals_map
in let print_function_map env =
  StringMap.iter (fun dt_key (vartype, ref) -> print_string ("Function: id: " ^
  dt_key ^ " type=" ^ vartype ^ " ref=" ^ (string_of_int ref)))
in let print_locals_map env =
  VarMap.iter (fun (dt_func, dt_var) (var_type, var_ref) ->
    print_string ("Local Variable Func: " ^ dt_func ^ " var: " ^ dt_var ^
      " type: " ^ var_type ^ " ref= " ^ string_of_int var_ref)
  ) env.locals_map

in let print_formals_map env =
  VarMap.iter (fun (dt_func, dt_arg) (arg_type) ->
    print_string ("Map Arguments Func: " ^ dt_func ^ " arg: " ^ dt_arg ^ " type: " ^
      arg_type)
  ) env.formals_map

in let print_formals_list env =
  List.iter (fun (fname_id, arg) ->
    print_string ("List Arguments Func: " ^ fname_id ^ arg_type) ^ (id_of_data_type arg) ^ " type: " ^ (get_data_type arg))
  ) env.formals_list

in let print_unvisited_funcs env =
  List.iter (fun f -> print_string ("unvisited: " ^ f)) env.unvisited_funcs

in let print_visited_funcs env =
  List.iter (fun f -> print_string ("visited: " ^ f)) env.visited_funcs

in let validate_global variable =
  if (get_data_type (fst variable) <> get_literal_type (snd variable)) then
    raise (Failure ("Global variable " ^ string_of_data_type (fst variable) ^ " is assigned a " ^ string_of_literal (snd variable)))

in let add_local_var_to_map fname_id var env =
  let id_var = id_of_data_type var in
  let found = VarMap.mem (fname_id, id_var) env.locals_map in
  if(found = false) then
    { env with locals_map =
      VarMap.add (fname_id, id_var) ((get_data_type var), 0)
      env.locals_map }
  else
    raise (Failure ("Function " ^ fname_id ^
        " contains more than one local variable with the name " ^ id_var))
in let process_formals fname_id arg env =
    let id_arg = id_of_data_type arg in
    if(reserved_keyword id_arg) then
        raise (Failure ("Function:" ^ fname_id ^ ": ": ^ id_arg ^ " is a reserved keyword. It cannot be ": ^ "used as an argument name"))
    else
        let found = VarMap.mem (fname_id, id_arg) env.formals_map in
        if(found = false) then
            let new_env = { env with formals_list =
                (fname_id, arg) :: env.formals_list }
        in
            { new_env with formals_map =
                VarMap.add (fname_id, id_arg) (get_data_type arg)
                new_env.formals_map }
        )
        else
            raise (Failure ("Function " ^ fname_id ^ ": ": contains more than one argument with the name " ^ id_arg))
    )

in let get_func_formals_list fname_id env =
    List.fold_left (fun arg_list (dt_func, arg) ->
        if(dt_func = fname_id) then
            arg :: arg_list
        else
            arg_list
    ) [] env.formals_list

in let rec process_expr fname_id env = function
    Literal(l) -> env
| Paren(s) -> process_expr fname_id env s
| Binop(e1, _, e2) ->
    let new_env = process_expr fname_id env e1 in
    process_expr fname_id new_env e2
| Not(e1) ->
process_expr fname_id env e1

<table>
<thead>
<tr>
<th>Assign(id, idx) -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign(id, idx) -&gt;</td>
</tr>
<tr>
<td>let new_env = process_expr fname_id env id in</td>
</tr>
<tr>
<td>process_expr fname_id new_env idx</td>
</tr>
</tbody>
</table>

| Id(s)| Pp(s)| Mm(s) -> |
| Id(s)| Pp(s)| Mm(s) -> |
|-------------------|
| Id(s)| Pp(s)| Mm(s) -> |
|-------------------|
| let (var_type, ref_count) = try VarMap.find (fname_id, s) env.locals_map |
| with Not_found -> ("", 0) in |
| if(var_type = "") then ( |
| let var_type = try VarMap.find (fname_id, s) env.formals_map |
| with Not_found -> "" in |
| if(var_type = "") then ( |
| let (var_type, ref_count) = try StringMap.find s env.globals_map |
| with Not_found -> ("", 0) in |
| if(var_type = "") then ( |
| raise (Failure ("Variable " ^ s ^ " in function " ^ |
| fname_id ^ " is not defined")) |
| ) else ( |
| { env with globals_map = |
| StringMap.add s (var_type, ref_count+1) env.globals_map |
| } |
| ) |
| else ( |
| env |
| ) |
| ) else ( |
| { env with locals_map = |
| VarMap.add (fname_id, s) (var_type, ref_count+1) |
| env.locals_map } |
| ) |

<table>
<thead>
<tr>
<th>Call(s1, al) -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call(s1, al) -&gt;</td>
</tr>
<tr>
<td>let (var_type, ref_count) = try StringMap.find s1 env.functions_map</td>
</tr>
<tr>
<td>with Not_found -&gt; (&quot;&quot;, 0) in</td>
</tr>
<tr>
<td>if(var_type = &quot;&quot; &amp;&amp; (s1 &lt;&gt; &quot;printf&quot;)) then</td>
</tr>
<tr>
<td>raise (Failure (&quot;Function &quot; ^ s1 ^ &quot; is not defined&quot;))</td>
</tr>
<tr>
<td>else (</td>
</tr>
<tr>
<td>if(List.mem s1 env.visited_funcs = false) then (</td>
</tr>
<tr>
<td>let env = { env with unvisited_funcs =</td>
</tr>
<tr>
<td>if((List.mem s1 env.unvisited_funcs = false) }</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>else (</td>
</tr>
<tr>
<td>if(List.mem s1 env.unvisited_funcs = false) }</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>else (</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>)</td>
</tr>
</tbody>
</table>
&& s1 <> "printf") then
    s1::env.unvisited_funcs
else
    env.unvisited_funcs
} in
let mod_env = { env with functions_map =
    StringMap.add s1 (var_type, ref_count+1) env.functions_map
} in List.fold_left (fun old_env arg ->
    process_expr fname_id old_env arg) mod_env al
else (List.fold_left (fun old_env arg ->
    process_expr fname_id old_env arg) env al)

| Noexpr -> env

in let update_env_map curr_env functions =
    List.fold_left (fun env argument ->
    process_formals (id_of_data_type functions.fname) argument env)
curr_env functions.formals

in let rec process_func_body fname env = function
    Block(stmts) ->
    List.fold_left (fun env_temp line ->
    process_func_body fname env_temp line) env stmts
| Expr(exp) -> process_expr (id_of_data_type fname) env exp
| Return(exp) -> process_expr (id_of_data_type fname) env exp
| Break(_) -> env
| Continue(_) -> env
| Declare(decl) ->
    add_local_var_to_map (id_of_data_type fname) decl env
| DeclareAssign(decl, exp) ->
    let new_env = add_local_var_to_map (id_of_data_type fname) decl env
    in process_expr (id_of_data_type fname) new_env exp
| If (cond, then_clause, else_clause) ->
    let first_env = process_expr (id_of_data_type fname) env cond
    in let second_env = process_func_body fname first_env then_clause
in process_func_body fname second_env else_clause

| For (init, cond, inc, body) -> |
| let first_env = process_expr (id_of_data_type fname) env init |
| in let second_env = process_expr (id_of_data_type fname) first_env cond |
| in let third_env = process_expr (id_of_data_type fname) second_env inc |
| in process_func_body fname third_env body |

| While (exp, body) -> |
| let first_env = process_expr (id_of_data_type fname) env exp |
| in process_func_body fname first_env body |

| Pfor (threads, counter, init, cond, body) -> |
| let first_env = process_expr (id_of_data_type fname) env threads |
| in let second_env = process_expr (id_of_data_type fname) first_env counter |
| in let third_env = process_expr (id_of_data_type fname) second_env init |
| in let fourth_env = process_expr (id_of_data_type fname) third_env cond |
| in process_func_body fname fourth_env body |

| Spawn (exp) -> process_expr (id_of_data_type fname) env exp |
| Lock (stmt) -> process_func_body fname env stmt |
| Barrier (exp) -> env |

in let process_func env =

let fname_id =
match env.unvisited_funcs with

[] -> ""
| hd::tl -> hd

in if (fname_id = "") then

{ env with visited_all = true }

else if (fname_id <> "printf") then ( |
let env = if (fname_id = "main") then

let (var_type, ref_count) = try StringMap.find fname_id
env.functions_map with Not_found -> raise (Failure ("function " ^
fname_id ^ " not defined")) in

{ env with functions_map = |
StringMap.add fname_id (var_type, ref_count+1) env.functions_map }
else

env

in let fdecl = try StringMap.find fname_id func_decl_map |
with Not_found -> |
raise (Failure ("function " ^ fname_id ^ " not defined")) |
in let p_env = { env with unvisited_funcs = List.tl env.unvisited_funcs }
in let p_env = { p_env with visited_funcs = fname_id :: p_env.visited_funcs } in
List.fold_left (fun env_temp line ->
  process_func_body fdecl.fname env_temp line) p_env fdecl.body
) else (evn)

in let rec process_code env =
  let new_env = process_func env in
  if(new_env.visited_all = false) then
    let _ = if(print_process_funcs) then (print_string ("process_code");
      print_unvisited_funcs new_env;
      print_visited_funcs new_env
    )
    in process_code new_env
  else
    new_env
  in let rec verify_expr fname_id env is_in_loop = function
    Literal(l) -> get_literal_type(l)
  | Paren(s) -> verify_expr fname_id env is_in_loop s
  | Binop(e1, o, e2) ->
    let type_e1 = verify_expr fname_id env is_in_loop e1
    in let type_e2 = verify_expr fname_id env is_in_loop e2
    in (match o with
      Add|Sub|Mult|Div|Mod ->
        if((type_e1 = "string") || (type_e2 = "string")) then
          raise (Failure ("Invalid operand to string in function " ^
            fname_id))
        else if((type_e1 = "void") || (type_e2 = "void")) then
          raise (Failure ("Invalid operand to void in function " ^
            fname_id))
        else if((type_e1 = "float") || (type_e2 = "float")) then
          "float"
        else if((type_e1 = "integer") || (type_e2 = "integer")) then
          "integer"
else if((type_e1 = "char") || (type_e2 = "char")) then
  "char"
else if((type_e1 = "boolean") || (type_e2 = "boolean")) then
  "boolean"
else
  raise (Failure ("Invalid data type in function " ^ fname_id))
| Equal|Neq|Greater|Geq|Less|Leq ->
  if((type_e1 = "string") && (type_e2 <> "string")) then
    raise (Failure ("Invalid operand to string in function " ^ fname_id))
  else if((type_e2 = "string") && (type_e1 <> "string")) then
    raise (Failure ("Invalid operand to string in function " ^ fname_id))
  else if((type_e1 = "void") || (type_e2 = "void")) then
    raise (Failure ("Invalid operand to void in function " ^ fname_id))
  else
    "boolean"
| And|Or -> "boolean"
)
| Not(e) -> "boolean"
| Assign(id, idx) ->
  let type_id = verify_expr fname_id env is_in_loop id
  in let type_right = verify_expr fname_id env is_in_loop idx
  in let id_name = (match id with
    Id(s) -> s
  | _ -> raise (Failure ("Unexpected error during Assign in function " ^ fname_id)))
  )
in if(type_id <> type_right) then (in
  if((supported_typecasting type_id type_right) = true) then
    type_id
  else
    raise (Failure ("Type mismatch:" ^ type_id ^ " variable " ^ id_name ^ " is assigned " ^ type_right ^ " in function " ^ fname_id))
  else
type_id
| Id(s) ->
| let (var_type, _) = try VarMap.find (fname_id, s) env.locals_map
| with Not_found -> ("", 0) in
| if(var_type = "") then (  
| let var_type = try VarMap.find (fname_id, s) env.formals_map
| with Not_found -> "" in
| if(var_type = "") then (    
| let (var_type, ref_count) = try StringMap.find s env.globals_map
| with Not_found -> ("", 0) in
| if(var_type = "") then (        
| raise (Failure ("Variable " ^ s ^ " in function " ^ fname_id ^ " is not defined")))
| ) else
| var_type
| ) else
| var_type
| ) else
| var_type
| Pp(s)\Mm(s) ->
| let (var_type, _) = try VarMap.find (fname_id, s) env.locals_map
| with Not_found -> ("", 0) in
| if(var_type = "") then (  
| let var_type = try VarMap.find (fname_id, s) env.formals_map
| with Not_found -> "" in
| if(var_type = "") then (    
| let (var_type, ref_count) = try StringMap.find s env.globals_map
| with Not_found -> ("", 0) in
| if(var_type = "") then (        
| raise (Failure ("Variable " ^ s ^ " in function " ^ fname_id ^ " is not defined")))
| ) else (    
| if(var_type <> "integer") then      
| raise (Failure ("Function: " ^ fname_id ^ ": Variable " ^ s ^ " is of type " ^ var_type ^ ". However ++ and -- are supported only for integers")))
| else
| var_type
if(var_type <> "integer") then
  raise (Failure ("Function: "^fname_id^": Variable "^s^" is of type "^var_type^"
    " is supported only for integers")
else
  var_type
)
) else ("
if(var_type <> "integer") then
  raise (Failure ("Function: "^fname_id^": Variable "^s^" is of type "^var_type^"
    " is supported only for integers")
else
  var_type
)
) else ("
if(var_type <> "integer") then
  raise (Failure ("Function: "^fname_id^": Variable "^s^" is of type "^var_type^"
    " is supported only for integers")
else
  var_type
)
) else ("
if(var_type <> "integer") then
  raise (Failure ("Function: "^fname_id^": Variable "^s^" is of type "^var_type^"
    " is supported only for integers")
else
  var_type
)
)

| Call(s1, al) ->
  let (var_type, _) = try StringMap.find s1 env.functions_map
  with Not_found -> ("", 0) in
  if(var_type = "" && (s1 <> "printf")) then
    raise (Failure ("Function "^s1^" is not defined")
  else ("
  if(s1 = "printf") then
    var_type
  else ("
    let arg_list = get_func_formals_list s1 env
    in let _ = try (List.iter2 (fun arg value ->
      let a_type = verify_expr fname_id env is_in_loop value in
      if(get_data_type(arg) <> a_type) then ("
        raise(Failure ("Function "^fname_id^": call to:"^s1^"
          " arg "^(id_of_data_type arg)^" is of type "^(get_data_type arg)^" but it is called with type "^a_type^")
      )arg_list al)
    with Invalid_argument(s) -> ("
      raise(Failure (fname_id^": call to:"^s1^"
        " number of arguments don't match")
      )))
      in var_type
in let rec verify_func_body fname env data_type is_in_loop = function

  Block(stmts) ->
    List.fold_left (fun new_data_type line ->
      verify_func_body fname env new_data_type is_in_loop line) data_type stmts

  | Expr(exp) -> verify_expr (id_of_data_type fname) env is_in_loop exp

  | Return(exp) ->
    let return_type = verify_expr (id_of_data_type fname) env
    is_in_loop exp
    in let func_return_type = get_data_type (fname)
    in if((func_return_type = return_type) || (func_return_type = "void" && return_type = "")
      then ""
      else(
        if(return_type = ") then
          raise (Failure (" The return type of function " ^
                              id_of_data_type fname ^ " is " ^ func_return_type ^
                              " but it returns " ^ "void" ^ " type"))
        else
          raise (Failure (" The return type of function " ^
                             id_of_data_type fname ^ " is " ^ func_return_type ^
                             " but it returns " ^ return_type ^ " type")))

  | Break(_) ->
    if(is_in_loop) then ""
    else
      raise (Failure ("Function:" ^ id_of_data_type fname ^
                      " break statement not within a loop"))

  | Continue(_) ->
    if(is_in_loop) then ""
    else
      raise (Failure ("Function:" ^ id_of_data_type fname ^
"continue statement not within a loop")

| Declare(decl) ->
| (*TODO: Throw a warning if global or arg is defined with same name *)
|   let func_name = id_of_data_type decl in
|   if(reserved_keyword func_name) then
|     raise (Failure ("Function:" ^id_of_data_type(fname)^": " ^ func_name ^ " is a reserved keyword. It cannot be " ^ "used as a variable name"))
|   else
|     ""

| DeclareAssign(decl, exp) ->
| (*TODO: Throw a warning if global or arg is defined with same name *)
|   let func_name = id_of_data_type decl in
|   if(reserved_keyword func_name) then
|     raise (Failure ("Function:" ^id_of_data_type(fname)^": " ^ func_name ^ " is a reserved keyword. It cannot be " ^ "used as a variable name"))
|   else(
|     let assign_type = verify_expr (id_of_data_type fname) env is_in_loop exp in
|     if(get_data_type (decl) = assign_type) then
|       ""
|     else
|       if(supported_typecasting (get_data_type (decl)) assign_type)) then
|         ""
|       else
|         raise (Failure ("Local variable " ^ string_of_data_type (decl) ^ " is assigned a " ^ assign_type ^ " in function " ^ id_of_data_type (fname)))
|   )

| If (cond, then_clause, else_clause) ->
|   let _ = verify_expr (id_of_data_type fname) env is_in_loop cond in
|   let _ = verify_func_body fname env "" is_in_loop then_clause in
|   let _ = verify_func_body fname env "" is_in_loop else_clause in

| For (init, cond, inc, body) ->
let _ = verify_expr (id_of_data_type fname) env is_in_loop init
in let _ = verify_expr (id_of_data_type fname) env is_in_loop cond
in let _ = verify_expr (id_of_data_type fname) env is_in_loop inc
in let _ = verify_func_body fname env "" true body
in ""
| While (exp, body) ->
  let _ = verify_expr (id_of_data_type fname) env is_in_loop exp
  in let _ = verify_func_body fname env "" true body
  in ""
| Pfor(threads, counter, init, cond, body) ->
  let _ = match counter with
    Id(s) ->
      let counter_type = verify_expr (id_of_data_type fname) env
                   is_in_loop counter in
      if(counter_type <> "integer") then
        raise (Failure ("Function: "^(id_of_data_type fname)^
                       "counter in pfor should be an integer variable ONLY.
                       "^ " Here it is "^(counter_type)))
    _ -> raise(Failure ("Function: "^(id_of_data_type fname)^
                          "counter in pfor should be an integer variable name ONLY"))
  in let data_type =
      verify_expr (id_of_data_type fname) env is_in_loop threads
  in if(data_type <> "integer") then
    raise (Failure ("Function:"^(id_of_data_type fname)^
                    " First argument to pfor should be of type integer"))
  else (let data_type = verify_expr (id_of_data_type fname) env
           is_in_loop init
  in if(data_type <> "integer") then
    raise (Failure ("Function:"^(id_of_data_type fname)^
                    " Second argument to pfor should be of type integer"))
  else (let data_type = verify_expr (id_of_data_type fname) env
           is_in_loop cond
  in if(data_type <> "integer") then
    raise (Failure ("Function:"^(id_of_data_type fname)^
                    " Third argument to pfor should be of type integer"))
  else (}
let _ = verify_func_body fname env "" true body
  in ""
)
)
)

<table>
<thead>
<tr>
<th>Spawn(exp) -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>let _ = match exp with</td>
</tr>
<tr>
<td>Call(s1, al) -&gt; if(s1 = &quot;printf&quot;) then</td>
</tr>
<tr>
<td>raise (Failure (&quot;Function:&quot;^(id_of_data_type fname)^ &quot; printf is not allowed in spawn&quot;)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>in let _ = verify_expr (id_of_data_type fname) env is_in_loop exp</td>
</tr>
</tbody>
</table>
| in ""

<table>
<thead>
<tr>
<th>Lock(stmt) -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>let _ = verify_func_body fname env &quot;&quot; is_in_loop stmt</td>
</tr>
</tbody>
</table>
| in ""

| Barrier(exp) -> ""

in let verify_function env str_dt fdecl = |
| List.fold_left (fun str_data_type line -> |
| verify_func_body fdecl.fname env str_data_type false line |
| ) str_dt fdecl.body |

in let final_env curr_env valid = |
| { curr_env with valid_syntax = valid } |

in let _ = List.iter validate_global globalvars |
in let first_env = List.fold_left (fun env func_list -> |
| update_env_map env func_list) global_env funcs |
in let second_env = process_code first_env |
in let _ = List.fold_left (fun str_dt func -> |
| let func_name = id_of_data_type func.fname |
| in if(reserved_keyword func_name) then |
| raise (Failure ("Function:"^func_name^": "^func_name^ " is a reserved keyword. It cannot be used as a"^ " function name")) |
else
    verify_function second_env str_dt func
) "" funcs
in if (print_map) then (  
    let _ = print_global_map second_env in
    let _ = print_function_map second_env in
    let _ = print_locals_map second_env in
    let _ = print_formals_map second_env in
    let _ = print_formals_list second_env
    in final_env second_env true)
else (  
    final_env second_env true)

compile.ml

# Primary author: Andrei Papancea

open Ast
open String

module VarMap = Map.Make(struct
    type t = string * string
    let compare x y = Pervasives.compare x y
end)

let includes =  
"#include <stdio.h>\n" ^
"#include <stdbool.h>\n" ^
"#include <stdlib.h>\n" ^
"#include <string.h>\n" ^
"#include <pthread.h>\n\n"

let rec string_of_data_type = function
    IntType(s) -> "int " ^ s
| FloatType(s) -> "float " ^ s
| BoolType(s) -> "bool " ^ s
| CharType(s) -> "char " ^ s
| StrType(s) -> "char *" ^ s
| VoidType(s) -> "void " ^ s

let convert_data_type = function
    | "integer" -> "int"
    | "float" -> "float"
    | "boolean" -> "bool"
    | "char" -> "char"
    | "string" -> "char *"
    | _ -> ""

let get_data_type = function
    IntType(s) -> "int"
| FloatType(s) -> "float"
| CharType(s) -> "char"
| StrType(s) -> "char *"
| VoidType(s) -> "void"

let id_of_data_type = function
    IntType(s) -> s
    FloatType(s) -> s
    BoolType(s) -> s
    CharType(s) -> s
    StrType(s) -> s
    VoidType(s) -> s

let rec string_of_literal = function
    Integer(i) -> string_of_int i
    Float(f) -> string_of_float f
    Boolean(b) -> string_of_bool b
    Char(c) -> "" ^ Char.escaped c ^ ""
    String(s) -> "\" ^ s ^ "\"

let type_of_literal = function
    Integer(i) -> "int"
    Float(f) -> "float"
| Boolean(b) -> "bool"
| Char(c) -> "char"
| String(s) -> "char *"

let is_literal = function
    Literal(l) -> true
| _ -> false

let get_literal_type = function
    Literal(l) -> type_of_literal(l)
| _ -> ""

let is_assignment = function
    Assign(a, b) -> true
| _ -> false

(* keep track of assignments
in pfor. REMOVE THE REFERENCE VARIABLES *)

let assignments_in_pfor = ref ""
let defs_in_pfor = ref ""
let globals_in_pfor_assignments = ref []
let globals_init_values = ref []

let generate_code (vars, funcs) env =
    let rec append_env env = function
        [] -> []
    | hd::tl -> (env, hd)::append_env env tl

    in let env_vars = append_env env vars in

    let global_inits =
        ignore (List.map (fun (env, globals) ->
            let data_type = fst globals in
            let var_id = id_of_data_type data_type in
            let (global_type, ref_count) = try Semantic_checker.StringMap.find
                var_id env.Semantic_checker.globals_map with Not_found -> (var_id, 0) in
            if(ref_count <> 0) then (
                match data_type with
                IntType(s) -> globals_init_values := (s,string_of_literal (snd globals))::!globals_init_values;
                FloatType(s) -> globals_init_values := (s,string_of_literal (snd globals))::!globals_init_values;

```
BoolType(s) -> globals_init_values := (s, string_of_literal (snd globals))::!globals_init_values;
CharType(s) -> globals_init_values := (s, "'", ^ string_of_literal (snd globals) ^ "'")::!globals_init_values;
StrType(s) -> globals_init_values := (s, string_of_literal (snd globals))::!globals_init_values;
VoidType(s) -> ("")
) else ("")
) env_vars);

in let rec convert_expr env is_pfor is_replace myA myB = function
Id(x) -> if is_replace && x = myA then
  myB
else
  if is_pfor then (
    let (var_type, ref_count) = try Semantic_checker.StringMap.find x
env.Semantic_checker.globals_map with Not_found -> ("", 0) in
    if(var_type = "") then x else (      
      (if not (List.mem_assoc x !globals_in_pfor_assignments) then (        
        globals_in_pfor_assignments := (x, var_type) :: !globals_in_pfor_assignments
      ) else ());
      "temp_" ^ x
    )
  ) else x
| Literal(l) -> string_of_literal l
| Paren(a) -> "(" ^ convert_expr env is_pfor is_replace myA myB a ^ "")"
| Binop(a, Add, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "+" ^ convert_expr env is_pfor
is_replace myA myB b)
| Binop(a, Sub, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "-" ^ convert_expr env is_pfor
is_replace myA myB b)
| Binop(a, Mult, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "*" ^ convert_expr env is_pfor
is_replace myA myB b)
| Binop(a, Div, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "/" ^ convert_expr env is_pfor
is_replace myA myB b)
| Binop(a, Mod, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "%" ^ convert_expr env is_pfor
is_replace myA myB b)
| Binop(a, And, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "&&" ^ convert_expr env is_pfor
is_replace myA myB b)
| Binop(a, Or, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "||" ^ convert_expr env is_pfor
is_replace myA myB b)
| Binop(a, Equal, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "==" ^ convert_expr env is_pfor is_replace myA myB b)
| Binop(a, Neq, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "!=" ^ convert_expr env is_pfor is_replace myA myB b)
| Binop(a, Less, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "<" ^ convert_expr env is_pfor is_replace myA myB b)
| Binop(a, Leq, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "\leq" ^ convert_expr env is_pfor is_replace myA myB b)
| Binop(a, Greater, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "\geq" ^ convert_expr env is_pfor is_replace myA myB b)
| Binop(a, Geq, b) -> (convert_expr env is_pfor is_replace myA myB a ^ "\geq" ^ convert_expr env is_pfor is_replace myA myB b)
| Not(e) -> "!" ^ convert_expr env is_pfor is_replace myA myB e
| Pp(id) -> (id ^ "++")
| Mm(id) -> (id ^ "--")
| Assign(a, b) -> if is_pfor then (assignments_in_pfor := ![assignments_in_pfor]

|   " pthread_mutex_lock(&pfor_global_lock);\n" ^
|   " " ^ (convert_expr env false is_replace myA myB a ^ "=" ^
|     (convert_expr env false true "i" (convert_expr env is_pfor false myA myB
|     a) b) ^ ";\n") ^
|   " " ^ "pthread_mutex_unlock(&pfor_global_lock);\n";
|   (convert_expr env is_pfor is_replace myA myB a ^ "=" ^ convert_expr env is_pfor false myA
|   myB b)
| ) else
|   (convert_expr env is_pfor is_replace myA myB a ^ "=" ^ convert_expr env is_pfor is_replace
|   myA myB b)
| Call(s1, al) -> if (s1 = "printf") then

|   ("printf" ^ "(" ^
|     ((convert_expr env is_pfor is_replace myA myB (List.hd al) ^
|     (List.fold_left (fun acc x -> acc ^ "," ^ (convert_expr env is_pfor is_replace myA myB x)) "")
|     (List.tl al)))
|   ^ ")")
| else (s1 ^ "(" ^ String.concat ","
|     (List.map (fun e -> convert_expr env is_pfor is_replace myA myB e ) al) ^ "))")
| Noexpr -> ""

in let rec indent n =
if n = 1 then
  " "
else
  " ^indent(n-1)

in let build_args (a,n,num_indent,env,is_pfor) =
  let list_size = List.length a in
  let lit_counter = Array.make 1 0 in
  if list_size > 0 then
    indent num_indent ^ "void *args_thread_" ^ n ^ "[" ^ string_of_int list_size ^ "]":\n" ^
    (fst (List.fold_left (fun (acc,k) x -> (acc ^
        (if is_literal(x) then
          indent num_indent ^ get_literal_type x ^ " arg_lit_" ^ n ^ " _" ^
          (string_of_int k) ^ " = " ^ convert_expr env is_pfor false "x" "y" x ^ ";\n"
        else
          ";",k+1)) (",",0) a) ^
    (fst (List.fold_left (fun (acc,k) x -> (acc ^
        indent num_indent ^ "args_thread_" ^ n ^ "[" ^ (string_of_int k) ^ "] =
        (void *)&" ^
        (if is_literal(x) then
          (lit_counter.(0) <- lit_counter.(0)+1;
          "arg_lit_" ^ n ^ " _" ^ (string_of_int (lit_counter.(0)-1)))
        else
          convert_expr env is_pfor false "x" "y" x)
        ^ ";\n",k+1)) (",",0) a))
  else
    ""
  in

in let get_func_formals_list fname_id env =
  List.fold_left (fun arg_list (dt_func, arg) ->
    if(dt_func = fname_id) then
      arg :: arg_list
    else
      arg_list
  ) [] env.Semantic_checker.formals_list

in let get_func_locals_list fname_id env =
  Semantic_checker.VarMap.fold (fun (my_fun, var_id) (var_type, ref) var_list ->
if(my_fun = fname_id) then {
    let (var_type, ref_count) = try Semantic_checker.VarMap.find
        (fname_id, var_id) env.Semantic_checker.locals_map with Not_found -> (var_id, 0) in
    if(ref_count <> 0) then (
        (var_id, var_type) :: var_list
    ) else var_list
) else var_list
} env.Semantic_checker.locals_map []

in let rec build_vars (f,n,counter,s,k,env) =
    let test = Array.make 1 0 in
    let data_type = Array.make 1 "" in
    let var_type = List.iter (fun arg ->
        if counter-n = test.(0) then (data_type.(0) <<= (get_data_type arg);
            test.(0) <<= test.(0)+1)
    ) (List.rev (get_func_formals_list f env)); data_type.(0) in
    let my_var = (indent 1) ^ var_type ^ " var" ^ (string_of_int (n-1)) ^ ";arg_list[" ^
        (string_of_int (n-1)) ^ "]])in in
    if n = 1 then
        (1,my_var ^ s)
    else
        build_vars (f,n-1,counter,my_var ^ s,k+1,env)

in let build_thread_func (f,a,n,env) =
    let list_size = List.length a in
    "void " ^ n ^ "(void *args)\n"
    if list_size > 0 then
        (indent 1) ^"void **arg_list = (void **)args;\n"
        (snd (build_vars (f,list_size,list_size,"",0,env)))
    else
        ""
(indent 1) ^ f ^ "(" ^
    if list_size > 0 then
        ("var0" ^ (fst (List.fold_left (fun (acc, k) x -> (acc ^ ", var" ^ (string_of_int k), k+1)) ("", 1) (List.tl a)))

64
in let build_pfor (f,n,i,l,c,remove,num_indent,env) =
  let num_threads = n in
  let init = i in
  let limit = l in
  let k = (string_of_int c) in
  let locals_list = List.rev (get_func.locals_list f env) in
  indent num_indent ^ "int num_threads_" ^ k ^ " = " ^ num_threads ^ " ; " ^ \n ^ indent num_indent ^ "int pfor_init_" ^ k ^ " = " ^ init ^ " ; " ^ \n ^ indent num_indent ^ "int pfor_limit_" ^ k ^ " = " ^ limit ^ " ; " ^ \n ^ indent num_indent ^ "thread_t pfor_threads_" ^ k ^ ":[num_threads_" ^ k ^ "] ; " ^ \n ^ indent num_indent ^ "int pfor_uppers_" ^ k ^ " [num_threads_" ^ k ^ "] ; " ^ \n ^ indent num_indent ^ "int pfor_lowers_" ^ k ^ " [num_threads_" ^ k ^ "] ; " ^ \n ^ indent num_indent ^ "void * pfor_args_" ^ k ^ " [num_threads_" ^ k ^ "] [3 ] ; " ^ \n ^ indent num_indent ^ "string_of_int (List.length locals_list) ; " ^ \n ^ (fst (List.fold_left (fun (acc, count) (var_id, var_type) ->
    if not (List.mem var_id remove) then (acc ^ indent num_indent ^ "pfor_local_vars_" ^ k ^ " [" ^ string_of_int count ^ "] = (void *)& " ^ var_id ^ "); ; " ^ \n ^ count+1)
  ) else
    (acc, count)
  ) ("",0) locals_list) ^
  indent num_indent ^ "int pfor_i_" ^ k ^ " = 0 ; pfor_i_" ^ k ^ "< num_threads_" ^ k ^ " ; pfor_i_" ^ k ^ "+" ^\n ^ indent num_indent+1 ^ "pfor_uppers_" ^ k ^ " [pfor_i_" ^ k ^ "] = pfor_init_" ^ k ^ " + ( pfor_init_" ^ k ^ "+ pfor_limit_" ^ k ^ ") / num_threads_" ^ k ^ ") ( pfor_i_" ^ k ^ " ^ + 1 ) + ( pfor_init_" ^ k ^ " + pfor_limit_" ^ k ^ ") / num_threads_" ^ k ^ " ^ + " ^\n ^ indent num_indent+1 ^ "pfor_lowers_" ^ k ^ " [pfor_i_" ^ k ^ "] = pfor_init_" ^ k ^ " + ( pfor_init_" ^ k ^ "+ pfor_limit_" ^ k ^ ") / num_threads_" ^ k ^ " ^ + " ^\n ^ indent num_indent+1 ^ "pfor_i_" ^ k ^ "] = pfor_init_" ^ k ^ " + ( pfor_init_" ^ k ^ "+ pfor_limit_" ^ k ^ " ^ + " ^\n ^ indent num_indent+1 ^ "pfor_args_" ^ k ^ " [pfor_i_" ^ k ^ "] [0 ] = ( void *) & pfor_lowers_" ^ k ^ " ^ [pfor_i_" ^ k ^ "] ; " ^ \n ^ indent num_indent+1 ^ "pfor_args_" ^ k ^ " [pfor_i_" ^ k ^ "] [1 ] = ( void *) & pfor_uppers_" ^ k ^ " ^ [pfor_i_" ^ k ^ "] ; " ^ \n ^
indent (num_indent+1) ^ "pfor_args_" ^ k ^ "[pfor_i_" ^ k ^ "])[2] = (void *)&pfor_local_vars_" ^ k ^ ";\n" ^

indent num_indent ^ 
"
indent num_indent ^ "for(pfor_i_" ^ k ^ "=0;pfor_i_" ^ k ^ "<num_threads_" ^ k ^ ";pfor_i_" ^ k ^ 
"++)(\n" ^

indent (num_indent+1) ^ "pthread_create(&pfor_threads_" ^ k ^ "[pfor_i_" ^ k ^ "].NULL,thread_" ^ k ^ 
",(void *)&pfor_args_" ^ k ^ "[pfor_i_" ^ k ^ ");\n" ^

indent num_indent ^ 
"
indent num_indent ^ "for(pfor_i_" ^ k ^ "=0;pfor_i_" ^ k ^ "<num_threads_" ^ k ^ ";pfor_i_" ^ k ^ 
"++)(\n" ^

indent (num_indent+1) ^ "pthread_join(pfor_threads_" ^ k ^ "[pfor_i_" ^ k ^ ");\n" ^

indent num_indent ^ "\n"

(* myenv stores counters for the
number of locks, threads, and
pfors -- in that order *)

in let myenv = Array.make 3 0
in let thread_funcs = Array.make 2 []
in let remove = Array.make 1 []

in let rec convert_stmt num_indent env currf is_pfor is_replace myA myB = function
  Expr(e) -> indent num_indent ^ convert_expr env is_pfor is_replace myA myB e ^ ";\n"
| Declare(e) -> let data_type = e in
  let var_id = id_of_data_type data_type in
  let (var_type, ref_count) = try Semantic_checker.VarMap.find (currf,var_id) env.Semantic_checker.locals_map with Not_found -> (var_id, 0) in
  if(ref_count <> 0) then (if is_pfor then (remove.(0) <- var_id::remove.(0); ()
    ) else ()
    )
    else ("")
| DeclareAssign(a, b) -> let data_type = a in
  let var_id = id_of_data_type data_type in
  let (var_type, ref_count) = try Semantic_checker.VarMap.find (currf,var_id) env.Semantic_checker.locals_map with Not_found -> (var_id, 0) in
  if(ref_count <> 0) then (if is_pfor then (}
remove.(0) <- var_id::remove.(0); ()
)
else ();
indent num_indent ^ string_of_data_type a ^ "=" ^ convert_expr env is_pfor is_replace
myA myB b ^ ";\n"
)
else ("")
| Return(e) -> indent num_indent ^ "return " ^ convert_expr env is_pfor is_replace myA myB e ^ ";\n"
| Break(e) -> indent num_indent ^ "break;\n"
| Continue(e) -> indent num_indent ^ "continue;\n"
| Block(s) -> if (List.length s = 0) then "" else "{\n" ^ (List.fold_left (fun acc x ->
acc ^ convert_stmt (num_indent + 1) env currf is_pfor is_replace myA myB x) "" s) ^ (indent
(num_indent)) ^ "}\n"
| If(e, s, n) -> let else_block = convert_stmt num_indent env currf is_pfor is_replace myA myB n in
if (else_block = "") then (indent num_indent) ^ "if(" ^ convert_expr env is_pfor is_replace myA
myB e ^ ")" ^
convert_stmt num_indent env currf is_pfor is_replace myA myB s
else (indent num_indent) ^ "if(" ^ convert_expr env is_pfor is_replace myA myB e ^ ")" ^
convert_stmt num_indent env currf is_pfor is_replace myA myB s ^(indent num_indent) ^ "else"
else_block
| For(i, c, u, s) -> (indent num_indent) ^ "for(" ^ convert_expr env is_pfor is_replace myA myB i ^ ";" ^
convert_expr env is_pfor is_replace myA myB c ^ ";" ^
convert_expr env is_pfor is_replace myA myB u ^ ");" ^
convert_stmt num_indent env currf is_pfor is_replace myA myB s ^
"pthread_mutex_unlock(&m" ^ lock ^ ");\n"
| While(e, s) -> (indent num_indent) ^ "while(" ^ convert_expr env is_pfor is_replace myA myB e ^ ");" ^
convert_stmt num_indent env currf is_pfor is_replace myA myB s
| Lock(s) -> myenv.(0) <- myenv.(0)+1; let lock = string_of_int myenv.(0) in
(indent num_indent) ^ "pthread_mutex_lock(&m" ^ lock ^ ");\n" ^ convert_stmt num_indent env
currf is_pfor is_replace myA myB s ^
"pthread_mutex_unlock(&m" ^ lock ^ ");\n"
| Barrier(e) -> (indent num_indent) ^ "int thread_counter;\n" ^ (indent num_indent) ^
"for(thread_counter=0;thread_counter<" ^
string_of_int myenv.(1) ^ ";thread_counter++){\n" ^ (indent (num_indent+1)) ^
"pthread_join(threads[thread_counter],NULL);\n" ^ (indent num_indent) ^ ")\n"
| Spawn(Call(id,args)) -> myenv.(1) <- myenv.(1)+1;
(thread_func.0) <- (build_thread func (id,args,string_of_int
(myenv.(1)-1),env)::thread_func.0);
let thread = string_of_int (myenv.(1)-1) in
(build_args (args,thread,num_indent,env,is_pfor)) ^
indent num_indent ^ "pthread_create(&threads[" ^ thread ^ "],NULL,thread_" ^ string_of_int
let build_pfor_func stmt n env =
  assignments_in_pfor := ""
  globals_in_pfor_assignments := []
  let counter = convert_expr env is_pfor is_replace myA myB k in
  (indent 1) ^ "pthread_mutex_t pfor_global_lock=PTHREAD_MUTEX_INITIALIZER;\n" ^
  (indent 1) ^ "void **arguments = (void **)args;\n" ^
  (indent 1) ^ "int lower = *((int *)arguments[0]);\n" ^
  (indent 1) ^ "int upper = *((int *)arguments[1]);\n" ^
  (let locals = (List.fold_left (fun (acc, count) (var_id, var_type) ->
    if not (List.mem var_id remove.(!globals_init_values)) then (acc ^ indent num_indent ^ (convert_data_type var_type) ^
      " var_id = \" \"
    ) else (acc, count)
  ) ("",0) (List.rev (get_func_locals_list currf env)) in
  if snd locals > 0 then ( (indent 1) ^ "void **thread_0_local_vars = (void **)arguments[2];\n" ^
    (fst locals)
  ) else "") ^
  (indent 1) ^ "for(\" \"counter = new; \" \"counter <\" \"upper; \" \"counter \" =\" \"++\") ^
  convert_stmt 1 env currf true is_replace myA myB stmt in
  let get_global_init var_id =
    List.fold_left (fun acc (id,value) -> if var_id = id then value else acc) "" !globals_init_values
  in
  myenv.(1) <- myenv.(1)+1;
  let pfor_func = (build_pfor_func s (string_of_int (myenv.(1)-1),env)) in
  let globals_in_assignments = List.fold_left (fun acc (var_id, var_type) ->
    acc ^ " var_id = \" \"
    get_global_init var_id ^ ";\n" ^
  ) "" !globals_in_pfor_assignments in
  (thread_funcs.(0) <- ("void *thread_\" \"string_of_int (myenv.(1)-1) ^ "(void *args)\n" ^
    globals_in_assignments ^
  )
in let rec locks (n,l) =
  if n = 0 then
    (0, l)
  else
    locks (n-1, ("pthread_mutex_t m" ^ string_of_int (n) ^ "=PTHREAD_MUTEX_INITIALIZER;\n") :: l)

in let rec threads (n,l) =
  if n = 0 then
    (0, l)
  else
    threads (n-1, ("void *thread_" ^ string_of_int (n-1) ^ "(void *args);\n") :: l)

in let convert Globals (env, globals) =
  let data_type = fst globals in
  let var_id = id_of_data_type data_type in
  let (global_type, ref_count) = try Semantic_checker.StringMap.find var_id env.
  Semantic_checker.globals_map with Not_found -> (var_id, 0) in
  if ref_count <> 0 then (
    match data_type with
    | IntType(s) -> "int " ^ s ^ " = " ^ string_of_literal (snd globals) ^ ";"
    | FloatType(s) -> "float " ^ s ^ " = " ^ string_of_literal (snd globals) ^ ";"
    | BoolType(s) -> "bool " ^ s ^ " = " ^ string_of_literal ( snd globals ) ^ ";"
    | CharType(s) -> "char " ^ s ^ " = " ^ string_of Literal ( snd globals ) ^ "\n"
    | StrType(s) -> "char " ^ s ^ "[" ^
                  string_of_int (String.length (string_of_literal (snd globals)) - 1)
                  ^ "]" ^ " = " ^ string_of literal (snd globals) ^ "\n"
    | VoidType(s) -> ""
  ) else ("")
in let string_of_fdecl (env,fdecl) = 
    let data_type = fdecl.fname in 
    let var_id = id_of_data_type data_type in 
    let (func_type, ref_count) = try Semantic_checker.StringMap.find var_id env. 
    Semantic_checker.functions_map with Not_found -> (var_id, 0) in 
    if(ref_count <> 0) then ( 
        string_of_data_type fdecl.fname ^ "(" ^ 
        String.concat "," (List.map string_of_data_type fdecl.formals) ^ 
        ")\n" ^ 
        String.concat "" (List.map (convert_stmt 1 env (id_of_data_type fdecl.fname) false false "x" "y") fdecl.body) ^ 
        "\n" 
    ) else ("") 

in let rec append_env env = function 
    [] -> [] 
    | hd::tl -> (env, hd)::append_env env tl 

in let env_vars = append_env env vars in 
let env_funcs = append_env env funcis in 
""/ Code Generated from SMPL */\n"^ includes ^ 
String.concat "\n" (List.map convert_globals env_vars) ^ global_inits ^ "\n\n"^ 
String.concat "\n" (List.map (fun (tenv,fdecl) -> 
        let name = string_of_data_type fdecl.fname in 
        if name <> "int main" then ( 
            let data_type = fdecl.fname in 
            let var_id = id_of_data_type data_type in 
            let (func_type, ref_count) = try Semantic_checker.StringMap.find var_id env. 
            Semantic_checker.functions_map with Not_found -> (var_id, 0) in 
            if(ref_count <> 0) then ( 
                string_of_data_type fdecl.fname ^ "(" ^ String.concat "," (List.map string_of_data_type 
                fdecl.formals) ^ 
                ");\n" 
            ) else ("") 
        ) else ("") 
    ) env_funcs) ^ 
    (List.fold_left (fun acc x -> acc ^ x) "" (snd (locks (myenv.(0), [])))) ^ "\n"^ 
    (List.fold_left (fun acc x -> acc ^ x) "" (snd (threads (myenv.(1), [])))) ^ "\n"^
(if myenv.(1) > 0 then "pthread_t threads[" ^ string_of_int myenv.(1) ^ "]:\ln\ln" else ") ^
(List.fold_left (fun acc x -> x ^ acc) "" thread_funcs.(0) ^
String.concat "\ln" (List.map string_of_fdecl env_funcs))

smplc.ml

# Primary Author: Devashi Tandon

type action = DumpAst | CodeGen | SyntaxCheck | InvalidArg

let help_msg = "NAME: \ntsmpl - Compiler for generating C code for SMPL" ^
" - SMall Paraller Language\nARGUMENTS:\n-a code-input Print the parse" ^
" results of the input SMPL code \n-c code-input Generate the C" ^
" code equivalent to the input SMPL code\n-s code-input Just check" ^
" the syntax of the SMPL code\n code-input can be one of the" ^
" following:\n-t Take input from terminal\n<f ile-name> Take input" ^
" from file\nEXAMPLES:\n\nt/smplc -a -t : Take code input from" ^
" terminal and print the parse results.\n\nt/smplc -c hello.smpl >" ^
" hello.c: Compile the SMPL code in hello.smpl and write the generated C" ^
" code to hello.c\nAUTHORS:\nWritten" ^
" by Ajay Siva Santosh, Andrei Papancea, Devashi Tandon\n"

let process_program lexbuf action =
let expr = Parser.program Scanner.token lexbuf in
let verified_env = Semantic_checker.validate_program expr in
if (verified_env.Semantic_checker.valid_syntax) then
  match action with
  | DumpAst ->
    let result = Printast.parse_program expr in
    print_string result
  | CodeGen ->
    let result = Compile.generate_code (expr) verified_env in
    print_string result
  | SyntaxCheck -> print_string "Validation is complete\n"
  | InvalidArg -> print_string help_msg

let _ =
if Array.length Sys.argv <> 3 then (  
    print_string help_msg  
) else (  
    let action = try List.assoc Sys.argv.(1)  
    [ ("-a", DumpAst); ("-c", CodeGen); ("-s", SyntaxCheck) ]  
    with Not_found -> InvalidArg in  
    match action with  
    InvalidArg -> print_string help_msg  
    | _ ->  
    let src_file = Sys.argv.(2) in  
    if(src_file = ".t") then  
        let lexbuf = Lexing.from_channel stdin in  
        process_program lexbuf action  
    else  
        let lexbuf = Lexing.from_channel (open_in src_file) in  
        process_program lexbuf action  
)
#!/bin/sh

SMPL="../smplc -s 

files="tests/semantic_pass_cases/*".smpl"

CleanUp() {
    rm test.out test.null
}

GetError() {
    eval "$SMPL $1" 2> test.out 1> test.null
    wc -l test.out | awk '{print $1}'
}

for file in $files
do
    errors=$(GetError $file)
    GetError $file
    if [ $errors -eq 0 ]
    then
        echo "Test: "$file "passed."
    else
        echo "$file " failed to pass."
        exit 1
    fi
done
CleanUp

echo "All tests passed."

run-execution.sh

#!/bin/sh

# Primary Author: Ajay Challa
# Primary Author: Ajay Challa
SMPL="../smplc -c"

files="tests/execution_cases/*.smpl"

CleanUp() {
    rm test.out test.null test.c result.out a.out
}

GetError() {
    eval "$SMPL $1" 2> test.out 1> test.null
    wc -l test.out | awk '{print $1}'
}

GenerateCode() {
    $SMPL $1 > test.c
gcc -pthread test.c
./a.out > result.out
    BASEFILE=`echo $1 | sed -e "s/smpl/out/g"`
diff -b "$BASEFILE" result.out 2>&1 || {
        echo "FAILED $BASEFILE differs for $1" 1>&2
        echo $1 " failed to pass."
        exit 1
    }
}

for file in $files
do
    errors=$(GetError $file)
    GetError $file
    if [ $errors -eq 0 ]
    then
        GenerateCode $file
        echo "Test: "$file " passed."
    else
        echo $file " failed to pass."
        exit 1
    fi
done
Demo Test Cases

pfor-test1.smpl

```c
# Primary Author: Ajay Challa
main()
{
    int i;

    pfor (10;i;0;100)
    {
        printf("%d\n",i);
    }
    printf("%d\n",42);
}
```

pfor-test2.smpl

```c
# Primary Author: Ajay Challa
int sum = 0;

int isPrime(int n){
    int limit = n/2;
    int i = 0;
    for(i=2; i<=limit; i++)
        if(n%i == 0)
            return 0;
    return 1;
}

int main(){
```
```c
int i;
int n = 1000000;
pfor(8; i; 1; n){
    if(isPrime(i))
        sum = sum+i;
}
printf("The sum of the first 1M primes is %d\n",sum);
}
```

**lock-test1.smpl**
```
#include <stdio.h>

#define LOCK_FILE "lock.lock"

int main()
{
    int i;
    int j=0;

    pfor (5;i;10;100)
    {
        printf("%d\n",i);
        lock
        {
            j=j+1;
        }
        printf("%d\n",42);
    }
    return 0;
}
```

**dead-code.smpl**
```
#include <stdio.h>

#define LOCK_FILE "lock.lock"

int unused1 = 0;
char unused2 = 'c';
```

dead-code2.smpl

# Primary Author: Ajay Challa

```
int unused1 = 0;
char unused2 = 'c';
float used1 = 2.0;
void main()
{
    used1 = 4.0;
    printf("%f \n", used1);
}

func1_unused()
{
    unused2 = 'd';
    func2_unused(unused2);
}

func2_unused(char var)
{
    printf("%c", var);
    func1_unused();
}
```

```
func2_unused(char var)
{
    unused1 = 5;
    printf("%c", var);
}

used-code.smpl

# Primary Author: Ajay Challa
int unused1 = 0;
char unused2 = 'c';
float used1 = 2.0;
void main()
{
    used1 = 4.0;
    printf("%f \n", used1);
    func1_unused();
}

func1_unused()
{
    unused2 = 'd';
    func2_unused(unused2);
}

func2_unused(char var)
{
    printf("%c", var);
    func1_unused();
}

Execution Test Cases
**barrier-test1.smpl**

```c
# Primary Author: Ajay Challa
int a=10;
main()
{
    int i=0;
    spawn fun();
    barrier;
    printf("The value of i is %d \n", i);
}
fun()
{
    int i=0;
    while(i<100)
    {
        i=i+1;
    }
    printf("Done");
}
```

**test-for1.smpl**

```c
# Primary Author: Ajay Challa
main()
{
    int i;
    for (i = 0 ; i < 5 ; i = i + 1) {
        printf("%d \n",i);
    }
    printf("%d \n",42);
}
```

**test-func1.smpl**

```c
# Primary Author: Ajay Challa
main()
{
}
int a=0;
a = add(39,3);
printf("%d \n",a);
}

int add(int a,int b)
{
    return(a+b);
}

test-func2.smpl

# Primary Author: Ajay Challa
int fun(int x, int y)
{
    return 0;
}

main()
{
    int i;
i = 1;
    fun(i = 2, i = i+1);
    printf("%d \n",i);
}

test-func3.smpl

# Primary Author: Ajay Challa
printem(int a, int b, int c, int d)
{
    printf("%d \n", a);
    printf("%d \n", b);
    printf("%d \n", c);
    printf("%d \n", d);
}

main()
{
    printem(42, 17, 192, 8);
}

test-func4.smpl

# Primary Author: Ajay Challa
int a=0; /* Global variable */

int inca() { a = a + 1; return a; } /* Increment a; return its new value */

int add2(int x, int y) { return x + y; }

main()
{
    a = 0;
    printf("%d \n", add2(inca(), a));
}

int a=0;
int b=0;

printa()
{
    printf("%d \n", a);
}

printb()
{
printf("%d \n", b);
}

incab()
{
    a = a + 1;
    b = b + 1;
}

main()
{
    a = 42;
    b = 21;
    printa();
    printb();
    incab();
    printa();
    printb();
}

test-gcd.smpl

# Primary Author: Ajay Challa
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}

main()
{
    printf("%d \n", gcd(2, 14));
    printf("%d \n", gcd(3, 15));
```c
printf("%d \n",gcd(99, 121));
}

test-global1.smpl

# Primary Author: Ajay Challa
int a=0;
int b=0;

printa()
{
    printf("%d \n",a);
}

printb()
{
    printf("%d \n",b);
}

incab()
{
    a = a + 1;
    b = b + 1;
}

main()
{
    a = 42;
    b = 21;
    printa();
    printb();
    incab();
    printa();
    printb();
}
```
test-hello.smpl

```c
# Primary Author: Ajay Challa
main()
{
    printf("%d \n", 42);
    printf("%d \n", 71);
    printf("%d \n", 1);
}
```

test-if1.smpl

```c
# Primary Author: Ajay Challa
main()
{
    if (1) printf("%d \n", 42);
    printf("%d \n", 17);
}
```

test-if2.smpl

```c
# Primary Author: Ajay Challa
main()
{
    if (1) printf("%d \n", 42); else printf("%d \n", 8);
    printf("%d \n", 17);
}
```

test-if3.smpl

```c
# Primary Author: Ajay Challa
```
main()
{
    if (0) printf("%d \n",42);
    printf("%d \n",17);
}

test-ops1.smpl

# Primary Author: Ajay Challa
main()
{
    printf("%d \n",1 + 2);
    printf("%d \n",1 - 2);
    printf("%d \n",1 * 2);
    printf("%d \n",100 / 2);
    printf("%d \n",99);
    printf("%d \n",1 == 2);
    printf("%d \n",1 == 1);
    printf("%d \n",99);
    printf("%d \n",1 != 2);
    printf("%d \n",1 != 1);
    printf("%d \n",99);
    printf("%d \n",1 < 2);
    printf("%d \n",2 < 1);
    printf("%d \n",99);
    printf("%d \n",1 <= 2);
    printf("%d \n",1 <= 1);
    printf("%d \n",2 <= 1);
    printf("%d \n",99);
    printf("%d \n",1 > 2);
    printf("%d \n",2 > 1);
    printf("%d \n",99);
    printf("%d \n",1 >= 2);
    printf("%d \n",1 >= 1);
    printf("%d \n",2 >= 1);
    printf("%d \n",99);
}
# Primary Author: Ajay Challa

### test-stmts1.smpl

```c
int foo(int a, int b) {
    int i;
    if (a)
        return b + 3;
    else
        for (i = 0 ; i < 5 ; i = i + 1)
            b = b + 5;
    return b;
}

main() {
    printf("%d \n",foo(1,42));
    printf("%d \n",foo(0,37));
}
```

### test-var1.smpl

```c
# Primary Author: Ajay Challa
main()
{
    int a;
    a = 42;
    printf("%d \n",a);
}
```

### test-var2.smpl

```c
# Primary Author: Ajay Challa
main()
```
```c
{ int a;
 int b;
 a = 42;
 b = 57;
 printf("%d \n", a + b * 3);
}
```
while (i > 0) {
    printf("%d \n", i);
    i = i - 1;
}
printf("%d \n", 42);

hello-world.smpl

# Primary Author: Ajay Challa
int itest = 13;
float ftest = 1.0;
char ctest = 'c';
boolean btest = true;
string stest = "I am working fine now\n";

int main(int a, int b)
{
    /* test program */
    printf("%s", stest);
}

lock-test2.smpl

# Primary Author: Ajay Challa
int sum=0;
int add_to_sum(int val){
    lock sum = sum+val;
}

int main(){
    int i;
    pfor(4;i=0;10000){
        add_to_sum(i);
    }
```c
printf("The sum is %d\n",sum);
}

pfor-test2.smpl

# Primary Author: Ajay Challa
int sum = 0;

int isPrime(int n){
    int limit = n/2;
    int i = 0;
    for(i=2; i<=limit; i++)
        if(n%i == 0)
            return 0;
    return 1;
}

int main(){
    int i;
    int n = 10000;

    pfor(8; i; 1; n){
        if(isPrime(i))
            sum = sum+i;
    }

    printf("The sum of the first 1M primes is %d\n",sum);
}

spawn-test1.smpl

# Primary Author: Ajay Challa
func()
{
    int i;
```
for(i=0;i<100;i=i+1)
{
    printf("%d \n",i+100);
}
}
main()
{ spawn func();
}

test-arith1.smpl

# Primary Author: Ajay Challa
main()
{
    printf("%d\n",39 + 3);
}

test-arith2.smpl

# Primary Author: Ajay Challa
main()
{
    printf("%d\n",1 + 2 * 3 + 4);
}

test-arith3.smpl

# Primary Author: Ajay Challa
int a=0; /* Global variable */

int inca() { a = a + 1; return a; } /* Increment a; return its new value */
main() {
    a = 42;    /* Initialize a */
    printf("%d \n", inca() + a);
}

test-arith4.smpl

# Primary Author: Ajay Challa
int g = 0;

main() {
    int l;
    l = 1;
    printf("%d \n", l);
    g = 3;
    printf("%d \n", g);
    l = 5;
    printf("%d \n", l + 100);
    g = 7;
    printf("%d \n", g + 100);
}

test-fib.smpl

# Primary Author: Ajay Challa
int fib(int x)
{
    if (x < 2) return 1;
    return fib(x-1) + fib(x-2);
}

main()
{
    printf("%d \n", fib(0));
    printf("%d \n", fib(1));
Semantic Pass Cases

Declaration.smpl

# Primary Author: Ajay Challa
int a=10;
char c='a';
boolean b=true;
float f=10.0;

void main()
{
}

Function.smpl

# Primary Author: Ajay Challa
int a=10;
char ch='a';
int b=5;

fun()
{ int c=b+c;
}
main()
{ fun();
}
# Test1.smpl

```c
# Primary Author: Ajay Challa
int a=1;
int k=2;
int c=3;
int b=12;
char d='a';

fun()
{
    int m;
    if(a-b){
        c=4;
    } else {
        m=c;
    }
}

main()
{
    if(a-b){
        c=4;
    } else {
        d=c;
    }
    fun();
}
```

# arithmetic.smpl

```c
# Primary Author: Ajay Challa
int a=10;
int b=5;
```
void main()
{
    printf(a+b);
    printf(a-b);
    printf(a/b);
    printf(a*b);
    printf(a%b);
}

barrier-test1.smpl

# Primary Author: Ajay Challa
int a=10;
main()
{
    int i=0;
    spawn fun();
    barrier;
    printf("The value of i is \%d", i);
}
fun()
{
    int i=0;
    while(i<100)
    {
        i=i+1;
    }
    printf("Done");
}

binop.smpl

# Primary Author: Ajay Challa
int a=5;
float f=3.0;
boolean b=false;
char c='k';
string s="hello";

main()
{
    /* binop arithmetic*/
    printf(a+b);
    printf(a+f);
    printf(a+c);
    printf(a/b);
    printf(a/f);
    printf(a/c);
    printf(a-b);
    printf(a-f);
    printf(a-c);
    printf(a*b);
    printf(a-f);
    printf(a-f);
    printf(a-c);

    /*binop logical*/

    boolean l=a>f;
    boolean l1=a<f;
    boolean l2=a>=f;
    boolean l3=a<b;
    boolean l4=a<=b;
    boolean l5=c!=b;
}

break_test.smpl

# Primary Author: Ajay Challa
int count = 0;
int i=0;
void main()
{ for(i=0;i<10;i+1)
{count=count+1;
  if(count==5)
  { printf("breaking");
    break;
  }
  else
  { printf("continuing");
   /* continue; */
  }
}
}

calling_fun.smpl

# Primary Author: Ajay Challa
int a=1;
int b=0;
char c='a';
int k=0;

fun()
{ if(a-b)
{c=4;}
else
{k=c;
 a=a/0;
}
}

int fail=10;

main()
{ if(a-b)
{c=4;
fun();
else
{k=c;}
fun();
}

casting.smpl

# Primary Author: Ajay Challa
int a=10;
char c='c';
float f=10.0;
boolean b=true;

main()
{
    a=c;
    a=b;
    c=a;
    f=a;
    f=b;
    f=c;
    int r=a+b;
}
data_types.smpl

# Primary Author: Ajay Challa
int a=10;
float b=10.0;
boolean flag=true;
string s="hello";

main()
{
}

for-test.smpl

# Primary Author: Ajay Challa
int i=0;

main()
{

for( i=0;i<10;i+1)
{
printf("%d",i);
}

}

hello-world.smpl

# Primary Author: Ajay Challa
int itest = 13;
float ftest = 1.0;
char ctest = 'c';
boolean btest = true;
string stest = "I am working fine nowin";
int main(int a, int b)
{
    /* test program */
    printf("Hello World.. I have arrived\n");
}

lock-test1.smpl

# Primary Author: Ajay Challa
main()
{
    int i;
    int j=0;

    pfor (5;i;10;100)
    {
        printf("%d",i);
        lock
        {
            j=j+1;
        }
    }
    printf("%d",42);
}

logical.smpl

# Primary Author: Ajay Challa
int a = 10;
int b = 5;
boolean b1=true;
boolean b2=false;

void main()
{

}
if(a>b)
{ printf("a is greater than b");
}

if(a<b)
{ printf("a is less than b");
}

if(a<=b)
{ printf("a is less than or equal to b");
}

if(a==b)
{ printf("a is equal to b");
}

if(a!=b)
{ printf("a is not equal to b");
}

if((a!=b))
{ printf("a is equal to b");
}

if(a>=b)
{ printf("a is greater than or equal to b");
}

if(b1 && b2)
{ printf("both are true");
}

if(b1 || b2)
{ printf("one or both are true");
}
main.smpl

```c
# Primary Author: Ajay Challa
int a=10;
boolean b=true;
int k=0;

void main()
{
    k=a+10;
}
```

multiplespawnstest.smpl

```c
# Primary Author: Ajay Challa
int j=5;
int i=4;
fun1()
{while(i<10) printf("%d",i);}
fun2()
{while(j<20) printf("%d",j);}

main(){
    int i=0;
    int j=10;
    int k=20;
    int l=30;
    spawn fun1();
    spawn fun2();
    barrier;
    printf("done \n");
}
```
parantheses.smpl

# Primary Author: Ajay Challa
main()
{
  int Rpar_test=(2*4)+(5*6);
}

pfortest1.smpl

# Primary Author: Ajay Challa
main()
{
  int i;

  pfor (10;i;0;100)
  {
    printf("%d",i);
  }
  printf("%d",42);
}

return_stmt.smpl

# Primary Author: Ajay Challa
int a=1;
int b=2;
int k=3;
int d=12;
char c='a';

int fun()
{  if(a-b)
else
{k=c;}
return(1);
}

int fail=10;

main()
{ if(a­b)
    {c=4;
    fun();}
    else
    {k=c;}
    }

test­prog.smpl

# Primary Author: Ajay Challa
int itest = 13;
float ftest = 1.0;
char ctest = 'c';
boolean btest = true;

funca(int a1, int a2)
{
    printf(a1, a2);
}

int main(int a, float b)
{
    /* test program */
    int i = 0;
    string stest = "I am working fine now\n\n";
    for(i=10;i<20;i++)
threadtest1.smpl

```c
#include <stdio.h>

#define LIMIT 10

int main()
{
    int i = 0;
    int sum = 0;
    int n;
    int t;
    printf("Enter the number of threads t: ");
    lock {
        sum = sum + i;
    }
    printf("The sum of the first %d numbers is %d\n", n, sum);
    return 0;
}
```

while_test.smpl
# Primary Author: Ajay Challa

```c
int i=10;

main()
{

while(i!=0)
{
    printf("%d",i);
    i-1;
}

}
```

## Semantic Fail Cases

### Declaration.smpl

```c
# Primary Author: Ajay Challa

int a=10;
char c='a';
main()
{
    a=7.0;
    c=8.0;
}
```

### keywords.smpl

```c
# Primary Author: Ajay Challa

int printf=1;
char f='a';
main()
```
binop.smpl

```c
# Primary Author: Ajay Challa

string s="hello";
string s1="world";
main()
{  s1=s+s1;
}
```

casting.smpl

```c
# Primary Author: Ajay Challa

int a=5;
boolean b=false;
char c='c';
float f=7.0;

main()
{
    
a=b;
a=c;
a=f;

    b=a;
b=c;
b=f;

    c=a;
c=b;
c=f;
```
fnkeyword.smpl

# Primary Author: Ajay Challa

printf()
{}
main()
{printf();
}

key.smpl

# Primary Author: Ajay Challa

int printf=5;

main()
{}

pfor.smpl

# Primary Author: Ajay Challa

{ pfor(3;i=0;i<10;i+1)
  printf("hello");
}
redeclaration.smpl

```c
# Primary Author: Ajay Challa

int a=10;
char a='a';
main()
{}
```

ref-count.smpl

```c
# Primary Author: Ajay Challa

int itest = 13;
float ftest = 1.0;
char ctest = 'c';
boolean btest = true;

funca(int a, int a2)
{
  int int_funca;
  printf(a, a2);
  /*funcb("abc");*/
}

int main(int a, float b)
{
  /* test program */
  {
    int i = 3;
    int j;
    int k;
    funca(itest, ftest);
    int n = (j+k)*funcb());
  }
```
n = (9+5)*(k+i);
funca(btest, itest);
string stest;
int main_int;
break;
if(i<10 || i>5)
{
    i = 9;
    return ((5+3)*funcd());
} else {
    return funca(1,3);
}
for(i=10;i<20;i=i+1)
{
    printf(itest, ftest, ctest, btest, a, b);
    printf("%s %d\n", stest, 9);
    break;
}
while(j < 100) {
    funce();
}
lock {
    funca(i,j);
    funca(j,k);
    funca(((j+k)*i) < n),(3));
    spawn funca(5,3);
}
barrier;
printf("This is a string\t\n");
}

int funcd(){
    return 5;
}

int funce(){
    return 6;
}

int funce()}
funcb (string abc)
{
    printf("got it");
    funcc();
    funcc();
    /*funca();*/
}
funcc()
{
    return;
}

---

samename.smpl

# Primary Author: Ajay Challa

int fun1=10;

fun1()
{
    int a=10;
    a=a+1;
}

main()
{
    fun1();
}

---

datatameargs.smpl

# Primary Author: Ajay Challa

fn(int a, int a)
{}

main()
{int a=10;
 int b=10;
 fn(a,b);
}

# Primary Author: Ajay Challa

main()
{ fn();
}

# Primary Author: Ajay Challa

fn()
{ int a;
  int b=c+a;
}

main()
{ fn();
}